PROGRAMACIÓN

BESTSELLER INTERNACIONAL

SAMS PUBLISHING

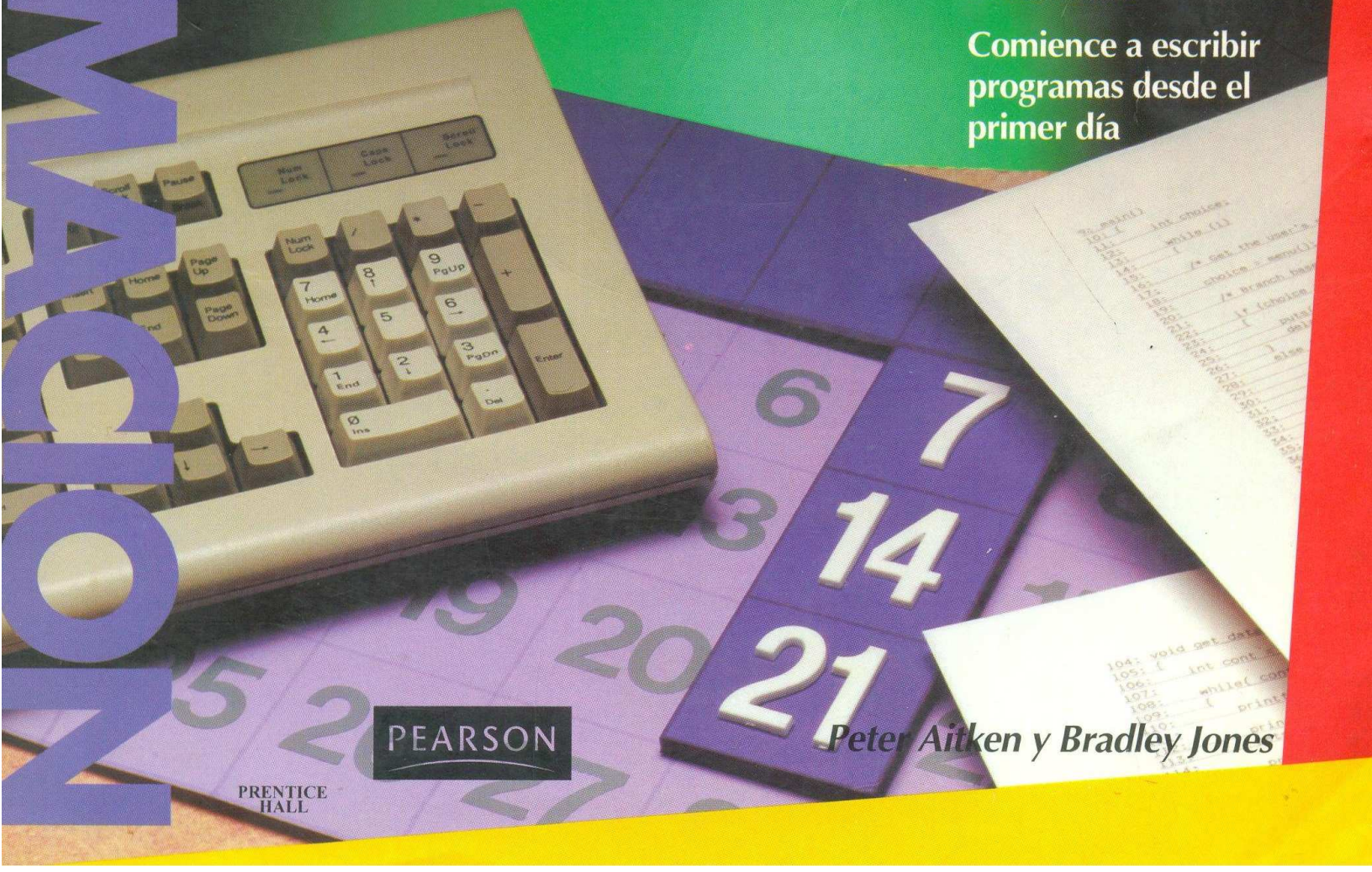
Edición Bestseller

Aprendiendo C en 21 Días

Trata el C de ANSI, y es compatible con todos los compiladores C ANSI

Una tarjeta de consulta instantánea pone la información al alcance de su mano

Comience a escribir programas desde el primer día



Peter Aitken y Bradley Jones

PEARSON

PRENTICE HALL

DIA 1	DIA 2	DIA 3	DIA 4	DIA 5	DIA 6	DIA 7
Comience con un día para acostumbrarse. Instale su compilador y su editor y trabaje con ellos. ¡En sólo tres semanas usted será un experto en C!	Aprenda las partes de su primer programa en C, línea por línea, y la manera de documentar sus programas con líneas de comentarios.	Examine las declaraciones de los diversos tipos de datos y constantes. Los programas de C guardan datos en variables y constantes.	Use los operadores para manipular las expresiones de C y haga su primera prueba de control de programa con el enunciado if.	Llegue al fondo de los programas de C con funciones y programación estructurada. Pase más tiempo viendo este material.	El control del programa es fácil con los tres enunciados de ciclo del C: el ciclo for, el ciclo while y el ciclo do...while.	Aquí se tratan las funciones printf( ), puts( ) y scanf( ): la entrada y salida de programa será muy fácil a partir de hoy.
DIA 8	**DIA 9**	**DIA 10**	**DIA 11**	**DIA 12**	**DIA 13**	**DIA 14**
Aprenda a agrupar datos similares con arreglos numéricos. Duerma muy bien esta noche porque mañana será un gran día.	Para aprender C tiene que entender los apuntadores, la clave de C. Tómese su tiempo con esta lección.	En C se guardan palabras y frases en cadenas. Aprenda los puntos específicos de las cadenas y lo básico del manejo de memoria.	Aprenda otro método de agrupar variables relacionadas: las estructuras. Se tratan temas de estructuras tanto básicos como avanzados.	Entienda cabalmente el alcance de las variables. Qué son una global y una local. Es el momento para los detalles.	Vuelva a ver el control de programa. Aprenda trucos de programación avanzados, tales como el enunciado switch y los ciclos infinitos.	Aprenda todo lo necesario acerca de los cinco flujos predefinidos de C y sus funciones. Hay mucho que aprender.
DIA 15	**DIA 16**	**DIA 17**	**DIA 18**	**DIA 19**	**DIA 20**	**DIA 21**
Apuntadores: la consecuencia. Este es un día de retos. Hoy trataremos algunas formas complejas de uso de los apuntadores.	Aprenda todo lo necesario acerca de los archivos de disco en un solo día. Los programas más útiles que se escriben emplean archivos de disco.	Aprenda las funciones para comparar, convertir y concatenar cadenas cuando se trata la manipulación de cadenas.	Funciones: la consecuencia. Explore temas avanzados de funciones, incluida la interacción de apuntadores y funciones.	Profundice en la biblioteca de funciones. Explore las funciones matemáticas, de tiempo y de manejo de errores. ¡Sólo quedan dos días!	Amarre hoy algunos cabos sueltos y reciba una segunda lección sobre el manejo de memoria. ¡Casi acabamos!	¡Hagamos fiesta! Todo lo que nos falta son los archivos de encabezado y las directivas del preprocesador. Luego, será usted un experto en C. ¡Felicidades!

Cómo usar este libro

Tal como se puede suponer por el título, este libro ha sido diseñado de tal forma que usted pueda aprender por sí mismo el lenguaje de programación C en 21 días. Dentro de los diversos lenguajes de programación disponibles, cada vez más programadores profesionales escogen al C debido a su poder y flexibilidad. Por las razones que mencionamos en el Día 1, usted no se ha equivocado al seleccionar al C como su lenguaje de programación.

Pensamos que ha hecho una decisión atinada seleccionando este libro como su medio para aprender el C. Aunque hay muchos sobre C, creemos que este libro presenta al C en su secuencia más lógica y fácil de aprender. Lo hemos diseñado pensando en que usted trabaje los capítulos en orden, diariamente. Los capítulos posteriores se apoyan en el material presentado en los primeros. No suponemos que usted tenga experiencia anterior de programación, aunque tenerla con otro lenguaje, como BASIC, puede ayudarle a que el aprendizaje sea más rápido. Tampoco hacemos hipótesis acerca de su computadora o compilador. Este libro se concentra sobre el aprendizaje del C sin importar el compilador.

Características especiales de este libro

El libro contiene algunas características especiales para ayudarle en su aprendizaje del C. Cuadros de sintaxis le muestran cómo usar un concepto específico del C. Cada cuadro proporciona ejemplos concretos y una explicación completa del comando o concepto del C. Para ambientarse al estilo de los cuadros de sintaxis, véase el siguiente ejemplo. (No trate de entender el material, ya que todavía no ha llegado al Día 1.)

La función *printf()*

```
#include <stdio.h>
printf( cadena de formato[,argumentos,...]);
```

`printf()` es una función que acepta una serie de *argumentos*, donde a cada uno se le aplica un *especificador de conversión* en la cadena de formateo dada. `printf()` imprime la información formateada en el dispositivo estándar de salida, que, por lo general, es la pantalla. Cuando se usa `printf()` se necesita incluir el archivo de encabezado de la entrada/salida estándar, STDIO.H.

La *cadena de formato* es imprescindible. Sin embargo, los *argumentos* son opcionales. Para cada argumento debe haber un especificador de conversión. La tabla 7.2 lista los especificadores de conversión más comunes. La cadena de formato también puede contener secuencias de escape. La tabla 7.1 lista las más usadas. A continuación se presentan ejemplos de llamadas a `printf()` y su salida:

Ejemplo 1

```
#include <stdio.h>
main()
```

```
{
   printf( "¡Este es un ejemplo de algo impreso!");
}
```

Despliega

```
¡Este es un ejemplo de algo impreso!
```

Ejemplo 2

```
printf( "Esto imprime un carácter, %c\n un número, %d\n un punto flotante,
%f", 'z', 123, 456.789 );
```

Despliega

```
Esto imprime un carácter, z
un número, 123
un punto flotante, 456.789
```

Otra característica de este libro son los cuadros de **DEBE/NO DEBE,** los cuales dan indicaciones sobre lo que hay que hacer y lo que no hay que hacer.

DEBE	NO DEBE
DEBE Lea el resto de esta sección. Ofrece una explicación de la sección de taller al final de cada día.	
NO DEBE No se salte ninguna de las preguntas del cuestionario o los ejercicios. Si usted puede terminar el taller del día, está listo para pasar al nuevo material.	

Proporcionamos numerosos ejemplos con explicaciones para ayudarle a aprender la manera de programar. Cada día termina con una sección, que contiene respuestas a preguntas comunes relacionadas con el material del día. También hay un taller al final de cada día. El taller contiene cuestionarios y ejercicios. El cuestionario prueba su conocimiento de los conceptos que han sido presentados en ese día. Si desea revisar las respuestas, o está confundido, éstas se encuentran en el apéndice G, "Respuestas".

Sin embargo, usted no aprenderá C solamente leyendo el libro. Si quiere ser un programador, tiene que escribir programas. A continuación de cada juego de preguntas del cuestionario se encuentra un juego de ejercicios. Le recomendamos que trate de hacer cada uno de ellos. Escribir código de C es la mejor manera de aprender el lenguaje de programación C.

Consideramos que los ejercicios de BUSQUEDA DE ERRORES son los más benéficos. Estos son listados de código que contienen problemas comunes. Es su tarea localizar y corregir los errores.

Conforme avance por el libro, algunas de las respuestas a los ejercicios tenderán a hacerse largas. Otros ejercicios tienen varias respuestas posibles. A consecuencia de esto, los últimos capítulos tal vez no den respuestas para todos los ejercicios.

Haciendo un mejor libro

Nada es perfecto, pero nos esforzamos por alcanzar la perfección. Esta edición bestseller tiene algunas nuevas características que vale la pena tener en cuenta. Si usted tiene preguntas específicas acerca de los diferentes compiladores de C, pase al apéndice H. Ahí encontrará listados de las principales características de los compiladores y sugerencias para la instalación. Esperamos que esto le sea de ayuda para elegir el compilador que se adapte mejor a sus necesidades.

Un concepto del C que no fue tratado en la primera edición fueron las *uniones*. Esta edición tiene una sección adicional en el capítulo 11, donde se detallan las uniones. Asegúrese de resolver completamente el nuevo ejercicio en el taller del capítulo 11 que trata este tema.

Al final de cada semana usted encontrará "La revisión de la semana". Esta sección contiene un amplio programa que usa varios de los conceptos tratados durante la semana anterior. Muchas de las líneas del programa tienen números a la izquierda de los números de línea. Estos números identifican el capítulo donde se trata el tema de esa línea. Si cualquiera de los conceptos lo confunde, regrese a ese capítulo.

Aun cuando usted haya dominado los conceptos de C, este libro será una referencia adecuada, y la tarjeta desprendible, en la parte inicial de este libro, es un recurso adicional para usted. La tarjeta, que contiene información por ambos lados, será un útil material de consulta de escritorio al estar escribiendo sus programas de C.

Convenciones usadas en este libro

Este libro usa diferentes tipos de letra para ayudarle a distinguir entre el código de C y el español normal y a identificar conceptos importantes. El código actual de C está escrito en un tipo de letra especial `monoespaciado`. Placeholders, es decir, los términos usados para representar lo que de hecho se tiene que teclear en el código, están escritos en un tipo *`cursivo monoespaciado`*. Los términos nuevos o importantes están escritos en *cursivas*.



Nota: El código fuente impreso en este libro está disponible a través de Peter Aitkin (véase la forma para ordenar el disco, en la parte final de este libro) o por medio de CompuServe. Se puede encontrar el código en el foro Prentice Hall Computer Publishing (PHCP). Para accesarlo, teclee `go sams`.

Aprendiendo C en 21 días

Aprendiendo C en 21 días

Peter Aitken / Bradley Jones

TRADUCCION
Ing. Sergio Luis Ma. Ruiz Faudon
Ingeniero Químico

REVISION TECNICA
Gabriel Guerrero Reyes
Doctor en Informática

Pearson Educación ®

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

APRENDIENDO C EN 21 DIAS

traducido del inglés de la obra: TEACH YOURSELF C IN 21 DAYS.

Autorized translation form the English language edition published by SAMS PUBLISHING
Copyright © 1994

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 in writing from the Publisher.
Spanish language edition published by
Prentice Hall Hispanoamericana, S.A.

Copyright © 1994

Traducción autorizada de la edición en inglés publicada por SAMS PUBLISHING
Copyright © 1994

Todos los derechos reservados. Ninguna parte de este libro puede reproducirse bajo ninguna forma o por ningún medio, electrónico ni mecánico, incluyendo fotocopiado y grabación, ni por ningún sistema de almacenamiento y recuperación de información, sin permiso por escrito del Editor.

Edición en español publicada por

Prentice Hall Hispanoamericana, S.A.

Derechos Reservados © 1994

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

ISBN 968-880-444-4

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

Original English Language Edition Published by

Copyright © 1994 By SAMS PUBLISHING
All Rights Reserved

ISBN 0-672-30448-1

Impreso en México/Printed in Mexico

Editor
Richard K. Swadley

Editor adjunto
Jordan Gold

Gerente de adquisiciones
Stacy Hiquet

Editor de adquisiciones
Gregory Croy

Editor de elaboración
Dean Miller

Redactores
Keith Davenport
Katherine Stuart Ewing
Fran Hatton
Tad Ringo

Formador
Pat Whitmer

Asistente editorial
Sharon Cox

Revisores técnicos
Timothy C. Moore
Scott Parker

Gerente de comercialización
Greg Wiegand

Diseñador de portada
Dan Armstrong

Director de producción y manufactura
Jeff Valler

Jefe de impresión
Kelli Widdifield

Diseñadora del libro
Michele Laseau

Analista de producción
Mary Beth Wakefield

Coordinadora de corrección e indización
Joelynn Gifford

Especialistas de imágenes y gráficos
Dennis Sheehan
Sue VandeWalle

Producción
Ayrika Bryant
Rich Evers
Mitzi Gianakos
Dennis Clay Hager
Juli Pavey
Angela M. Pozdol
Linda Quigley
Beth Rago
Michelle M. Self
Dennis Wesner
Alyssa Yesh

Indizadores
John Sleeva
Suzzane Snyder

Resumen del contenido

La semana 1 de un vistazo

1 Comienzo 3
2 Los componentes de un programa C 21
3 Variables y constantes numéricas 35
4 Enunciados, expresiones y operadores 53
5 Funciones: lo básico 87
6 Control básico del programa 115
7 Entrada/salida básica 139

Revisión de la semana 1 **159**

La semana 2 de un vistazo

8 Arreglos numéricos 169
9 Apuntadores 189
10 Caracteres y cadenas 215
11 Estructuras 241
12 Alcance de las variables 281
13 Más sobre el control de programa 301
14 Trabajando con la pantalla, la impresora y el teclado 331

Revisión de la semana 2 **379**

La semana 3 de un vistazo

15 Más sobre apuntadores 391
16 Uso de archivos de disco 425
17 Manipulación de cadenas 463
18 Cómo obtener más de las funciones 495
19 Exploración de la biblioteca de funciones 513
20 Otras funciones 541
21 Cómo aprovechar las directivas del preprocesador y más 563

Revisión de la semana 3 **585**

Apéndices

A Tabla de caracteres ASCII 595
B Palabras reservadas del C 599

C Precedencia de operadores en C 603
D Notación binaria y hexadecimal 605
E Prototipos de función y archivos de encabezado 609
F Funciones comunes en orden alfabético 619
G Respuestas 627
H Puntos específicos de los compiladores 679

Indice **693**

Contenido

La semana de un vistazo 1

1 Comienzo **3**
Una breve historia del lenguaje C 4
¿Por qué usar C? 4
Preparación para la programación 5
El ciclo de desarrollo del programa 6
Creación del código fuente 7
Compilación del código fuente 8
Enlazar para crear un archivo ejecutable 9
Completando el ciclo de desarrollo 10
El primer programa en C 11
Tecleo y compilación de HELLO.C 12
Resumen 15
Preguntas y respuestas 16
Taller 17
Cuestionario 17
Ejercicios 18

2 Los componentes de un programa C **21**
Un programa corto en C 22
Los componentes de un programa 23
La función *main()* (líneas 5-18) 23
La directiva *#include* (línea 2) 23
Definición de variables (línea 3) 24
Prototipo de función (línea 4) 24
Enunciados del programa (líneas 8, 9, 12, 13, 16, 17, 23) 24
Definición de función (líneas 21-24) 25
Comentarios del programa (líneas 1, 7, 11, 15, 20) 26
Llaves (líneas 6, 18, 22, 24) 27
Ejecución del programa 27
Una nota sobre la precisión 27
Revisión de las partes de un programa 28
Resumen 30
Preguntas y respuestas 30
Taller 31
Cuestionario 31
Ejercicios 32

3 Variables y constantes numéricas **35**
Memoria de la computadora 36
Variables 37

Nombres de variable 37
Tipos de variables numéricas 39
Declaración de variables 42
La palabra clave *typedef* 43
Inicialización de variables numéricas 43
Constantes 44
Constantes literales 44
Constantes simbólicas 46
Resumen 49
Preguntas y respuestas 50
Taller 51
Cuestionario 51
Ejercicios 52

4 Enunciados, expresiones y operadores 53

Enunciados 54
Enunciados y el espacio en blanco 54
Enunciados compuestos 55
Expresiones 56
Expresiones simples 56
Expresiones complejas 56
Operadores 58
El operador de asignación 58
Operadores matemáticos 58
Precedencia de operadores y los paréntesis 63
Orden para la evaluación de subexpresiones 65
Operadores relacionales 65
El enunciado *if* 67
Evaluación de expresiones relacionales 72
Precedencia de los operadores relacionales 73
Operadores lógicos 75
Más sobre valores cierto/falso 76
Precedencia de los operadores lógicos 77
Operadores de asignación compuestos 79
El operador condicional 80
El operador coma 80
Resumen 81
Preguntas y respuestas 82
Taller 83
Cuestionario 83
Ejercicios 84

5 Funciones: lo básico 87

¿Qué es una función? 88
La definición de una función 88
La ilustración de una función 89

La manera en que trabaja una función 91
Las funciones y la programación estructurada 93
Las ventajas de la programación estructurada 93
La planeación de un programa estructurado 93
El enfoque descendente 95
Escritura de una función 96
El encabezado de la función 96
El cuerpo de la función 99
El prototipo de la función 104
Paso de argumentos a una función 105
Llamado de funciones 106
Recursión 107
¿Dónde se ponen las funciones? 109
Resumen 110
Preguntas y respuestas 110
Taller 111
Cuestionario 111
Ejercicios 112

6 Control básico del programa 115

Arreglos: lo básico 116
Control de la ejecución del programa 117
El enunciado *for* 117
Enunciados *for* anidados 123
El enunciado *while* 125
Enunciados *while* anidados 128
El ciclo *do...while* 130
Ciclos anidados 134
Resumen 135
Preguntas y respuestas 136
Taller 136
Cuestionario 137
Ejercicios 137

7 Entrada/salida básica 139

Desplegado de la información en la pantalla 140
La función *printf()* 140
Desplegado de mensajes con *puts()* 148
Entrada de datos numéricos con *scanf()* 149
Resumen 154
Preguntas y respuestas 154
Taller 155
Cuestionario 155
Ejercicios 156

Revisión de la semana 159

La semana de un vistazo 167

8 Arreglos numéricos **169**

¿Qué es un arreglo? 170
Arreglos de una sola dimensión 170
Arreglos multidimensionales 175
Denominación y declaración de arreglos 176
Inicialización de arreglos 178
Tamaño máximo del arreglo 182
Resumen 184
Preguntas y respuestas 185
Taller 186
Cuestionario 186
Ejercicios 186

9 Apuntadores **189**

¿Qué es un apuntador? 190
La memoria de la computadora 190
Creación de un apuntador 191
Los apuntadores y las variables simples 192
Declaración de apuntadores 192
Inicialización de apuntadores 192
Uso de apuntadores 193
Los apuntadores y los tipos de variables 195
Los apuntadores y los arreglos 197
El nombre del arreglo como un apuntador 197
Almacenamiento de elementos de arreglo 198
Aritmética de apuntadores 200
Precauciones con los apuntadores 204
Notación de subíndices de arreglo y apuntadores 205
Paso de arreglos a funciones 206
Resumen 210
Preguntas y respuestas 211
Taller 211
Cuestionario 212
Ejercicios 212

10 Caracteres y cadenas **215**

El tipo de dato *char* 216
Uso de variables de carácter 217
Uso de cadenas 219
Arreglos de caracteres 219
Inicialización de arreglos de caracteres 220
Cadenas y apuntadores 221
Cadenas sin arreglos 221
Asignación de espacio para la cadena en la compilación 222

La función *malloc()* 222
Desplegado de cadenas y caracteres 227
La función *puts()* 227
La función *printf()* 228
Lectura de cadenas desde el teclado 229
Entrada de cadenas con la función *gets()* 229
Entrada de cadenas con la función *scanf()* 232
Resumen 234
Preguntas y respuestas 235
Taller 236
Cuestionario 236
Ejercicios 238

11 Estructuras 241

Estructuras simples 242
Definición y declaración de estructuras 242
Acceso de los miembros de la estructura 243
Estructuras más complejas 245
Estructuras que contienen estructuras 245
Estructuras que contienen arreglos 249
Arreglos de estructuras 251
Inicialización de estructuras 255
Estructuras y apuntadores 257
Apuntadores como miembros de estructuras 258
Apuntadores a estructuras 260
Apuntadores y arreglos de estructuras 262
Paso de estructuras como argumentos a funciones 265
Uniones 267
Definición, declaración e inicialización de uniones 267
Acceso de miembros de la unión 267
Listas encadenadas 272
La organización de una lista encadenada 273
La función *malloc()* 275
Implementación de una lista encadenada 275
typedef y las estructuras 275
Resumen 276
Preguntas y respuestas 277
Taller 277
Cuestionario 278
Ejercicios 278

12 Alcance de las variables 281

¿Qué es el alcance? 282
Una demostración del alcance 282
¿Por qué es importante el alcance? 284

Variables externas 284
Alcance de las variables externas 285
Cuándo usar variables externas 285
La palabra clave *extern* 286
Variables locales 287
Variables estáticas versus automáticas 287
El alcance de los parámetros de la función 290
Variables estáticas externas 291
Variables de registro 291
Variables locales y la función *main()* 292
¿Qué clase de almacenamiento se debe usar? 293
Variables locales y bloques 294
Resumen 295
Preguntas y respuestas 296
Taller 297
Cuestionario 297
Ejercicios 298

13 Más sobre el control de programa 301

Terminación anticipada de ciclos 302
El enunciado *break* 302
El enunciado *continue* 304
El enunciado *goto* 306
Ciclos infinitos 309
El enunciado *switch* 312
Terminación del programa 321
La función *exit()* 321
La función *atexit()* (sólo para el DOS) 322
Ejecución de comandos del sistema operativo en un programa 325
Resumen 327
Preguntas y respuestas 327
Taller 328
Cuestionario 328
Ejercicios 328

14 Trabajando con la pantalla, la impresora y el teclado 331

Los flujos y el C 332
¿Qué es exactamente la Entrada/Salida de un programa? 332
¿Qué es un flujo? 333
Flujos de texto contra flujos binarios 334
Los flujos predefinidos 334
Funciones de flujo del C 335
Un ejemplo 335
Aceptando entrada del teclado 336
Entrada de caracteres 336
Entrada formateada 351

Salida a pantalla 359
Salida de caracteres con *putchar()*, *putc()* y *fputc()* 359
Uso de *puts()* y *fputs()* para la salida de flujos 361
Uso de *printf()* y *fprintf()* para la salida formateada 362
Redirección de la entrada y la salida 369
Cuándo usar *fprintf()* 371
Uso de *stderr* 371
Resumen 373
Preguntas y respuestas 374
Taller 375
Cuestionario 375
Ejercicios 376

Revisión de la semana **379**

La semana de un vistazo **389**

15 Más sobre apuntadores 391

Apuntadores a apuntadores 392
Apuntadores y arreglos de varias dimensiones 393
Arreglos de apuntadores 402
Cadenas y apuntadores: una revisión 402
Arreglos de apuntadores a *char* 403
Un ejemplo 405
Apuntadores a funciones 411
Declaración de un apuntador a una función 411
Inicialización y uso de un apuntador a una función 412
Resumen 421
Preguntas y respuestas 421
Taller 422
Cuestionario 422
Ejercicios 423

16 Uso de archivos de disco 425

Flujos y archivos de disco 426
Tipos de archivos de disco 426
Nombres de archivo 427
Apertura de un archivo para usarlo 427
Escritura y lectura de datos de archivo 431
Entrada y salida de archivos formateados 431
Entrada y salida de caracteres 436
Entrada y salida directas de archivos 438
Bufer con archivos: cierre y vaciado de archivos 441
Acceso de archivos secuencial contra aleatorio 443
Las funciones *ftell()* y *rewind()* 444
La función *fseek()* 446
Detección del fin de archivo 449

Funciones para manejo de archivos 452
Borrado de un archivo 452
Renombrado de un archivo 453
Copiado de un archivo 454
Uso de archivos temporales 457
Resumen 459
Preguntas y respuestas 459
Taller 460
Cuestionario 460
Ejercicios 461

17 Manipulación de cadenas 463

Longitud y almacenamiento de cadenas 464
Copia de cadenas 465
La función *strcpy()* 465
La función *strncpy()* 467
La función *strdup()* 468
Concatenación de cadenas 469
La función *strcat()* 469
La función *strncat()* 471
Comparación de cadenas 472
Comparación de dos cadenas 473
Comparación de dos cadenas: ignorando mayúsculas y minúsculas 475
Comparación parcial de cadenas 475
Búsqueda en cadenas 476
La función *strchr()* 476
La función *strrchr()* 478
La función *strcspn()* 478
La función *strspn()* 479
La función *strpbrk()* 480
La función *strstr()* 481
Conversión de cadenas 482
Funciones diversas para cadenas 483
La función *strrev()* 483
Las funciones *strset()* y *strnset()* 484
Conversión de cadenas a números 485
La función *atoi()* 485
La función *atol()* 486
La función *atof()* 486
Funciones de prueba de caracteres 487
Resumen 492
Preguntas y respuestas 492
Taller 493
Cuestionario 493
Ejercicios 493

18 Cómo obtener más de las funciones 495

Paso de apuntadores a funciones 496
Apuntadores tipo *void* 500
Funciones con número variable de argumentos 504
Funciones que regresan un apuntador 507
Resumen 509
Preguntas y respuestas 510
Taller 510
Cuestionario 510
Ejercicios 511

19 Exploración de la biblioteca de funciones 513

Funciones matemáticas 514
Funciones trigonométricas 514
Funciones exponenciales y logarítmicas 515
Funciones hiperbólicas 515
Otras funciones matemáticas 516
Manejo del tiempo 518
Representación del tiempo 518
Las funciones de tiempo 518
Uso de las funciones de tiempo 522
Funciones para el manejo de errores 524
La función *assert()* 524
El archivo de encabezado ERRNO.H 526
La función *perror()* 527
Búsqueda y ordenamiento 529
Búsqueda con *bsearch()* 529
Ordenamiento con *qsort()* 530
Dos demostraciones de búsqueda y ordenamiento 531
Resumen 537
Preguntas y respuestas 537
Taller 538
Cuestionario 538
Ejercicios 538

20 Otras funciones 541

Conversiones de tipo 542
Conversiones automáticas de tipo 542
Conversiones explícitas con modificadores de tipo 543
Asignación de espacio de almacenamiento en memoria 545
La función *malloc()* 546
La función *calloc()* 546
La función *realloc()* 547
La función *free()* 549
Uso de argumentos de la línea de comandos 551

Operaciones sobre bits 554
Los operadores de desplazamiento 554
Los operadores lógicos a nivel de bit 555
El operador de complemento 556
Campos de bits en estructuras 556
Resumen 558
Preguntas y respuestas 559
Taller 560
Cuestionario 560
Ejercicios 561

21 Cómo aprovechar las directivas del preprocesador y más 563

Programación con varios archivos fuente 564
Ventajas de la programación modular 564
Técnicas de la programación modular 564
Componentes de los módulos 566
Variables externas y la programación modular 567
Uso de archivos .OBJ 568
El preprocesador de C 569
La directiva del preprocesador *#define* 569
La directiva *#include* 575
Uso de *#if, #elif, #else* y *#endif* 576
Uso de *#if...#endif* para ayudarse en la depuración 577
Cómo evitar la inclusión múltiple de archivos de encabezado 578
La directiva *#undef* 579
Macros predefinidas 579
Resumen 580
Preguntas y respuestas 581
Taller 581
Cuestionario 581
Ejercicios 582

Revisión de la semana 585

Apéndices

A Tabla de caracteres ASCII 595

B Palabras reservadas del C 599

C Precedencia de operadores en C 603

D Notación binaria y hexadecimal 605

E Prototipos de función y archivos de encabezado 609

F Funciones comunes en orden alfabético 619

G Respuestas 627

Respuestas para el Día 1 "Comienzo" 628
Cuestionario 628
Ejercicios 628
Respuestas para el Día 2 "Los componentes de un programa C" 629
Cuestionario 629
Ejercicios 630
Respuestas para el Día 3 "Variables y constantes numéricas" 631
Cuestionario 631
Ejercicios 632
Respuestas para el Día 4 "Enunciados, expresiones y operadores" 633
Cuestionario 633
Ejercicios 634
Respuestas para el Día 5 "Funciones: lo básico" 637
Cuestionario 637
Ejercicios 637
Respuestas para el Día 6 "Control básico del programa" 641
Cuestionario 641
Ejercicios 642
Respuestas para el Día 7 "Entrada/salida básica" 643
Cuestionario 643
Ejercicios 644
Respuestas para el Día 8 "Arreglos numéricos" 648
Cuestionario 648
Ejercicios 649
Respuestas para el Día 9 "Apuntadores" 654
Cuestionario 654
Ejercicios 655
Respuestas para el Día 10 "Caracteres y cadenas" 656
Cuestionario 656
Ejercicios 658
Respuestas para el Día 11 "Estructuras" 658
Cuestionario 658
Ejercicios 659
Respuestas para el Día 12 "Alcance de las variables" 661
Cuestionario 661
Ejercicios 662
Respuestas para el Día 13 "Más sobre el control del programa" 666
Cuestionario 666
Ejercicios 667
Respuestas para el Día 14 "Trabajando con la pantalla,
la impresora y el teclado" 668
Cuestionario 668
Ejercicios 669
Respuestas para el Día 15 "Más sobre apuntadores" 669
Cuestionario 669
Ejercicios 670

Respuestas para el Día 16 "Uso de archivos de disco" 671
Cuestionario 671
Ejercicios 672
Respuestas para el Día 17 "Manipulación de cadenas" 672
Cuestionario 672
Ejercicios 673
Respuestas para el Día 18 "Obteniendo más de las funciones" 674
Cuestionario 674
Ejercicios 674
Respuestas para el Día 19 "Exploración de la biblioteca de funciones" 675
Cuestionario 675
Ejercicios 676
Respuestas para el Día 20 "Otras funciones" 676
Cuestionario 676
Ejercicios 677
Respuestas para el Día 21 "Aprovechando las directivas del preprocesador y más" 678
Cuestionario 678

H Puntos específicos de los compiladores 679

Instalación de la edición estándar del Visual C/C++ de Microsoft 682
Instalación de lo mínimo 683
Instalación del Turbo C/C++ para DOS de Borland 685
Instalación de lo mínimo para el Turbo C/C++ para DOS de Borland 686
¿Qué ofrecen los compiladores? 688
Borland C++ 688
Turbo C++ para DOS de Borland 689
Edición estándar del Visual C++ de Microsoft 690
Otros compiladores 691

Indice 693

Reconocimientos

Mi agradecimiento a todas las personas que me ayudaron a llevar este libro a su término: la gente de Sams Publishing. Si esta obra le resulta a usted una útil guía de enseñanza, gran parte del mérito es de ellos. Cualesquiera errores son, desde luego, de mi absoluta responsabilidad.

Peter Aitken

Querría agradecerle a Greg Guntle por darme la confianza necesaria para emprender proyecto tal como el de escribir un libro; querría agradecerle también a Peter Aitken y a Joe Wikert por aportar la base de esta obra. Y, sobre todo, querría agradecerle a Stacy Hiquet el tiempo que pasó respondiendo a todas mis preguntas y orientándome hacia la culminación del libro.

Bradley Jones

Revisión por la Indianapolis Computer Society
Diane VanOsdol
Brenda Havens
Jay Ferguson
Jeffrey Callaway

Acerca de los autores

Peter Aitken es Profesor Adjunto en el Centro Médico de la Universidad de Duke, donde somete a intenso uso las PC en sus investigaciones sobre el sistema nervioso. Es experimentado escritor de temas relativos a las microcomputadoras, con una producción de unos 60 artículos y 12 libros en su haber. Los escritos de Aitken abarcan tanto los temas de aplicaciones como los de programación; entre sus libros se cuentan *QuickBasic Advanced Techniques* (Que), *Learning C* (Howard W. Sams) y *The First Book of 1-2-3 for Windows* (Howard W. Sams). También es Editor colaborador de la revista *PC Techniques*.

Bradley Jones es Programador C de profesión. Ha ayudado a la creación de sistemas para varias empresas estadounidenses. También es miembro activo de la Indianapolis Computer Society, en la que dirige la enseñanza de C y C++ como jefe de C/C++ SIG. Asimismo, es colaborador regular de la revista *Indy PC News*.

SEMANA 1

UN VISTAZO

Como preparación para la primera semana del aprendizaje de la programación en C se necesitan unas cuantas cosas: un compilador, un editor y este libro. Si no se tiene un compilador o un editor, todavía se puede usar este libro; sin embargo, su valor será limitado. La mejor forma de aprender un lenguaje de programación va más allá de la sola lectura de un libro: tiene que ver con el tecleo y la ejecución de varios programas en C. Los diversos programas en C, incluidos en este libro, proporcionan un entrenamiento práctico para el nuevo programador.

Este libro está organizado de tal forma que cada día termina con un taller que contiene un cuestionario y algunos ejercicios. Al final de cada día, usted deberá ser capaz de responder todas las preguntas del cuestionario y de resolver los ejercicios. Al principio se proporcionan las respuestas a todas las preguntas y ejercicios en el apéndice G, "Respuestas". En los días siguientes no se dan respuestas para todos los ejercicios, ya que hay muchas soluciones posibles. Le recomendamos encarecidamente que aproveche los ejercicios y revise sus respuestas.

Dónde andamos...

La primera semana trata el material básico que se necesita para saber cómo comprender el C completamente. En los días 1, "Comienzo", y 2, "Los componentes de un programa C", usted aprenderá la manera de crear un programa C y reconocer los elementos básicos de un programa simple. El día 3, "Variables y constantes numéricas", complementa lo tratado en los primeros dos días definiendo los tipos de variables. El día 4, "Enunciados, expresiones y operadores", toma las variables y añade expresiones simples, para que, de esta forma, puedan ser creados nuevos valores. El día también proporciona información sobre la manera de tomar decisiones y cambiar el flujo del programa usando enunciados `if`. El día 5, "Funciones: lo básico", trata las funciones del C y la programación estructurada. El día 6, "Control básico del programa", presenta más comandos que le permitirán controlar el flujo de los programas. La semana termina en el día 7, "Entrada/salida básica", con un análisis sobre la impresión de información y una ayuda para hacer que los programas interactúen con el teclado y la pantalla.

Esta es una gran cantidad de material para tratarla en solamente una semana, pero si se toma la información de un capítulo por día, no se debe tener problemas.



Nota: Este libro trata el lenguaje C de acuerdo con el estándar ANSI. Esto significa que no importa qué compilador use usted, siempre y cuando siga las reglas del estándar ANSI. El apéndice H, "Puntos específicos de los compiladores", ofrece alguna información general sobre los compiladores más comunes.

DIA 1

Comienzo

Bienvenido a *¡Aprenda C por usted mismo en 21 días!* Este capítulo le da los medios para llegar a ser un programador de C eficiente. Hoy aprenderá:

- ❑ Por qué el C es la mejor alternativa entre los lenguajes de programación.
- ❑ Los pasos en el ciclo de desarrollo de un programa.
- ❑ La manera de escribir, compilar y ejecutar el primer programa en C.
- ❑ Acerca de los mensajes de error generados por el compilador y el enlazador.

Una breve historia del lenguaje C

Tal vez se pregunte cuál ha sido el origen del lenguaje C y de dónde le vino su elegante nombre. El C fue creado por Dennis Ritchie en los laboratorios de la Bell Telephone, en 1972. El lenguaje no fue creado por el gusto de hacerlo, sino para un fin específico: el diseño del sistema operativo UNIX (el cual se usa en muchas minicomputadoras). Desde el principio, el C tuvo como propósito ser útil: permitir a los programadores atareados que las cosas se pudieran hacer.

Como el C es un lenguaje muy poderoso y flexible, su uso se difundió rápidamente más allá de los laboratorios Bell. Los programadores de todo el mundo comenzaron a usarlo para escribir todo tipo de programas. Sin embargo, diferentes organizaciones comenzaron a utilizar muy pronto sus propias versiones del C, y las pequeñas diferencias entre las implementaciones comenzaron a dar problemas a los programadores. Para resolver este problema, el American National Standards Institute (ANSI) formó un comité en 1983 para establecer una definición estándar del C, que llegó a ser conocida como el *C estándar ANSI.* Con unas cuantas excepciones, todos los compiladores de C modernos se adhieren a este estándar.

Ahora, ¿por qué tiene este nombre? El lenguaje C se llama de esta forma debido a que su predecesor fue llamado B. El lenguaje B fue desarrollado por Ken Thompson también en los laboratorios Bell. Tal vez se imagine fácilmente por qué fue llamado B.

Por qué usar C

En el mundo actual de la programación de computadoras, hay muchos lenguajes de alto nivel entre los que se puede escoger, como C, Pascal, BASIC y Modula. Todos éstos son lenguajes excelentes, adecuados para la mayoría de las labores de programación. No obstante, hay varias razones por las cuales muchos profesionales de la computación sienten que el C se encuentra a la cabeza de la lista:

- C es un lenguaje poderoso y flexible. Lo que se puede lograr con el C está limitado solamente por la imaginación. El lenguaje, por sí mismo, no le pone límites. El C se usa para proyectos tan diversos como sistemas operativos, procesadores de palabras, gráficos, hojas de cálculo y hasta compiladores para otros lenguajes.
- El C es un lenguaje común, preferido por los programadores profesionales. Como resultado, se tienen disponibles una amplia variedad de compiladores de C y accesorios útiles.
- El C es un lenguaje transportable. *Transportable* significa que un programa en C escrito para un sistema de computadora (por ejemplo, una PC de IBM) puede ser compilado y ejecutado en otro sistema (tal vez en un sistema DEC VAX) con pocas o ninguna modificación. La transportabilidad es aumentada con el estándar ANSI para el C, el juego de reglas para los compiladores C que se mencionaron anteriormente.
- El C es un lenguaje de pocas palabras, que contiene solamente unos cuantos términos llamados *palabras clave* que son la base sobre la que está construida la funcionalidad del lenguaje. Tal vez piense usted que un lenguaje con más palabras clave (llamadas, algunas veces, palabras reservadas) pudiera ser más poderoso. Esto no es cierto. Conforme programe en C, encontrará que puede ser programado para ejecutar cualquier tarea.
- El C es modular. El código de C puede (y debe) ser escrito en rutinas llamadas *funciones.* Estas funciones pueden ser reutilizadas en otras aplicaciones o programas. Pasando información a las funciones, se puede crear código útil y reutilizable.

Como muestran estas características, el C es una alternativa excelente para ser el primer lenguaje de programación. ¿Qué hay acerca de este nuevo lenguaje llamado C++ (pronunciado *C plus plus)?* Tal vez ya haya oído acerca del C++ y de una nueva técnica de programación llamada *programación orientada a objetos.* Tal vez se pregunte cuáles son las diferencias entre C y C++, y si debe aprender por sí mismo C++ en vez de C.

¡No se preocupe! C++ es una versión *mejorada* del C, lo que significa que el C++ contiene todo lo que tiene el C, y nuevos agregados para la programación orientada a objetos. Si va a aprender el C++, casi todo lo que aprenda acerca del C todavía será aplicable al C++. Al aprender C, no sólo estará aprendiendo el lenguaje de programación actual más poderoso y generalizado, sino también se estará preparando para la programación orientada a objetos del mañana.

Preparación para la programación

Cuando se trate de resolver un problema, se deben tomar ciertos pasos. En primer lugar, el problema debe ser definido . ¡Si no se sabe cuál es el problema, no se puede encontrar una solución! Una vez que se conoce el problema, se puede pensar un plan para componerlo. Una vez que se tiene un plan, por lo general se le puede implementar fácilmente. Por último, una vez que se implementa el plan, se deben probar los resultados para ver si el problema se resuelve. Esta misma lógica también puede ser aplicada a muchas otras áreas, incluida la programación.

Cuando se cree un programa en C (o en sí un programa de computadora en cualquier lenguaje), se debe seguir una secuencia de pasos similar:

1. Determinar el objetivo del programa.
2. Determinar el método que se quiere usar para la escritura del programa.
3. Crear el programa para resolver el problema.
4. Ejecutar el programa para ver los resultados.

Un ejemplo de un objetivo (véase el paso 1) puede ser escribir un procesador de palabras o un programa de base de datos. Un objetivo mucho más simple es desplegar el nombre de uno en la pantalla. Si no se tiene un objetivo, no se podrá escribir un programa, por lo que ya se tiene dado el primer paso.

El segundo paso es determinar el método que se quiere usar para la escritura del programa. ¿Se necesita un programa de computadora para resolver el problema? ¿Qué información necesita ser registrada? ¿Qué fórmulas serán utilizadas? Durante este paso se debe tratar de determinar lo que se necesita saber y en qué orden debe ser implementada la solución.

Como un ejemplo, supongamos que alguien nos pide escribir un programa para determinar el área de un círculo. El paso 1 está completo, ya que se sabe el objetivo: determinar el área de un círculo. El paso 2 consiste en determinar lo que se necesita saber para calcular el área. En este ejemplo, supongamos que el usuario del programa proporcionará el radio del círculo. Sabiendo esto, se puede aplicar la fórmula πr^2 para obtener la respuesta. Ahora se tienen las piezas que se necesitan, por lo que se puede continuar a los pasos 3 y 4, que son llamados "ciclo de desarrollo del programa".

El ciclo de desarrollo del programa

El ciclo de desarrollo del programa tiene sus propios pasos. En el primer paso se usa un editor para crear un archivo de disco que contiene el *código fuente*. En el segundo paso se *compila* el código fuente para crear un *archivo objeto*. En el tercer paso se enlaza el código compilado

para crear un *archivo ejecutable.* Por último, el cuarto paso es ejecutar el programa para ver si funciona como se planeó originalmente.

Creación del código fuente

El código fuente es una serie de enunciados o comandos usados para darle instrucciones a la computadora de que ejecute las tareas que se desean. Como se dijo anteriormente, el primer paso en el ciclo de desarrollo del programa es teclear el código fuente con un editor. Por ejemplo, a continuación se presenta una línea de código fuente de C:

```
printf("Hello, Mom!");
```

Este enunciado le indica a la computadora que despliegue el mensaje `Hello, Mom!` en la pantalla. (Por ahora, no se preocupe sobre la manera en que funciona este enunciado.)

Uso de un editor

Algunos compiladores vienen con un editor que puede usarse para teclear el código fuente, y otros no. Consulte los manuales del compilador para ver si el compilador viene con un editor. En caso de no ser así, se tienen disponibles muchos editores.

La mayoría de los sistemas de cómputo incluyen un programa que puede usarse como editor. Si se está utilizando un sistema UNIX, se pueden usar comandos como ed, ex, edit, emacs o vi. Si se está usando Windows de Microsoft, se dispone del Notepad. Con DOS 5.0, se puede usar Edit, y si se está usando una versión de DOS anterior a la 5.0, se puede usar Edlin.

La mayoría de los procesadores de palabras usan códigos especiales para formatear sus documentos. Estos códigos no pueden ser leídos correctamente por otros programas. El American Standard Code for Information Interchange (ASCII) ha especificado un formato de texto estándar que casi cualquier programa, incluyendo el C, puede usar. La mayoría de los procesadores de palabras, como WordPerfect, Display Write, Word y WordStar, tienen la capacidad de guardar archivos fuente en formato ASCII (como un archivo de texto, en vez de un archivo de documento). Cuando se quiere guardar un archivo de procesador de palabras como un archivo ASCII, seleccione la opción ASCII o texto al momento de guardarlo.

Si usted no quiere usar ninguno de estos editores, puede comprar un editor diferente. Hay paquetes tanto comerciales como de dominio público que han sido diseñados específicamente para teclear código fuente.

Cuando se guarda un archivo fuente, se le debe dar un nombre. ¿Cómo debe ser llamado un archivo fuente? El nombre que se le dé al archivo debe describir lo que hace el programa. Además, cuando se guardan archivos fuente de programas C se le debe dar al archivo una extensión .C. Aunque se le puede dar al archivo fuente cualquier nombre y extensión que se desee, se considera adecuado usar la extensión .C.

DEBE	NO DEBE
Este libro trata el C estándar ANSI. Esto significa que no importa qué compilador de C se use, siempre y cuando se apegue al estándar ANSI. El apéndice H, "Puntos específicos de los compiladores", proporciona alguna información genérica sobre los compiladores más populares.	

Compilación del código fuente

Aunque uno puede ser capaz de entender el código fuente del C (¡por lo menos después de leer este libro usted será capaz de hacerlo!), la computadora no puede. Una computadora requiere instrucciones digitales, o binarias, en lo que es llamado *lenguaje de máquina.* Antes de que un programa en C pueda ejecutarse en una computadora, debe ser traducido del código fuente a lenguaje de máquina. Esta traducción, el segundo paso en el desarrollo del programa, es ejecutada por un programa llamado *compilador.* El compilador toma el archivo del código fuente como entrada y produce un archivo en disco que contiene las instrucciones de lenguaje de máquina que corresponden a los enunciados del código fuente. Las instrucciones del lenguaje de máquina creadas por el compilador son llamadas *código objeto,* y el archivo de disco que las contiene, *archivo objeto.*

Cada compilador requiere que se usen sus propios comandos para crear el código objeto. Para compilar típicamente se usa el comando que pone en ejecución el compilador seguido del nombre de archivo del archivo fuente. Los siguientes son ejemplos de comandos dados para compilar un archivo fuente llamado RADIUS.C usando varios compiladores para DOS:

C de Microsoft	cl radius.c
Turbo C de Borland	tcc radius.c
C de Borland	bcc radius.c
C de Zortec	ztc radius.c

Para compilar RADIUS.C en una máquina UNIX, use

```
cc radius.c
```

Consulte el manual del compilador para determinar el comando exacto para su compilador.

Después de que se compile, se tiene un archivo objeto. Si se ve una lista de los archivos del directorio donde se hizo la compilación, se deberá encontrar un archivo con el mismo nombre que el archivo fuente pero con una extensión .OBJ (en vez de extensión .C). La extensión .OBJ es reconocida como un archivo objeto, y usada por el enlazador. En sistemas UNIX el compilador crea archivos objeto con la extensión .O, en vez de la extensión .OBJ.

Enlazar para crear un archivo ejecutable

Se requiere un paso adicional antes de que se pueda ejecutar el programa. Parte del lenguaje C es una *biblioteca de funciones* que contiene el código objeto (esto es, código que ya ha sido compilado) para funciones predefinidas. Una *función predefinida* contiene código C que ya ha sido escrito, y se proporciona en una forma lista para usarse con el paquete del compilador. La función `printf()`, usada en el ejemplo anterior, es una función de biblioteca.

Estas funciones de biblioteca ejecutan tareas que se necesitan frecuentemente, como el desplegado de la información en la pantalla y la lectura de datos a partir de archivos de disco. Si el programa usa cualquiera de estas funciones (y difícilmente existe un programa que no use por lo menos una de ellas), el archivo objeto producido cuando fue compilado el código fuente debe ser combinado con el código objeto de la biblioteca de funciones para crear el programa final ejecutable. (*Ejecutable* significa que el programa puede correr, o ser ejecutado, en la computadora.) Este proceso es llamado *enlazado* y es ejecutado por un programa llamado (¡adivínelo!) *enlazador.*

Los pasos desde el código fuente al código objeto y al programa ejecutable están diagramados en la figura 1.1.

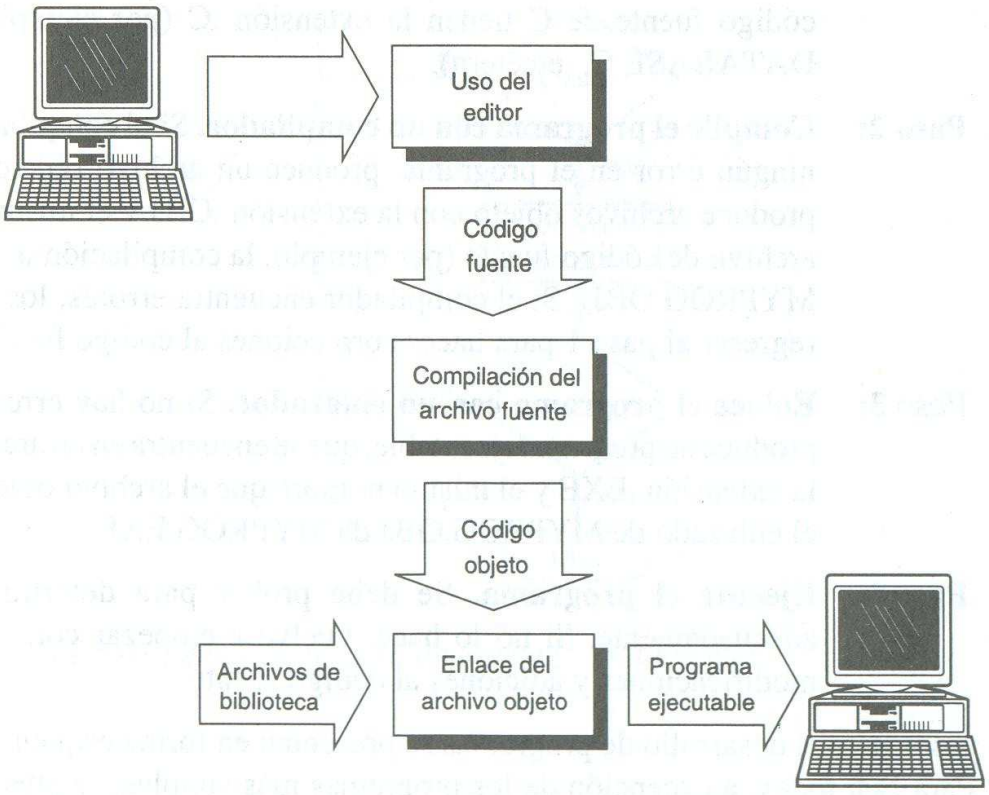


Figura 1.1. *El código fuente del C que se escribe se convierte en código objeto mediante el compilador y, después, en un archivo ejecutable por el enlazador.*

Completando el ciclo de desarrollo

Una vez que el programa está compilado y enlazado para crear un archivo ejecutable, se puede correr tecleando su nombre en la línea de comandos del sistema, en forma similar como se hace con cualquier otro programa. Si se ejecuta el programa y se reciben resultados diferentes a lo que se creía o debiera, hay que regresar al primer paso. Se debe identificar lo que causó el problema y corregirlo en el código fuente. Cuando se hace un cambio al código fuente, se necesita volver a compilar y enlazar el programa para crear una versión corregida del archivo ejecutable. ¡Seguiremos este ciclo hasta que se logre que el programa ejecute exactamente como se pretende!

Una nota final sobre la compilación y el enlazado. Aunque la compilación y el enlazado se mencionan como pasos separados, muchos compiladores, como los de DOS mencionados anteriormente, hacen ambas cosas en un solo paso. Sin tomar en cuenta el método por el cual se logra la compilación y el enlazado, comprenda que estos dos procesos, aunque se hagan con un solo comando, son dos acciones separadas.

Ciclo de desarrollo del C

Sintaxis

Paso 1: **Use un editor para escribir el código fuente.** Por tradición, los archivos del código fuente de C tienen la extensión .C (por ejemplo, MYPROG.C, DATABASE.C, etcétera).

Paso 2: **Compile el programa con un compilador.** Si el compilador no encuentra ningún error en el programa, produce un archivo objeto. El compilador produce archivos objeto con la extensión .OBJ y el mismo nombre que el archivo de código fuente (por ejemplo, la compilación de MYPROG.C da MYPROG.OBJ). Si el compilador encuentra errores, los reporta. Se debe regresar al paso 1 para hacer correcciones al código fuente.

Paso 3: **Enlace el programa con un enlazador.** Si no hay errores, el enlazador produce un programa ejecutable, que se encuentra en un archivo de disco con la extensión .EXE y el mismo nombre que el archivo objeto (por ejemplo, el enlazado de MYPROG.OBJ da MYPROG.EXE).

Paso 4: **Ejecute el programa.** Se debe probar para determinar si funciona adecuadamente. Si no lo hace, vuelva a empezar con el paso 1 y haga modificaciones y adiciones al código fuente.

Los pasos para el desarrollo de programas se presentan en forma esquemática en la figura 1.2. Para casi todos, a excepción de los programas más simples, se puede pasar por esta secuencia muchas veces antes de terminar el programa. ¡Incluso los programadores más experimentados no pueden sentarse y escribir un programa completo y sin errores en un solo paso! Debido a que va a estar pasando por el ciclo: editar-compilar-enlazar-probar muchas

veces, es importante que se familiarice con las herramientas: el editor, el compilador y el enlazador.

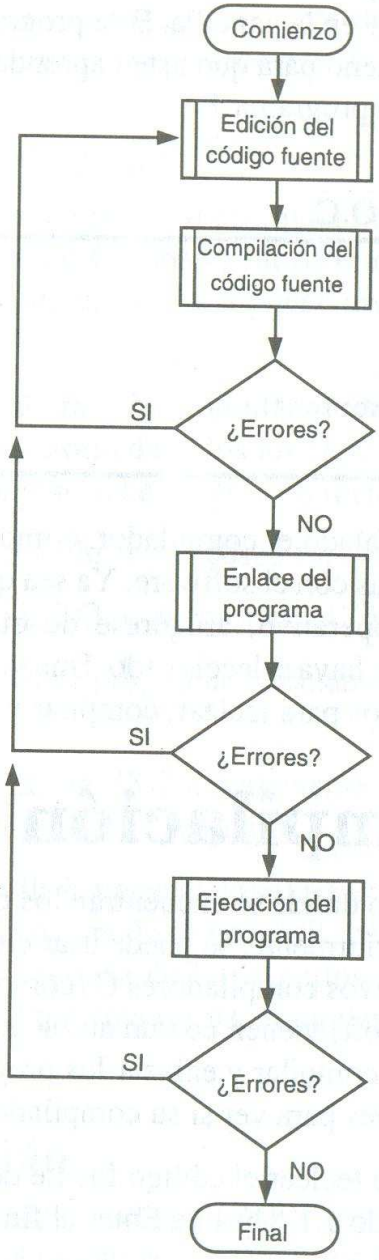


Figura 1.2. *Los pasos en el desarrollo de la programación en C.*

El primer programa en C

¡Probablemente esté ansioso de intentar el primer programa C! Para ayudarle a familiarizarse con el compilador, a continuación se presenta una muestra rápida para que la haga. Tal vez

no entienda todo en este momento, pero deberá ambientarse al proceso de escritura, compilación y ejecución de un programa C real.

Esta muestra usa un programa llamado HELLO.C que no hace más que desplegar las palabras `Hello, World!` en la pantalla. Este programa, una introducción tradicional a la programación en C, es bueno para que usted aprenda. El código fuente para HELLO.C se encuentra en el listado de programa 1.1.

Captura **Listado 1.1. HELLO.C.**

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf("Hello, World!");
6: }
```

Asegúrese de que ha instalado el compilador, como se especifica en las instrucciones de instalación proporcionadas con el software. Ya sea que esté trabajando con UNIX, DOS o cualquier otro sistema operativo, asegúrese de que comprende la manera de usar el compilador y el editor que haya seleccionado. Una vez que se encuentre listo el compilador y el editor, siga estos pasos para teclear, compilar y ejecutar a HELLO.C.

Tecleo y compilación de HELLO.C

1. Active el directorio donde se encuentran los programas de C y arranque el editor. Como se dijo anteriormente, se puede usar cualquier editor de texto, pero la mayoría de los nuevos compiladores C (como el Turbo C++ de Borland y el QuickC de Microsoft) vienen con un ambiente de desarrollo integrado (IDE) que le permite teclear, compilar y enlazar los programas en un ambiente adecuado. Véanse los manuales para ver si su compilador tiene un IDE disponible.
2. Use el teclado para teclear el código fuente de HELLO.C exactamente como se muestra en el listado 1.1. Oprima Enter al final de cada línea de código.
3. Guarde el código fuente. A este archivo debe darle el nombre HELLO.C.
4. Verifique que HELLO.C se encuentre en el disco haciendo un listado de los archivos del directorio. Se debe ver a HELLO.C dentro de este listado.
5. Compile y enlace a HELLO.C. Ejecute el comando adecuado que especifique el manual de su compilador. Se debe obtener un mensaje que indique que no hubo errores.

6. Revise los mensajes del compilador. Si no se reciben errores, todo debe estar correcto.

   Pero, ¿qué pasa si se comete algún error al teclear el programa? El compilador se da cuenta de ello y despliega un mensaje de error en la pantalla. Por ejemplo, si en vez de teclear la palabra `printf` se tecleó `prntf`, se desplegará un mensaje similar al siguiente:

   ```
   Error: símbolos no definidos: _prntf en hello.c (hello.OBJ)
   ```

7. Regrese al paso 2, si se despliega éste o cualquier otro mensaje de error. Abra el archivo HELLO.C en el editor. Compare cuidadosamente el contenido del archivo con el listado 1.1 de este capítulo, haga cualquier corrección necesaria y continúe con el paso 3 y los siguientes.

8. Su primer programa de C ahora debe compilar y estar listo para ejecutar. Si se despliega un listado del directorio de todos los archivos llamados HELLO (y que tengan cualquier extensión), se deberá ver lo siguiente:

   - ❑ HELLO.C (que es el archivo de código fuente que se creó con el editor).
   - ❑ HELLO.OBJ o HELLO.O (que contiene el código objeto de HELLO.C).
   - ❑ HELLO.EXE (que es el programa ejecutable creado cuando se compiló y enlazó a HELLO.C).

9. Para correr o ejecutar a HELLO.EXE, simplemente teclee hello. El mensaje `Hello, world.` es desplegado en la pantalla.

¡Felicidades! Ya ha tecleado, compilado y ejecutado su primer programa en C. Evidentemente, HELLO.C es un programa simple que no hace nada útil, pero es bueno para un comienzo. De hecho, debe recordar que la mayoría de los expertos programadores de C de ahora comenzaron aprendiendo C de esta misma forma, compilando HELLO.C, por lo que está usted en buena compañía.

Errores de compilación

Un *error de compilación* sucede cuando el compilador encuentra algo en el código fuente que no puede compilar. Una palabra mal escrita, un error de tecleo o cualquier otra cosa puede hacer que el compilador se atragante. Afortunadamente los compiladores modernos no solamente se atragantan, ¡sino que dicen qué es lo que los está atragantando y dónde se encuentra! Esto facilita encontrar y corregir los errores en el código fuente.

Esto puede ilustrarse introduciendo un error en forma deliberada en HELLO.C. Si se hizo este ejemplo (tal como debiera), ahora tiene una copia de HELLO.C en el disco. Usando el editor, mueva el cursor al final de la línea que contiene la llamada a `printf()` y borre el punto y coma de terminación. HELLO.C ahora debe verse como en el listado 1.2.

Captura **Listado 1.2. HELLO.C with an error.**

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf("Hello, World!")
6: }
```

Ahora guarde el archivo. Se encuentra listo para compilar el archivo. Hágalo tecleando el comando para el compilador. Debido al error que se ha cometido, la compilación no puede completarse. En vez de ello, el compilador despliega un mensaje en la pantalla similar al siguiente:

```
hello.c(6) : Error: ';' esperado
```

Viendo esta línea, se puede ver que consta de tres partes.

```
hello.c, es el nombre del archivo donde se encuentra el error
(6), el número de línea donde se encuentra el error
Error: ';' expected, una descripción del error
```

Esto es bastante informativo, diciéndole que en la línea 6 de HELLO.C el compilador esperaba encontrar un punto y coma, pero no lo encontró. Claro, usted sabe que el punto y coma fue quitado de la línea 5, y hay una discrepancia. Nos encontramos con la incongruencia de por qué el compilador reporta un error en la línea 6, cuando de hecho el punto y coma fue omitido en el final de la línea 5. La respuesta se encuentra en el hecho de que el C no toma en cuenta cosas como los cortes entre líneas. El punto y coma que corresponde después del enunciado `printf()` pudo haber sido puesto en la siguiente línea (aunque hacerlo así hubiera sido una mala práctica de programación). Sólo después de haber llegado a la llave de la línea 6 el compilador está seguro de que le falta el punto y coma; por lo tanto, reporta que el error está en la línea 6.

Esto saca a relucir un hecho innegable acerca de los compiladores de C y los mensajes de error. Aunque el compilador es muy listo acerca de la detección y localización de errores, no es Einstein. Uno debe usar su conocimiento de lenguaje C, interpretar los mensajes del compilador y determinar la posición actual de cualquier error que reporte. A veces se encuentra en la línea reportada por el compilador y, en caso de no ser así, casi siempre se encuentra en la línea anterior. Al principio cuesta un poco de trabajo encontrar los errores, pero luego le será más fácil.

Antes de abandonar este tema, veamos otro ejemplo de un error de compilación. Cargue HELLO.C nuevamente en el editor y haga los siguientes cambios:

1. Reemplace el punto y coma al final de la línea 5.
2. Borre las comillas que se encuentran antes de la palabra `Hello`.

Guarde el archivo en disco y vuelva a compilar el programa. En esta ocasión, el compilador debe desplegar mensajes de error similares a los siguientes:

```
hello.c(5) : Error: identificador no definido 'Hello'
hello.c(6) : Error léxico: cadena no terminada
Error léxico: cadena no determinada
Error léxico: cadena no determinada

Error fatal: final prematuro del archivo fuente
```

El primer mensaje de error encuentra al error correctamente, ubicándolo en la línea 5 en la palabra `Hello`. El mensaje de error `unidentified identifier` significa que el compilador no sabe qué hacer con la palabra `Hello`, ya que no está entre comillas. Sin embargo, ¿qué hay acerca de los otros cuatro errores que reporta? Estos errores, de cuyo significado no tenemos que preocuparnos ahora, ilustran el hecho de que un solo error en un programa C algunas veces puede causar varios mensajes de error.

La lección por aprender de todo esto es la siguiente: si el compilador reporta varios errores y solamente se puede encontrar uno, siga adelante, corrija el error y vuelva a compilar. Tal vez encuentre que una sola corrección es todo lo que se necesita y que ahora el programa compila sin errores.

Mensajes de error del enlazador

Los errores del enlazador son relativamente raros y, por lo general, se deben a errores de escritura del nombre de una función de biblioteca de C. En este caso, se obtiene el mensaje de error `Error: undefined symbols:`, seguido del nombre mal tecleado (y precedido por un signo de subrayado). Una vez que se corrige la palabra, el problema debe desaparecer.

Resumen

Después de leer este capítulo, debe usted tener confianza de que la selección del C como su lenguaje de programación es una buena selección. El C proporciona una combinación de poder, popularidad y portabilidad sin paralelo. Estos factores, junto con la íntima relación del C con el nuevo lenguaje orientado a objetos C++, hacen al C inmejorable.

Este capítulo ha explicado los varios pasos involucrados en la escritura de un programa en C, el proceso conocido como desarrollo de programa. Se debe tener una clara comprensión del ciclo editar-compilar-enlazar-probar, así como de las herramientas que se han de usar para cada paso.

Los errores son una parte inevitable del desarrollo de programas. El compilador de C detecta errores en el código fuente y despliega mensajes de error dando la naturaleza y la ubicación del error. Con esta información se puede editar el código fuente para corregir el error. Sin

embargo, recuerde que el compilador no siempre puede reportar con precisión la naturaleza y ubicación del error. Algunas veces necesitará usar su conocimiento del C para localizar exactamente lo que está causando un determinado mensaje de error.

Preguntas y respuestas

1. Si quiero darle a alguien un programa que escribí, ¿qué archivos debo darle?

   Una de las cosas buenas acerca del C es que es un lenguaje compilado. Esto significa que después de que el código fuente es compilado, se tiene un programa ejecutable. Este programa ejecutable es un programa aislado. Si quiere dar HELLO a todos sus amigos que tengan computadora, lo puede hacer. Todo lo que necesita darles es el programa ejecutable HELLO.EXE. Ellos no necesitan el archivo fuente HELLO.C ni el archivo objeto HELLO.OBJ. ¡Incluso ni necesitan tener su propio compilador C!

2. Después de haber creado un archivo ejecutable, ¿necesito guardar el archivo fuente (.C) o el archivo objeto (.OBJ)?

   Si se deshace del archivo fuente, no tiene manera de hacer cambios al programa en el futuro; por lo tanto, ¡debe guardar este archivo! Los archivos objeto son una cosa aparte. Hay razones para guardar los archivos objeto. Sin embargo, por el momento está fuera del alcance de lo que estamos haciendo. Por ahora, usted puede deshacerse de los archivos objeto una vez que tenga el archivo ejecutable. Si necesita el archivo objeto, se puede recompilar el archivo fuente.

3. Si mi compilador viene con un editor, ¿tengo que usarlo?

   Definitivamente no. Se puede usar cualquier editor, siempre y cuando guarde el código fuente en formato de texto. Si el compilador viene con un editor, trate de usarlo. Si usted tiene un editor mejor, úselo. Yo (Brad) uso un editor que compré por separado, aunque todos mis compiladores tienen sus propios editores. Los editores que vienen con los compiladores son cada vez mejores. Algunos de ellos formatean automáticamente el código C. Otros codifican con color diferentes partes del archivo fuente para facilitar la búsqueda de errores.

4. ¿Puedo ignorar los mensajes de advertencia?

   Algunos mensajes de advertencia no afectan la manera en que un programa ejecuta, pero otros sí. Si el compilador le da un mensaje de advertencia, es señal de que algo no está completamente bien. La mayoría de los compiladores le permiten ajustar el nivel de mensajes de advertencia. Ajustando el nivel de los mensajes, se pueden obtener solamente los mensajes más delicados u obtener todos los mensajes, aun los más banales. Algunos compiladores hasta le dan

varios niveles intermedios. En los programas se debe ver cada mensaje y tomar una determinación. Siempre es mejor tratar de escribir todos los programas sin que aparezca ningún mensaje de advertencia o de error. (Con un mensaje de error el compilador no creará el archivo ejecutable.)

Taller

El taller le proporciona preguntas que le ayudarán a afianzar su comprensión del material tratado así como ejercicios que le darán experiencia en el uso de lo aprendido. Trate de comprender el cuestionario y dé las respuestas antes de continuar al siguiente capítulo. Las respuestas se proporcionan en el apéndice G, "Respuestas".

Cuestionario

1. Dé tres razones por las cuales el C es la mejor selección de lenguaje de programación.
2. ¿Qué hace el compilador?
3. ¿Cuáles son los pasos en el ciclo de desarrollo en el programa?
4. ¿Qué comando se necesita teclear para compilar un programa llamado PROGRAM1.C en su compilador?
5. ¿Su compilador ejecuta el enlazado y la compilación con un solo comando o se tienen que dar comandos separados?
6. ¿Qué extensión se debe usar para los archivos fuente del C?
7. ¿Es FILENAME.TXT un nombre válido para un archivo fuente del C?
8. Si se ejecuta un programa que se ha compilado y no funciona como se esperaba, ¿qué se debe hacer?
9. ¿Qué es el lenguaje de máquina?
10. ¿Qué hace el enlazador?

Ejercicios

1. Use el editor de texto para ver el archivo objeto creado por el listado 1.1. ¿Se parece el archivo objeto al archivo fuente? (No guarde este archivo cuando salga del editor.)
2. Teclee el siguiente programa y compílelo. ¿Qué hace este programa? (No incluya los números de línea.)

```
1: #include <stdio.h>
2:
3: int radius, area;
4:
5: main()
6: {
7:     printf( "Enter radius (i.e. 10): " );
8:     scanf( "%d", &radius );
9:     area = 3.14159 * radius * radius;
10:    printf( "\n\nArea = %d", area );
11:    return 0;
12: }
```

3. Teclee y compile el siguiente programa. ¿Qué hace este programa?

```
1: #include <stdio.h>
2:
3: int x,y;
4:
5: main()
6: {
7:     for ( x = 0; x < 10; x++, printf( "\n" ) )
8:         for ( y = 0; y < 10; y++ )
9:             printf( "X" );
10:
11:    return 0;
12: }
```

4. BUSQUEDA DE ERRORES: El siguiente programa tiene un problema. Tecléelo en el editor y compílelo. ¿Qué línea genera mensajes de error?

```
1: #include <stdio.h>
2:
3: main();
4: {
5:     printf( "Keep looking!" );
6:     printf( "You\'ll find it!" );
7:     return 0;
8: }
```

5. BUSQUEDA DE ERRORES: El siguiente programa tiene un problema. Tecléelo en el editor y compílelo. ¿Qué línea da problemas?

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf( "This is a program with a " );
6:     do_it( "problem!");
7:     return 0;
8: }
```

6. Haga los siguientes cambios al programa del ejercicio número 3. Vuélvalo a compilar y ejecute este programa. ¿Qué hace ahora el programa?

```
9:              printf( "%c", 1 );
```

7. Teclee y compile el siguiente programa. Este programa puede usarse para imprimir sus listados. Si se tienen errores, asegúrese de haber tecleado el programa correctamente.

   El uso de este programa es PRINT_IT nombre de *archivo.ext*, donde nombre de archivo.ext es el nombre de archivo fuente junto con su extensión. Observe que este programa añade números de línea al listado. (No se preocupe por la longitud de este programa; no espero que lo entienda todavía. Se incluye aquí para ayudarle a comparar las impresiones de sus programas con las que se dan en el libro.)

```
1:  /* PRINT_IT.C- Este programa imprime un listado con números de
           línea*/
2:
3:  #include <stdio.h>
4:
5:  void do_heading(char *filename);
6:
7:  int line, page;
8:
9:  main( int argv, char *argc[] )
10: {
11:    char buffer[256];
12:    FILE *fp;
13:
14:    if( argv < 2 )
15:    {
```

```
16:     fprintf(stderr, "\nProper Usage is: " );
17:     fprintf(stderr, "\n\nPRINT_IT filename.ext\n" );
18:       exit(1);
19:   }
20:
21:   if (( fp = fopen( argc[1], "r" )) == NULL )
22:   {
23:     fprintf( stderr, "Error opening file, %s!", argc[1]);
24:       exit(1);
25:   }
26:
27:   page = 0;
28:   line = 1;
29:   do_heading( argc[1]);
30:
31:   while( fgets( buffer, 256, fp ) != NULL )
32:   {
33:      if( line % 55 == 0 )
34:         do_heading( argc[1] );
35:
36:      fprintf( stdprn, "%4d:\t%s", line++, buffer );
37:   }
38:
39:   fprintf( stdprn, "\f" );
40:   fclose(fp);
41:   return 0;
42:   }
43:
44:   void do_heading( char *filename )
45:   {
46:    page++;
47:
48:    if ( page > 1)
49:       fprintf( stdprn, "\f" );
50:
51:    fprintf( stdprn, "Page: %d, %s\n\n", page, filename );
52:   }
```

DIA 2

Los componentes de un programa C

Cada programa en C consiste en varios componentes combinados de cierta forma. La mayor parte de este libro está dedicada a explicar estos diversos componentes del programa y la manera en que se les usa. Sin embargo, para tener la visión general se debe comenzar viendo un programa en C completo (aunque pequeño) donde se identifique a todos sus componentes. Hoy aprenderá

- ❑ Un pequeño programa en C con la identificación de sus componentes.
- ❑ El objeto de cada componente del programa.
- ❑ A compilar y ejecutar un programa de ejemplo.

Un pequeño programa en C

El listado 2.1 presenta el código fuente para MULTIPLY.C. Este es un programa muy simple; todo lo que hace es recibir dos números desde el teclado y calcular su producto. En este momento no se preocupe acerca de la comprensión de los detalles del funcionamiento del programa. El objetivo es familiarizarse con las partes de un programa en C, para que se pueda tener una mejor comprensión de los listados que se presentan posteriormente en el libro.

Antes de ver el programa de ejemplo, se necesita saber lo que es una función, como las funciones son el punto medular de la programación en C. Una *función* es una sección independiente de código de programa, que ejecuta una tarea determinada y a la que se le ha asignado un nombre. Al hacer referencia al nombre de la función, el programa puede ejecutar el código que se encuentra en la función. El programa también puede enviar información, llamada *argumentos,* a la función, y ésta puede regresar información al programa. Los dos tipos de funciones de C son *funciones de biblioteca,* que son parte del paquete del compilador C, y las *funciones definidas por el usuario,* que, el programador, crea. Se aprenderá acerca de ambos tipos de función en este libro.

Tome en cuenta que los números de línea que aparecen en el listado 2.1, así como en todos los listados de este libro, no son parte del programa. Han sido incluidos solamente para propósitos de identificación.

Captura

Listado 2.1. MULTIPLY.C.

```
1: /* Programa para calcular el producto de dos números. */
2: #include <stdio.h>
3: int a,b,c;
4: int product(int x, int y);
5: main()
6: {
7:     /* Pide el primer número */
8:     printf("Enter a number between 1 and 100: ");
9:     scanf("%d", &a);
10:
```

```
11:     /* Pide el segundo número */
12:     printf("Enter another number between 1 and 100: ");
13:     scanf("%d", &b);
14:
15:     /* Calcula y despliega el producto */
16:     c = product(a, b);
17:     printf ("\n%d times %d = %d", a, b, c);
18: }
19:
20: /* Función que regresa el producto de sus dos argumentos */
21: int product(int x, int y)
22: {
23:     return (x * y);
24: }
```



La salida del listado 2.1 es

```
Enter a number between 1 and 100: 35
Enter another number between 1 and 100: 23

35 times 23 = 805
```

Los componentes de un programa

Los siguientes párrafos describen los diversos componentes del programa de ejemplo anterior. Se incluyen los números de línea, para que de esta manera pueda identificar fácilmente las partes del programa que se están tratando.

La función *main()* (líneas 5-18)

El único componente que es obligatorio en cada programa en C es la función `main()`. En su forma más simple la función `main()` consiste en el nombre `main`, seguido por un par de paréntesis vacíos (()) y un par de llaves ({ }). Dentro de las llaves se encuentran enunciados que forman el cuerpo principal del programa. Bajo circunstancias normales la ejecución del programa comienza con el primer enunciado de `main()` y termina con el último enunciado de `main()`.

La directiva *#include* (línea 2)

La directiva `#include` da instrucciones al compilador C para que añada el contenido de un archivo de inclusión al programa durante la compilación. Un archivo *de inclusión* es un archivo de disco separado que contiene información necesaria para el compilador. Varios de estos archivos (algunas veces llamados *archivos de encabezado)* se proporcionan con el compilador. Nunca se necesita modificar la información de estos archivos y ésta es la razón por la cual se mantienen separados del código fuente. Todos los archivos de inclusión deben tener la extensión .H (por ejemplo, STDIO.H).

Se usa la directiva `#include` para darle instrucciones al compilador que añada un archivo de inclusión específico al programa durante la compilación. La directiva `#include`, en este programa de ejemplo, significa "añada el contenido del archivo STDIO.H". La mayoría de los programas en C requieren uno o más archivos de inclusión. Se dará mayor información acerca de los archivos de inclusión que es dada en el Día 21, "Aprovechando las directivas del preprocesador y más".

Definición de variables (línea 3)

Una *variable* es un nombre asignado a una posición de almacenamiento de datos. El programa utiliza variables para guardar varios tipos de datos durante la ejecución del programa. En C, una variable debe ser definida antes de que pueda ser usada. Una *definición de variable* le informa al compilador el nombre de la variable y el tipo de datos que va a guardar. En el programa de ejemplo la definición de la línea 3, `int a,b,c;`, define tres variables, llamadas a, b y c, que guardarán cada una un valor *entero*. Se presentará más información acerca de las variables y las definiciones de variables en el Día 3, "Variables y constantes numéricas".

Prototipo de función (línea 4)

Un *prototipo de función* proporciona al compilador C el nombre y los argumentos de una función contenida en el programa, y debe aparecer antes de que la función sea usada. Un prototipo de función es diferente de una *definición de función*, que contiene las instrucciones actuales que hacen a la función. (Las definiciones de función se tratan a mayor detalle, posteriormente, en este capítulo.)

Enunciados del programa (líneas 8, 9, 12, 13, 16, 17, 23)

El trabajo real de un programa C es hecho por sus *enunciados*. Los enunciados de C despliegan información en la pantalla, leen entrada del teclado, ejecutan operaciones matemáticas, llaman funciones, leen archivos de disco y hacen todas las otras operaciones que un programa necesita ejecutar. La mayor parte de este libro está dedicada a enseñarle los diversos enunciados de C. Por el momento, recuerde que en el código fuente los enunciados de C son escritos uno por línea y siempre terminan con un punto y coma. Los enunciados en MULTIPLY.C se explicarán brevemente en las siguientes secciones.

printf()

El enunciado `printf()` (líneas 8, 12 y 17) es una función de biblioteca que despliega información en la pantalla. El enunciado `printf()` puede desplegar un simple mensaje de

texto (tal como sucede en las líneas 8 y 12) o un mensaje y el valor de una o más variables del programa (tal como sucede en la línea 17).

scanf()

El enunciado `scanf()` (líneas 9 y 13) es otra función de biblioteca. Ella lee datos desde el teclado y asigna los datos a una o más variables del programa.

c = product (a,b);

Este enunciado del programa *llama* a la función denominada `product()`. Esto es, ejecuta los enunciados de programa contenidos en la función product(). También envía los *argumentos* a y b a la función. Después de que se completa la ejecución de los enunciados que se encuentran en `product()`, `product()` regresa un valor al programa. Este valor es guardado en la variable llamada c.

*return (x * y);*

Este enunciado es parte de la función `product()`. Este calcula el producto de las variables x y y, y regresa el resultado al programa que llamó a `product()`.

Definición de función (líneas 21-24)

Una *función* es una sección de código independiente y autocontenida que es escrita para ejecutar determinada tarea. Cada función tiene un nombre, y el código de cada función es ejecutado, incluyendo el nombre de la función, en una instrucción de programa. A esto se le llama *llamado* de la función.

La función denominada `product()`, que se encuentra en las líneas 21 a 24 en el listado 2.1, es una función *definida por el usuario*. Tal como lo indica su nombre, las funciones definidas por el usuario son escritas por el programador durante el desarrollo del programa. Esta función es simple, ya que todo lo que hace es multiplicar dos valores y regresar la respuesta al programa que la llamó. En el Día 5, "Funciones: lo básico", aprenderá que el uso adecuado de las funciones es una parte importante de la programación correcta en C.

Tome en cuenta que en un programa real en C probablemente no usará una función para una tarea tan simple como la multiplicación de dos números. Aquí lo hacemos solamente para efectos de demostración.

El C también incluye *funciones de biblioteca* que son parte del paquete del compilador C. Las funciones de biblioteca ejecutan la mayoría de las tareas comunes (como la entrada/salida de la pantalla, el teclado y disco) que necesita el programa. En el programa de ejemplo, `printf()` y `scanf()` son funciones de biblioteca.

Comentarios del programa (líneas 1, 7, 11, 15, 20)

Cualquier parte del programa que comienza con /* y termina con */ es llamado un *comentario.* El compilador ignora todos los comentarios y, por lo tanto, no tienen ningún efecto sobre la manera en que funciona el programa. Se puede poner lo que se quiera en un comentario, y esto no modificará la manera en que trabaja el programa. Un comentario puede ocupar parte de una línea, una línea completa o varias líneas. Algunos ejemplos son

```
/* Un comentario de una sola línea */
int a,b,c; /* Un comentario de una línea parcial */
/* Un
comentario
de varias
líneas */
```

Sin embargo, no se deben usar comentarios anidados (lo que significa que no se debe incluir un comentario dentro de otro). La mayoría de los compiladores no aceptarán lo siguiente:

```
/*
/* Comentario anidado */
*/
```

Sin embargo, algunos compiladores sí permiten los comentarios anidados. Aunque esta característica puede ser tentadora, le sugerimos que la evite. Como uno de los beneficios del C es su portabilidad, usar una característica como los comentarios anidados puede limitar la portabilidad del código. Los comentarios anidados también pueden dar lugar a problemas difíciles de encontrar.

Muchos programadores novatos consideran innecesarios los comentarios de programa y creen que son una pérdida de tiempo. ¡Este es un error! La operación del programa puede ser muy clara cuando se está escribiendo, en particular cuando se escriben programas simples. Sin embargo, conforme se van haciendo más grandes y más complejos, o cuando se necesita modificar un programa que se escribió hace seis meses, considerará que los comentarios son muy valiosos. Este es el momento para desarrollar el hábito de usar comentarios libremente, para documentar todas las estructuras y operaciones del programa.

DEBE	NO DEBE

DEBE Añadir muchos comentarios al código fuente del programa, en especial cerca de los enunciados o funciones que pueden ser no muy claras para uno o para quien tenga que modificarlas posteriormente.

NO DEBE Añadir comentarios innecesarios a las instrucciones que ya son bastante claras. Por ejemplo, teclear

/ Lo siguiente imprime Hello World! en la pantalla */*
printf("Hello World!");

puede ser una exageración, sobre todo cuando ya se conoce bastante la función `printf()` y la manera en que funciona.

DEBE Aprender a desarrollar un estilo que le ayude. ¡Un estilo muy parco no ayuda y tampoco uno muy detallado, donde se ocupe más tiempo haciendo comentarios que programación!

Llaves (líneas 6, 18, 22, 24)

Se usan llaves ({}) para agrupar las líneas de programa que forman cada función de C, incluyendo la función `main()`. Un grupo de uno o más enunciados encerrados dentro de llaves es llamado un *bloque*. Como verá en los capítulos siguientes, el C tiene muchos usos para los bloques.

Ejecución del programa

Ahora tome su tiempo para teclear, compilar y ejecutar a MULTIPLY.C. Proporciona práctica adicional sobre el uso del editor y el compilador. Recuerde estos pasos que se mencionaron en el Día 1, "Comienzo".

1. Haga al directorio donde va a programar el directorio de trabajo.
2. Inicie el editor.
3. Teclee el código fuente para MULTIPLY.C, exactamente como se muestra en el listado 2.1, pero omita los números de línea.
4. Guarde el archivo de programa.
5. Compile y enlace el programa, dando los comandos adecuados para el compilador. Si no aparecen mensajes de error, se puede ejecutar el programa tecleando MULTIPLY en la línea de comandos.
6. Si aparece uno o más mensajes de error, regrese al paso 2 y corrija los errores.

Una nota sobre la precisión

Una computadora es rápida y precisa, pero también es completamente literal. No sabe lo suficiente para corregir el más simple error. Toma todo al pie de la letra y ¡no como se le quiso decir!

Esto también se aplica al código fuente C. Un simple error de tecleo en el programa puede hacer que el compilador C falle. Afortunadamente, aunque el compilador no es lo suficientemente listo para corregir los errores (y usted cometerá errores, ¡todo el mundo lo hace!), es lo suficientemente listo para reconocerlos como errores y reportarlos. La manera en que el compilador reporta los mensajes de error y la forma de interpretarlos, fue tratada en el Día 1, "Comienzo".

Revisión de las partes de un programa

Ahora que han sido descritas todas las partes del programa, usted deberá ser capaz de ver cualquier programa y encontrar algunas similitudes. Examine el listado 2.2, LIST_IT.C y vea si puede identificar las diferentes partes.

Captura

Listado 2.2. LIST_IT.C.

```
1:    /* LIST_IT.C - Este programa despliega un listado con números de línea */
2:    #include <stdio.h>
3:
4:    void display_usage(void);
5:
6:    int line;
7:
8:    main( int argv, char *argc[] )
9:    {
10:   char buffer[256];
11:   FILE *fp;
12:
13:   if( argv < 2 )
14:   {
15:      display_usage();
16:      exit(1);
17:   }
18:
19:   if (( fp = fopen( argc[1], "r" )) == NULL )
20:   {
21:        fprintf( stderr, "Error opening file, %s!", argc[1] );
22:        exit(1);
23:   }
24:
25:   line = 1;
26:
27:   while( fgets( buffer, 256, fp ) != NULL )
28:      fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
```

```
30:     fclose(fp);
31:     return 0;
32: }
33:
34: void display_usage(void)
35: {
36:      fprintf(stderr, "\nProper Usage is: " );
37:      fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
38: }
```

A continuación se presenta la salida del listado 2.2

Salida

```
E:\X>list_it list_it.c
1:    /* LIST_IT.C - Este programa despliega un listado con números
         de línea */
2:    #include <stdio.h>
3:
4:    void display_usage(void);
5:
6:    int line;
7:
8:    main( int argv, char *argc[] )
9:    {
10:        char buffer[256];
11:             FILE *fp;
12:
13:             if( argv < 2 )
14:             {
15:             display_usage();
16:             exit(1);
17:             }
18:
19:             if (( fp = fopen( argc[1], "r" )) == NULL )
20:             {
21:                  fprintf( stderr, "Error opening file, %s!",
                                              argc[1] );
22:             exit(1);
23:             }
24:
25:       line = 1;
26:
27:         while( fgets( buffer, 256, fp ) != NULL )
28:            fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
30:             fclose(fp);
31:                  return 0;
32:             }
33:
34:             void display_usage(void)
35:             {
```

```
36:             fprintf(stderr, "\nProper Usage is: " );
37:             fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
38:           }
```



LIST_IT.C es muy similar a PRINT_IT.C, que se tecleó en el ejercicio siete del Día 1, "Comienzo". El listado 2.2 despliega en la pantalla listados de programas C guardados, en vez de enviarlos a la impresora.

Viendo el listado se puede resumir dónde se encuentran las diferentes partes. La función obligatoria `main()` se encuentra en las líneas 8-32. En la línea 2 se tiene una directiva `#include`. Las líneas 6, 10 y 11 tienen definiciones de variables. Un prototipo de función, `void display_usage(void)`, se encuentra en la línea 4. Este programa tiene muchos enunciados (líneas 13, 15, 16, 19, 21, 22, 25, 27, 28, 30, 31, 36 y 37). Una definición de función para `display_usage()` ocupa las líneas 34-38. Las llaves encierran bloques por todo el programa. Por último, sólo la línea 1 tiene un comentario. ¡En la mayoría de los programas probablemente incluirá más de una línea de comentarios!

LIST_IT.C llama muchas funciones. Solamente llama una función definida por el usuario, `display_usage()`. Las funciones de biblioteca que usa son `exit()` en las líneas 16 y 22, `fopen()` en la línea 19, `printf()` en las líneas 21, 28, 36 y 37, `fgets()` en la línea 27 y `fclose()` en la línea 30. Estas funciones de biblioteca se tratarán a mayor detalle a lo largo de este libro.

Resumen

Este capítulo es corto pero importante, como presenta los componentes principales de un programa C. En él se aprendió que la única parte obligatoria de cada programa C es la función `main()`. También se aprendió que el trabajo real del programa es hecho por enunciados del programa, que le dicen a la computadora que ejecute las acciones deseadas. Este capítulo también presenta las variables y definiciones de variables, y muestra cómo usar comentarios en el código fuente.

Además de la función `main()` un programa en C puede usar dos tipos de funciones auxiliares: funciones de biblioteca, proporcionadas como parte del paquete del compilador, y funciones definidas por el usuario, creadas por el programador.

Preguntas y respuestas

1. ¿Qué efecto tienen los comentarios en un programa?

   Los comentarios son para el programador. Cuando el compilador convierte el código fuente a código objeto desecha los comentarios y espacios en blanco. Esto significa que ellos no tienen efecto en el programa ejecutable. Los comentarios

hacen que el archivo fuente sea más grande, pero por lo general esto no tiene importancia. Resumiendo, se deben usar comentarios y espacios en blanco para que sea fácil, en la medida de lo posible, la comprensión y el mantenimiento del código fuente.

2. ¿Cuál es la diferencia entre un enunciado y un bloque?

   Un bloque es un grupo de enunciados encerrados dentro de llaves ({}). Un bloque puede ser usado en muchos lugares donde puede ser usado un enunciado.

3. ¿Cómo se sabe cuáles funciones de biblioteca están disponibles?

   La mayoría de los compiladores vienen con un manual dedicado específicamente a la documentación de las funciones de biblioteca. Por lo general, vienen en orden alfabético. Otra manera de conocer las funciones de biblioteca disponibles es comprar un libro que las liste. El apéndice E, "Prototipos de función y archivos de encabezado", y el apéndice F, "Funciones comunes en orden alfabético", listan las funciones por categoría y, desde luego, en orden alfabético, respectivamente. Después de que comience a entender más del C, es buena idea leer estos apéndices para que no reescriba una función de biblioteca. (¡No vuelva a inventar el hilo negro!)

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Cómo se llama a un grupo de uno o más enunciados del C encerrados entre llaves?
2. ¿Cuál es el único componente obligatorio de todo programa en C?
3. ¿Cómo se añaden comentarios al programa y para qué se usan?
4. ¿Qué es una función?
5. El C proporciona dos tipos de funciones. ¿Qué son y cómo se diferencian?
6. ¿Para qué se usa la directiva `#include`?
7. ¿Se pueden anidar los comentarios?
8. ¿Los comentarios pueden ser más grandes que una línea?
9. ¿Qué otro nombre se le da a los archivos de inclusión?

10. ¿Qué es un archivo de inclusión?

Ejercicios

1. Escriba el programa más pequeño posible.
2. Usando el siguiente programa, conteste las preguntas:
   a. ¿Qué líneas contienen enunciados?
   b. ¿Qué líneas contienen definiciones de variables?
   c. ¿Qué líneas contienen prototipos de función?
   d. ¿Qué líneas contienen definiciones de función?
   e. ¿Qué líneas contienen comentarios?

```
1: /* EX2-2.C */
2: #include <stdio.h>
3:
4: void display_line(void);
5:
6: main()
7: {
8:     display_line();
9:     printf("\n Teach Yourself C In 21 Days!\n");
10:     display_line();
11:
12:     return 0;
13: }
14:
15: /* Imprime una línea de asteriscos */
16: void display_line(void)
17: {
18:     int counter;
19:
20:     for( counter = 0; counter < 21; counter++ )
21:         printf("*" );
22: }
23: /* Fin del programa */
```

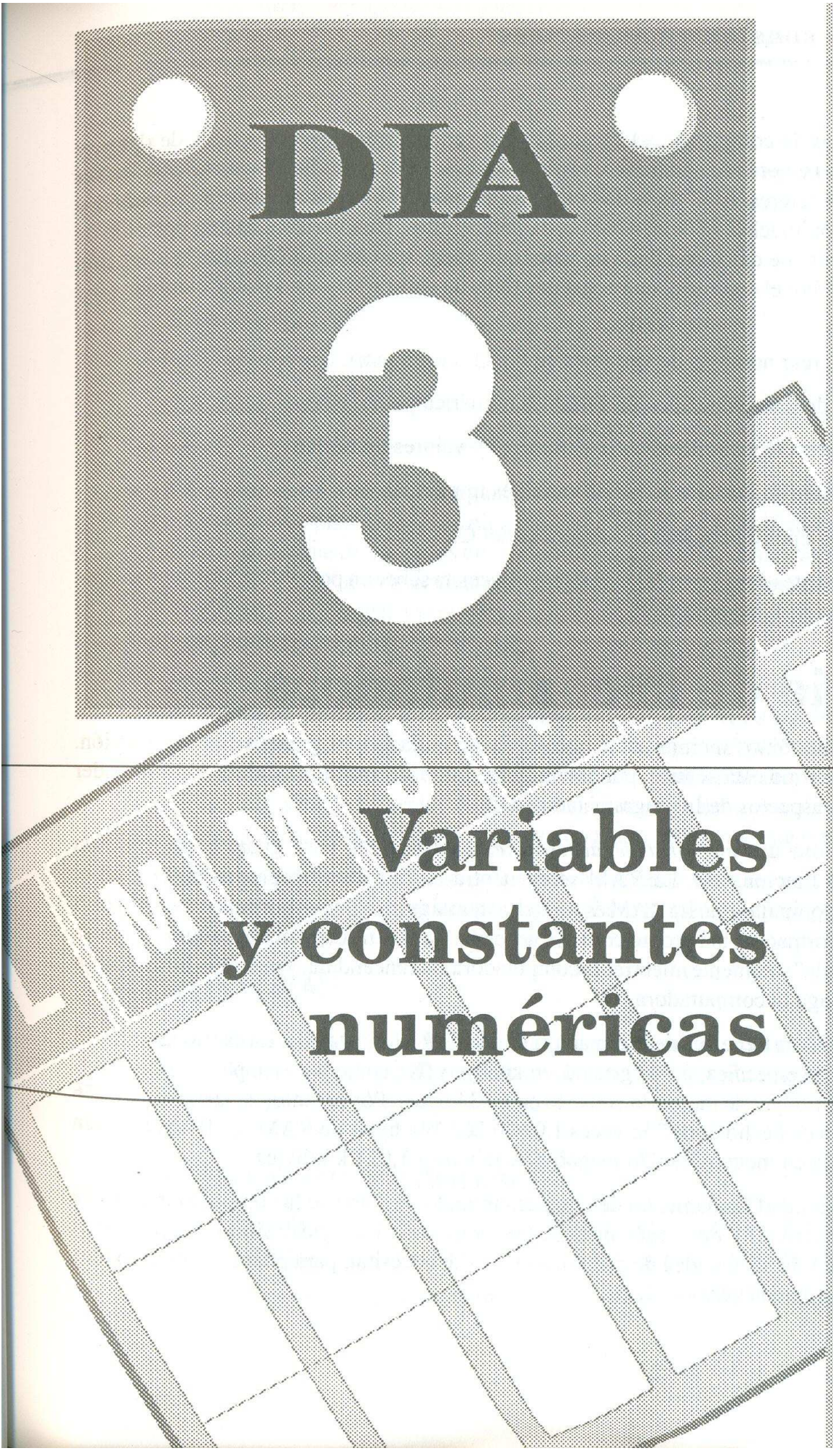
3. Escriba un ejemplo de un comentario.

4. ¿Qué hace el siguiente programa? (Tecléelo, compílelo y ejecútelo.)

```
1: /* EX2-4.C */
2: #include <stdio.h>
3:
4: main()
5: {
6:     int ctr;
7:
8:     for( ctr = 65; ctr < 91; ctr++ )
9:         printf("%c", ctr );
10:
11:     return 0;
12: }
13: /* Fin del programa */
```

5. ¿Qué hace el siguiente programa? (Tecléelo, compílelo y ejecútelo.)

```
1: /* EX2-5.C */
2: #include <stdio.h>
3: #include <string.h>
4: main()
5: {
6:     char buffer[256];
7:
8:     printf( "Enter your name and press <Enter>:\n");
9:     gets( buffer );
10:
11:     printf( "\nYour name has %d characters and spaces!",
                        strlen( buffer ));
12:
13:     return 0;
14: }
```



Variables y constantes numéricas

Los programas de computadora trabajan, por lo general, con diferentes tipos de datos, y necesitan una manera para guardar los valores que están usando. Estos valores pueden ser números o caracteres. El C tiene dos maneras de guardar valores numéricos, *variables* y *constantes,* con muchas opciones para cada una de ellas. Una variable es una posición de almacenamiento de datos que tiene un valor que puede ser cambiado durante la ejecución del programa. Por el contrario, una constante tiene un valor fijo que no puede cambiar. Hoy aprenderá

- ❑ Cómo crear nombres de variables en C.
- ❑ El uso de diferentes tipos de variables numéricas.
- ❑ La diferencia y similitud entre caracteres y valores numéricos.
- ❑ La manera de declarar e iniciar variables numéricas.
- ❑ Los dos tipos de constantes numéricas del C.

Sin embargo, antes de entrar a las variables se necesita saber un poco acerca de la operación de la memoria de la computadora.

Memoria de la computadora

Si usted ya sabe cómo funciona la memoria de la computadora, se puede saltar esta sección. Sin embargo, si no está seguro, por favor léala. Esta información ayudará a comprender mejor ciertos aspectos de la programación en C.

La computadora usa *memoria de acceso aleatorio* (RAM) para guardar información mientras está funcionando. La RAM se encuentra en circuitos integrados o *chips* en el interior de la computadora. La RAM es *volátil,* lo que significa que es borrada y reemplazada con nueva información tan pronto como se necesita. La volatilidad también significa que la RAM "recuerda" solamente mientras la computadora está encendida, y pierde su información cuando se apaga la computadora.

Cada computadora tiene una determinada cantidad de RAM instalada. La cantidad de RAM en un sistema se especifica, por lo general, en kilobytes (K), como por ejemplo, 256 K, 512 K o 640 K. Un kilobyte de memoria consiste en 1,024 bytes. Por lo tanto, un sistema con 256 K de memoria de hecho tiene 256 veces 1,024 ó 262,144 bytes de RAM. La RAM también es mencionada en megabytes. Un megabyte equivale a 1,024 kilobytes.

Un *byte* es la unidad fundamental del almacenamiento de datos de la computadora. El Día 20, "Otras funciones", tiene más información acerca de los bytes. Sin embargo, por el momento, para darse una idea de qué tantos bytes se necesitan para guardar determinados tipos de datos, puede ver la tabla 3.1.

Tabla 3.1. Espacio de memoria requerido para guardar datos.

Datos	Bytes requeridos
La letra *x*	1
El número *100*	2
El número *120.145*	4
La frase *Aprenda usted mismo C*	22
Una página escrita a máquina	3000 (aproximadamente)

La RAM en la computadora está organizada en forma secuencial, un byte tras otro. Cada byte de memoria tiene una *dirección* única mediante la cual es identificado, una dirección que también lo distingue de todos los otros bytes de la memoria. Las direcciones son asignadas a la memoria en orden, comenzando en 0 y aumentando hasta llegar al límite del sistema. Por el momento no necesita preocuparse acerca de las direcciones, ya que son manejadas automáticamente por el compilador C.

¿Para qué se usa la RAM de la computadora? Tiene varios usos, pero solamente uno, el almacenamiento de datos, le interesa al programador. Los *datos* significan la información con la cual trabaja el programa en C. Ya sea que el programa esté trabajando con una lista de direcciones, monitoreando la bolsa de valores, manejando un presupuesto familiar o cualquier otra cosa, la información (nombres, precios de acciones, gastos o lo que sea) es guardada en la RAM de la computadora mientras el programa esté ejecutando.

Ahora que ya entiende un poco acerca del almacenamiento de memoria, podemos regresar a la programación en C y la manera en que el C usa la memoria para guardar información.

Variables

Una *variable* es una posición de almacenamiento de datos de la memoria de la computadora que tiene un nombre. Al usar un nombre de variable en el programa de hecho se está haciendo referencia al dato que se encuentra guardado ahí.

Nombres de variable

Para usar variables en los programas en C se debe saber cómo crear nombres de variables. En C, los nombres de variables se deben ajustar a las siguientes reglas:

- El nombre puede contener letras, dígitos y el carácter de subrayado (_).

- El primer carácter del nombre debe ser una letra. El carácter de subrayado también es un carácter inicial aceptado, pero no se recomienda su uso.
- Tiene importancia el uso de mayúsculas y minúsculas. Por lo tanto, los nombres `contador` y `Contador` hacen referencia a dos variables diferentes.
- Las palabras claves del C no pueden usarse como nombres de variable. Una palabra clave es una palabra que es parte del lenguaje C. (Una lista completa de las 33 palabras claves del C está en el apéndice B, "Palabras reservadas del C".)

El siguiente código contiene algunos ejemplos de nombres de variable de C legales e ilegales:

```
porcentaje              /* legal */
y2x5_fg7h               /* legal */
utilidades_anuales      /* legal */
_1990_tax               /* legal pero no recomendable */
cuenta#gasto            /* ilegal: contiene el carácter ilegal # */
double                  /* ilegal: es una palabra clave del C */
9winter                 /* ilegal: el primer carácter es un dígito */
```

Debido a que el C toma en cuenta las mayúsculas y las minúsculas, los tres siguientes nombres, porcentaje, `PORCENTAJE` y Porcentaje, se considera que hacen referencia a tres variables distintas. Los programadores de C, por lo general, usan solamente minúsculas en los nombres de variable, aunque no es obligatorio. Las mayúsculas se reservan por lo general para los nombres de constantes (tratadas posteriormente, en este capítulo).

Para muchos compiladores un nombre de variable de C puede ser hasta de 31 caracteres de largo. (De hecho, pueden ser más largos que esto, pero el compilador solamente toma en cuenta los 31 primeros caracteres del nombre.) Con esta flexibilidad se pueden crear nombres de variable que reflejen los datos que están siendo guardados. Por ejemplo, un programa que calcula los pagos de un préstamo puede guardar el valor de la tasa de interés en una variable llamada `tasa_interés`. El nombre de variable ayuda a aclarar su uso. También se podría haber creado un nombre de variable como x o `juan_perez`, ya que no le importa al compilador de C. Sin embargo, el uso de la variable no será tan claro para cualquier otra persona que vea el código fuente. Aunque puede llevar algo más de tiempo teclear nombres de variable descriptivos, la mejora en claridad del programa hace que valga la pena.

Se usan muchas convenciones de denominación para los nombres de variables creados con varias palabras. Ya ha visto un estilo: `tasa_interés`. Al usar un carácter de subrayado para separar palabras en los nombres de variable se facilita la interpretación. El segundo estilo es la *notación de camello*. En vez de usar espacios, se pone en mayúscula la primera letra de cada palabra. En vez de `tasa_interés`, la variable sería nombrada `TasaInterés`. La *notación de camello* está ganando popularidad, ya que es más fácil teclear una mayúscula que un subrayado. Usaremos el subrayado en este libro, porque es más fácil de leer para la mayoría de la gente. Usted decidirá cuál estilo prefiere adoptar.

DEBE	NO DEBE

DEBE Usar nombres de variable que sean descriptivos.

DEBE Adoptar un estilo para nombrar las variables y sígalo.

NO DEBE Comenzar los nombres de variable con el carácter subrayado innecesariamente.

NO DEBE Usar nombres de variables en mayúsculas innecesariamente.

Tipos de variables numéricas

El C proporciona varios tipos diferentes de variables numéricas. ¿Para qué se necesitan diferentes tipos de variables? Diferentes valores numéricos tienen requisitos de almacenamiento de memoria variables, y difieren en la facilidad con que ciertas operaciones matemáticas pueden ser ejecutadas con ellos. Los números enteros pequeños (por ejemplo, 1, 199, -8) requieren menos espacio de memoria para almacenamiento, y las operaciones matemáticas (suma, multiplicación, etc.) con esos números pueden ser rápidamente ejecutadas por la computadora . En contraste, los enteros largos y los valores de punto flotante (123,000,000 o 0.000000871256, por ejemplo) requieren más espacio de almacenamiento y más tiempo para las operaciones matemáticas. Usando los tipos de variables adecuados se asegura que el programa ejecuta lo más eficientemente posible.

Las variables numéricas del C caen en las siguientes dos categorías principales:

- Las *variables enteras* guardan valores que no tienen fracciones (esto es, solamente números enteros). Las variables enteras son de dos tipos: las variables enteras con signo pueden guardar valores positivos o negativos, y en cambio las variables enteras sin signo solamente pueden guardar valores positivos (y 0, por supuesto).
- Las *variables de punto flotante* guardan valores que tienen fracciones (esto es, números reales).

Dentro de estas categorías se encuentran dos o más tipos específicos de variables. Ellos están resumidos en la tabla 3.2, que también muestra la cantidad de memoria en bytes que se requiere para guardar una sola variable de cada tipo cuando se usa una microcomputadora con arquitectura de 16 bits.

Tabla 3.2. Tipos de datos numéricos del C.

Tipo de variable	Palabra clave	Bytes requeridos	Rango
Carácter	char	1	-128 a 127
Entero	int	2	-32768 a 32767
Entero corto	short	2	-32768 a 32767
Entero largo	long	4	-2,147,483,648 a 2,147,483,647
Carácter sin signo	unsigned char	1	0 a 255
Entero sin signo	unsigned int	2	0 a 65535
Entero corto sin signo	unsigned short	2	0 a 65535
Entero largo sin signo	unsigned long	4	0 a 4,294,967,295
Punto flotante de precisión sencilla	float	4	1.2E-38 a 3.4E38[1]
Punto flotante de doble precisión	double	8	2.2E-308 a 1.8E308[2]

[1] Rango aproximado; precisión = 7 dígitos.

[2] Rango aproximado; precisión = 19 dígitos.

El *rango aproximado* (véase la tabla 3.2) significa los valores máximo y mínimo que puede guardar una variable dada. (Las limitaciones de espacio impiden listar los rangos exactos para los valores de cada una de estas variables.) *Precisión* significa la cantidad de dígitos con los cuales es guardada la variable. (Por ejemplo, si se evalúa 1/3, la respuesta es 0.33333... con un número de 3 hasta el infinito. Una variable con precisión de 7 guarda siete números 3.)

Al ver la tabla 3.2 puede darse cuenta de que los tipos de variable `int` y `short` son idénticos. ¿Por qué tienen dos tipos diferentes? Los tipos de variable `int` y `short` son idénticos solamente en los sistemas compatibles con la PC de IBM de 16 bits, pero pueden ser diferentes en otro tipo de hardware. En un sistema VAX, un `short` y un `int` no son del mismo tamaño. En este caso, un `short` es de dos bytes y un `int` es de cuatro. Recuerde que el C es un lenguaje flexible y portable, por lo que proporciona diferentes palabras claves para los dos tipos. Si se está trabajando en una PC se puede usar `int` y `short` indistintamente.

No se necesita palabra clave especial para hacer que una variable entera tenga signo, ya que las variables enteras por omisión tienen signo. Sin embargo, se puede incluir la palabra clave `signed` si se desea. Las palabras claves de la tabla 3.2 son usadas en las declaraciones de variable que se tratan en la siguiente sección de este capítulo.

El listado 3.1 le ayudará a determinar el tamaño de las variables en su computadora particular:

Captura

Listado 3.1. Un programa que despliega el tamaño de los tipos de variable.

```
1:   /* SIZEOF.C - Programa para obtener el tamaño de los tipos de */
2:   /*            variables del C en bytes  */
3:
4:   #include <stdio.h>
5:
6:   main()
7:   {
8:
9:        printf( "\nA char      is %d bytes", sizeof( char ));
10:       printf( "\nAn int      is %d bytes", sizeof( int ));
11:       printf( "\nA short     is %d bytes", sizeof( short ));
12:       printf( "\nA long      is %d bytes", sizeof( long ));
13:       printf( "\nAn unsigned char  is %d bytes", sizeof( unsigned
                  char ));
14:       printf( "\nAn unsigned int   is %d bytes", sizeof( unsigned
                  int ));
15:       printf( "\nAn unsigned short is %d bytes", sizeof( unsigned
                  short ));
16:       printf( "\nAn unsigned long  is %d bytes", sizeof( unsigned
                  long ));
17:       printf( "\nA float     is %d bytes", sizeof( float ));
18:       printf( "\nA double    is %d bytes", sizeof( double ));
19:
20:       return 0;
21:  }
```

Como muestra lo siguiente, la salida del listado 3.1 le dice exactamente qué tantos bytes ocupa cada tipo de variable en una computadora en particular. Si se está usando una PC de 16 bits, las cifras deben coincidir con las que se presentan en la tabla 3.2.

Salida

```
A char      is 1 bytes
An int      is 2 bytes
A short     is 2 bytes
A long      is 4 bytes
An unsigned char  is 1 bytes
An unsigned int   is 2 bytes
An unsigned short is 2 bytes
```

```
An unsigned long  is 4 bytes
A float           is 4 bytes
A double          is 8 bytes
```

Análisis No se preocupe en tratar de comprender todos los componentes individuales del programa. Aunque algunos conceptos son nuevos, como `sizeof()`, otros deben serle familiares. Las líneas 1 y 2 son comentarios acerca del nombre del programa y una breve descripción. La línea 4 incluye el archivo de encabezado estándar de entrada/salida, para ayudarle a imprimir la información en la pantalla. Este es un programa simple, ya que sólo contiene una sola función, `main()` (líneas 7-21). Las líneas 9-18 son el cuerpo del programa. Cada una de estas líneas imprime un texto de descripción con el tamaño de cada uno de los tipos de variable, lo cual se logra usando el operador `sizeof`. El Día 19, "Exploración de la biblioteca de funciones", trata a detalle al operador `sizeof`. La línea 20 del programa regresa el valor 0 al sistema operativo antes de terminar el programa.

El C garantiza ciertas cosas gracias al estándar ANSI. Hay cinco cosas con las que se puede contar.

- El tamaño de char es 1 byte.
- El tamaño de un `short` es menor que o igual al tamaño de un `int`.
- El tamaño de un `int` es menor que o igual al tamaño de un `long`.
- El tamaño de un `unsigned` es igual al tamaño de un `int`.
- El tamaño de un `float` es menor que o igual al tamaño de un `double`.

Declaración de variables

Antes de que pueda usar una variable en un programa C debe declararla. Una *declaración de variable* le informa al compilador el nombre y tipo de la variable, y opcionalmente inicia la variable a un valor específico. Si el programa trata de usar una variable que no ha sido declarada, el compilador genera un mensaje de error. Una declaración de variable tiene la siguiente forma:

```
nombre de tipo   nombre de variable;
```

`nombre de tipo` especifica el tipo de la variable y debe ser una de las palabras claves dadas en la tabla 3.2. *`nombre de variable`* es el nombre de la variable, que debe ajustarse a las reglas mencionadas anteriormente. Se pueden declarar varias variables del mismo tipo en una línea, separando los nombres de variable con comas.

```
int contador, número, inicio;        /* tres variables enteras */
float porcentaje, total;             /* dos variables flotantes */
```

En el Día 12, "Alcance de las variables", aprenderá que la posición de la declaración de la variable dentro del código fuente es importante, debido a que afecta la manera en la que el

programa usa las variables. Por el momento, puede poner todas las declaraciones de variable juntas, inmediatamente antes del comienzo de la función `main()`.

La palabra clave *typedef*

La palabra clave `typedef` es usada para crear un nuevo nombre para un tipo de dato existente. De hecho, `typedef` crea un sinónimo. Por ejemplo, el enunciado

```
typedef int entero;
```

crea `entero` como un sinónimo de `int`. Luego puede usar entero para definir variables de tipo `int`.

```
entero contador;
```

Tome en cuenta que `typedef` no crea un nuevo tipo de dato, sino que solamente permite usar un nombre diferente para un tipo de dato predefinido. El uso más común de `typedef` se refiere a los *tipos de datos agregados,* que son explicados en el Día 11, "Estructuras". Un tipo de dato agregado consiste en una combinación de los tipos de datos presentados en este capítulo.

Inicialización de variables numéricas

Cuando se declara una variable, se le da instrucción al compilador para que reserve espacio de almacenamiento para la variable. Sin embargo, el valor guardado en ese espacio, es decir, el valor de la variable, no está definido. Puede ser cero o algún valor de "basura" al azar. Antes de usar una variable siempre se le debe iniciar a un valor conocido. Esto puede hacerse en forma independiente a la declaración de la variable, usando un enunciado de asignación.

```
int contador;       /* Reserva espacio de almacenamiento para contador */
contador = 0;       /* Guarda 0 en contador */
```

Tome en cuenta que este enunciado usa el signo de igual (=), que es el operador de asignación del C y se trata más adelante en el Día 4, "Enunciados, expresiones y operadores". Por el momento no necesita tomar en cuenta que el signo de igual en programación no es lo mismo que el signo de igual en álgebra. Si se escribe

```
x = 12
```

en un enunciado algebraico, se está estableciendo un hecho: "*x* es igual a 12". Sin embargo, en C significa algo un poco diferente: "Asigne el valor 12 a la variable llamada x".

También se puede iniciar una variable cuando es declarada. Para hacerlo ponga a continuación del nombre de variable, en el enunciado de declaración, un signo de igual y el valor inicial deseado.

```
int contador = 0;
double porcentaje = 0.01, tasa_impuesto = 28.5;
```

Tenga cuidado de no iniciar una variable con un valor que se encuentre fuera del rango permitido. A continuación se presentan algunos ejemplos de iniciaciones fuera de rango:

```
int peso = 100000;
unsigned int valor = -2500;
```

El compilador de C no se da cuenta de estos errores. El programa puede compilar y encadenar pero se pueden obtener resultados inesperados cuando se ejecuta el programa.

DEBE	NO DEBE

DEBE Entender la cantidad de bytes que ocupan los tipos de variables en su computadora.

DEBE Usar `typedef` para hacer que el programa sea más legible.

DEBE Iniciar las variables cuando las declare siempre que sea posible.

NO DEBE Usar una variable que no ha sido iniciada. ¡Los resultados pueden ser impredecibles!

NO DEBE Usar una variable `float` o `double` si solamente está guardando enteros. Aunque funcionan, usarlas es ineficiente.

NO DEBE ¡Tratar de poner números en tipos de variables que son demasiado pequeños para guardarlos!

NO DEBE Poner números negativos en variables que tengan tipo `unsigned`.

Constantes

De manera similar a las variables, una *constante* es una posición de almacenamiento de datos usada por el programa. A diferencia de la variable, el valor guardado en una constante no puede ser cambiado durante la ejecución del programa. El C tiene dos tipos de constantes, teniendo cada una de ellas su uso específico.

Constantes literales

Una *constante literal* es un valor que es tecleado directamente en el código fuente cada vez que se necesita. A continuación se presentan dos ejemplos:

```
int contador = 20;
float tasa_impuesto = 0.28;
```

El 20 y el 0.28 son constantes literales. Los enunciados anteriores guardan estos valores en las variables contador y `tasa_impuesto`. Tome en cuenta que una de estas constantes contiene un punto decimal y la otra no. La presencia o ausencia del punto decimal distingue a las constantes de punto flotante de las constantes enteras.

Una constante literal escrita con un punto decimal es una *constante de punto flotante,* y es representada por el compilador C como un número de doble precisión. Las constantes de punto flotante pueden ser escritas en la notación decimal estándar, como se muestra en estos ejemplos:

```
123.456
0.019
100.
```

Observe la tercera constante, 100., que es escrita con un punto decimal aunque es un entero (esto es, no tiene parte fraccionaria). El punto decimal hace que el compilador C trate la constante como un valor de doble precisión. Sin el punto decimal, se trata como una constante entera.

Las constantes de punto flotante también pueden ser escritas en *notación científica.* Tal vez se acuerde, por las matemáticas de secundaria, que la notación científica representa a un número como una parte decimal multiplicada por 10 elevado a una potencia positiva o negativa. La notación científica es especialmente útil para representar valores extremadamente grandes y extremadamente pequeños. En C la notación científica es escrita como un número decimal seguido inmediatamente por una E o e y el exponente.

```
1.23E2      1.23 por 10 elevado al cuadrado, o 123
4.08e6      4.08 por 10 elevado a la sexta, o 4,080,000
0.85e-4     0.85 por 10 elevado a la menos cuatro, o 0.000085
```

Una constante escrita sin un punto decimal es representada por el compilador como un número entero. Las constantes enteras pueden ser escritas en tres notaciones diferentes:

- Una constante que comience con cualquier dígito diferente de 0 es interpretada como un entero *decimal* (esto es, el sistema de numeración estándar base 10). Las constantes decimales pueden contener los dígitos 0-9 y un signo de menos o de más al principio. (Cuando no tiene signo, se supone que la constante es positiva.)
- Una constante que comienza con el dígito 0 es interpretada como un entero *octal* (el sistema numérico de base 8). Las constantes octales pueden contener los dígitos 0-7 y un signo de menos o más al principio.
- Una constante que comienza con 0x o 0X es interpretada como una constante *hexadecimal* (el sistema numérico de base 16). Las constantes hexadecimales pueden contener los dígitos 0-9, las letras A-F y un signo de menos o de más al principio.



Nota: Véase el apéndice D: "Notación binaria y hexadecimal", para una explicación más completa de la notación decimal y hexadecimal.

Constantes simbólicas

Una *constante simbólica* es una constante que está representada por un nombre (símbolo) en el programa. De manera similar a una constante literal, una constante simbólica no puede cambiar. Cada vez que se necesite el valor de la constante en el programa se usa su nombre, como si se usara un nombre de variable. El valor actual de la constante simbólica solamente necesita ser dado una vez, cuando es definida por primera vez.

Las constantes simbólicas tienen dos ventajas importantes sobre las constantes literales, como lo muestra el siguiente ejemplo. Supongamos que se está escribiendo un programa que ejecuta varios cálculos geométricos. El programa necesita frecuentemente el valor de pi (3.14) para sus cálculos. (Tal vez recuerde de la geometría que pi es la relación de la circunferencia de un círculo a su diámetro.) Por ejemplo, para calcular la circunferencia y el área de un círculo de un radio conocido, se podría escribir

```
circunferencia = 3.14 * (2 * radio);
área = 3.14 * (radio)*(radio);
```

Observe que el asterisco (*) es el operador de multiplicación del C, y se trata en el Día 4, "Enunciados, expresiones y operadores". Por lo tanto, el primero de los enunciados anteriores significa "multiplique por dos el valor guardado en la variable `radio` y luego multiplique el resultado por 3.14. Por último, asigne el resultado a la variable llamada `circunferencia`".

Sin embargo, si se define una constante simbólica con el nombre PI y el valor 3.14 se podría escribir

```
circunferencia = PI * (2 * radio);
área = PI * (radio)*(radio);
```

El código resultante es más claro. En vez de andar adivinando a qué se refiere el valor 3.14, se ve inmediatamente que se usa la constante PI.

La segunda ventaja de las constantes simbólicas se manifiesta cuando se necesita cambiar una constante. Continuando con el ejemplo anterior, tal vez decida que para darle mayor precisión al programa necesita usar un valor de PI con más decimales: 3.14159 en vez de 3.14. Si se hubieran usado constantes literales en vez de PI se habría tenido que ir por todo el código fuente y cambiar cada aparición del valor 3.14 a 3.14159. Con una constante simbólica sólo necesita hacer un cambio en el lugar donde es definida la constante.

El C tiene dos métodos para definir una constante simbólica, la directiva `#define` y la palabra clave `const`. La directiva `#define` es una de las directivas del preprocesador de C, que se trata a fondo en el Día 21, "Aprovechando las directivas del preprocesador y más". La directiva `#define` es usada de la manera siguiente:

```
#define NOMBREDECONSTANTE literal
```

Esta línea de programa crea un nombre de constante llamado *NOMBREDECONSTANTE* con el valor de *literal*. *literal* representa una constante numérica, como se describió anteriormente en este capítulo. *NOMBREDECONSTANTE* sigue las mismas reglas descritas para los nombres de variable, anteriormente en este capítulo. Por convención, los nombres de constantes simbólicas se ponen en mayúsculas. Esto facilita el distinguirlas de los nombres de variable, que por convención se ponen en minúscula. Del ejemplo anterior, la directiva `#define` requerida sería

```
#define PI 3.14159
```

Observe que la línea `#define` no termina con punto y coma (;). `#define` puede ser puesto en cualquier lugar del código fuente, pero tiene efecto solamente para las partes de código fuente que se encuentran a continuación de la directiva `#define`. Por lo general, los programadores agrupan todos los `#defines` cerca del principio del archivo y antes del comienzo de `main()`.

La acción precisa de la directiva `#define` es dar instrucciones al compilador para que "en el código fuente reemplace a `NOMBREDECONSTANTE` con la literal". El efecto es exactamente el mismo que si se hubiera usado al editor para ir por todo el código fuente haciendo los cambios manualmente. Tome en cuenta que `#define` no reemplaza las apariciones del nombre que se dan como parte de nombres más largos, o cuando se encuentran encerradas entre comillas dobles o como parte de comentarios de programa.

```
#define PI 3.14
/* Se ha definido una constante para PI. */ no se cambia
#define PIPETTE 100                          no se cambia
```

La segunda manera de definir una constante simbólica es con la palabra clave `const`. `const` es un modificador que puede ser aplicado a cualquier declaración de variable. Una variable a la que se le aplica `const` no puede ser modificada durante la ejecución del programa, sino solamente iniciada al momento de la declaración. A continuación se presentan algunos ejemplos:

```
const int contador = 100;
const float pi = 3.14159;
const long deuda = 12000000, float tasa_impuesto = 0.21;
```

`const` afecta a todas las variables de la línea de declaración. En el último ejemplo `deuda` y `tasa_impuesto` son constantes simbólicas. Si el programa trata de modificar una variable `const`, el compilador genera un mensaje de error. Por ejemplo,

```
const int contador = 100;
contador = 200          /* ¡No compila! No se puede reasignar o
                    alterar el valor de una constante. */
```

¿Cuál es la diferencia práctica entre las constantes simbólicas creadas con la directiva `#define` y las creadas con la palabra clave `const`? Las diferencias tienen que ver con los apuntadores y el alcance de las variables. Los apuntadores y el alcance de las variables son dos aspectos muy importantes de la programación de C, y son tratados en los Días 9 y 12, "Apuntadores" y "Alcance de las variables", respectivamente.

Veamos ahora un programa que muestra las declaraciones de variables y el uso de constantes literales y simbólicas. El código que se encuentra en el listado 3.2 le pide al usuario que teclee su peso en libras y el año de nacimiento. Luego calcula y despliega el peso del usuario en gramos y la edad que tendrá en el año 2000. Se puede teclear, compilar y ejecutar este programa usando los procedimientos explicados en el Día 1, "Comienzo".



Listado 3.2. Un programa que muestra el uso de variables y constantes.

```
1:   /* Muestra las variables y constantes */
2:   #include <stdio.h>
3:   /* Define una constante para convertir libras a gramos */
4:   #define GRAMS_PER_POUND 454
5:   /* Define una constante para el comienzo del siguiente ciclo */
6:   const int NEXT_CENTURY = 2000;
7:   /* Declara las variables necesarias */
8:   long weight_in_grams, weight_in_pounds;
9    int year_of_birth, age_in_2000;
10:
11:
12:  main()
13:  {
14:      /* Recibe entrada de datos del usuario */
15:
16:      printf("Enter your weight in pounds: ");
17:      scanf("%d", &weight_in_pounds);
18:      printf("Enter your year of birth: ");
19:      scanf("%d", &year_of_birth);
20:
21:      /* Ejecuta conversiones */
22:
23:      weight_in_grams = weight_in_pounds * GRAMS_PER_POUND;
24:      age_in_2000 = NEXT_CENTURY - year_of_birth;
25:
26:      /* Despliega los resultados en la pantalla */
27:
28:      printf("\nYour weight in grams = %ld", weight_in_grams);
29:      printf("\nIn 2000 you will be %d years old",
             age_in_2000);
```

```
30:     return 0;
31: }
```

El listado 3.2 produce la siguiente salida:

Salida

```
Enter your weight in pounds: 175
Enter your year of birth: 1960

Your weight in grams = 13914
In 2000 you will be 40 years old
```

Análisis

El programa declara los dos tipos de constantes simbólicas en las líneas 4 y 6. En la línea 4 es usada una constante para hacer más comprensible al valor 454.

Debido a que usa `GRAMS_PER_POUND`, la línea 23 es entendible. Las líneas 8 y 9 declaran las variables usadas en el programa. Observe el uso de nombres descriptivos, como `weight_in_grams`. Leyendo su nombre se ve para qué se usa esta variable. Las líneas 16 y 18 imprimen mensajes en la pantalla. La función `printf()` se trata a mayor detalle posteriormente. Para permitir que el usuario responda a los mensajes, las líneas 17 y 19 usan otra función de biblioteca, `scanf()`, que se trata posteriormente. `scanf()` obtiene información del teclado. Por ahora, acepte que esto funciona como se muestra en el listado. Posteriormente aprenderá exactamente cómo funciona. Las líneas 23 y 24 calculan el peso del usuario en gramos y la edad que tendrá en el año 2000. Estos y otros enunciados serán tratados a detalle el día de mañana. Para terminar el programa, las líneas 28 y 29 despliegan el resultado.

DEBE	NO DEBE
DEBE Usar constantes para hacer que los programas sean más fáciles de leer.	
	NO DEBE Tratar de asignar un valor a una constante después de que ha sido iniciada.

Resumen

En este capítulo se han explorado las variables numéricas que son usadas por un programa en C para guardar datos durante la ejecución del programa. Se ha visto que hay dos amplias clases de variables numéricas, enteras y de punto flotante. Dentro de cada clase hay tipos específicos de variables. El tipo de variable, `int`, `long`, `float` o `double`, que se use para una aplicación específica, depende de la naturaleza de los datos que serán guardados en la variable. También se vio que en un programa C se debe declarar a una variable antes de que pueda ser usada. Una declaración de variable le informa al compilador sobre el nombre y el tipo de la variable.

Este capítulo también ha tratado dos tipos de constantes del C, literales y simbólicas. A diferencia de las variables, el valor de una constante no puede ser cambiado durante la ejecución del programa. Se teclean constantes literales en el código fuente cada vez que se necesite el valor. Las constantes simbólicas tienen un nombre asignado a ellas, que es usado cada vez que se necesita el valor de la constante. Las constantes simbólicas pueden ser creadas con la directiva `#define` o con la palabra clave `const`.

Preguntas y respuestas

1. Las variables `long int` guardan números más grandes; entonces, ¿por qué no usarlas siempre en vez de las variables `int`?

   Una variable `long int` ocupa más RAM que la `int`, que es más pequeña. En programas pequeños esto no da problema. Sin embargo, conforme los programas se hacen más grandes, hay que tratar de ser eficiente en el uso de memoria.

2. ¿Qué pasa si asigno un número con un decimal a un entero?

   Se puede asignar un número con un decimal a una variable `int`. Si se está usando una variable constante el compilador probablemente le dará un aviso de precaución. El valor asignado tendrá truncada la porción decimal. Por ejemplo, si se asigna 3.14 a una variable entera llamada pi, pi contendrá solamente 3. El .14 será truncado y desechado.

3. ¿Qué pasa si pongo un número en un tipo que no es lo suficientemente grande como para guardarlo?

   Muchos compiladores permitirán esto sin indicar un error. Sin embargo, el número es acomodado para que quepa y es incorrecto. Por ejemplo, si se asigna 32768 a un entero con signo de dos bytes, el entero en realidad contiene el valor -32768. Si se asigna el valor 65535 a este entero, en realidad contiene el valor -1. El restar el valor máximo que el campo pueda contener, por lo general le da el valor que será almacenado.

4. ¿Qué pasa si pongo un número negativo en una variable sin signo?

   Como se indicó en el ejemplo anterior, el compilador tal vez no marque ningún error si se hace esto. El compilador hace el mismo ajuste que cuando se asigna un número que es demasiado grande. Por ejemplo, si se asigna -1 a una variable `int` que es de dos bytes de longitud, el compilador pondrá el número mayor posible en la variable (65535).

5. ¿Cuál es la diferencia práctica entre las constantes simbólicas creadas con la directiva `#define` y las creadas con la palabra clave `const`?

La diferencia tiene que ver con los apuntadores y el alcance de la variable. Los apuntadores y el alcance de la variable son dos aspectos muy importantes de la programación en C, y son tratados en los Días 9 y 12, "Apuntadores" y "Alcance de las variables", respectivamente. Por el momento, basta saber que usando `#define` para crear constantes se logra que los programas sean más fáciles de leer.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado y ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Cuál es la diferencia entre una variable entera y una de punto flotante?
2. Dé dos razones para usar una variable de punto flotante de doble precisión (tipo `double`) en vez de una variable de punto flotante de precisión sencilla (tipo `float`).
3. ¿Cuáles son las cinco reglas que indica el estándar ANSI que siempre serán ciertas cuando se ubica espacio para las variables?
4. ¿Cuáles son las dos ventajas de usar una constante simbólica en vez de una literal?
5. Muestre dos métodos para definir una constante simbólica llamada `MAXIMUM` y que tenga un valor de 100.
6. ¿Qué caracteres son permitidos en los nombres de variables del C?
7. ¿Qué reglas hay que seguir para la creación de nombres para variables y constantes?
8. ¿Cuál es la diferencia entre una constante simbólica y una literal?
9. ¿Cuál es el valor mínimo que puede contener una variable de tipo `int`?

Ejercicios

1. ¿Qué tipo de variable sería más adecuado para guardar los siguientes valores?
   a. La edad de una persona redondeada a años.
   b. El peso de una persona en libras.
   c. El radio de un círculo.
   d. Su salario anual.

e. El costo de una cosa.

f. La calificación máxima de un examen (suponga que es siempre 100).

g. La temperatura.

h. El valor neto de una persona.

i. La distancia a una estrella, en millas.

2. Determine nombres de variable adecuados para los valores del ejercicio 1.
3. Escriba declaraciones para las variables del ejercicio 2.
4. ¿Cuáles de los siguientes nombres de variable son válidos?

```
a. 123variable
b. x
c. anotación_total
d. Peso_en_#s
e. uno
f. costo-bruto
g. RADIO
h. Radio
i. radio
j. ésta_es_una _variable_para_guardar_el_ancho_de_una_caja
```

DIA 4

Enunciados, expresiones y operadores

Los programas de C consisten en enunciados, y la mayoría de ellos están compuestos de expresiones y operadores. Se necesita comprender estos tres temas para ser capaz de escribir programas en C. Hoy aprenderá

- ❑ Lo que es un enunciado.
- ❑ Lo que es una expresión.
- ❑ Los operadores matemáticos, relacionales y lógicos del C.
- ❑ Qué es la precedencia de operadores.
- ❑ El enunciado `if`.

Enunciados

Un *enunciado* es una indicación completa que le da instrucciones a la computadora para ejecutar alguna tarea. En C, los enunciados son escritos, por lo general, uno en cada línea, aunque algunos enunciados pueden extenderse a varias líneas. Los enunciados del C siempre terminan con un punto y coma (a excepción de las directivas del preprocesador, como `#define` y `#include`, que se tratan en el Día 21, "Aprovechando las directivas del preprocesador y más"). Ya le han sido presentados varios tipos de enunciados del C. Por ejemplo,

```
x = 2 + 3;
```

es un *enunciado de asignación.* Este le da instrucciones a la computadora para que sume 2 y 3 y asigne el resultado a la variable `x`. Otros tipos de enunciados son presentados conforme se les necesita a lo largo de este libro.

Enunciados y el espacio en blanco

El término *espacio en blanco* se refiere a los espacios en blanco, tabuladores y líneas en blanco que se encuentran en el código fuente. El compilador C no es sensible al espacio en blanco. Cuando el compilador está leyendo un enunciado en el código fuente, toma en cuenta los caracteres en el enunciado y el punto y coma terminal, pero ignora los espacios en blanco. Por lo tanto, el enunciado `x=2+3;` es exactamente equivalente a

```
x = 2 + 3;
```

y también es equivalente a

```
   x                    =
2
+
            3;
```

Esto le da una gran flexibilidad en el formateo del código fuente. Sin embargo, no se debe usar formateo como en el ejemplo anterior. Los enunciados deben ser dados uno por renglón, con un número de espacios alrededor de las variables y operadores. Si se siguen las convenciones de formateo usadas en este libro, se tendrá una buena forma. Conforme tenga más experiencia descubrirá que prefiere ligeras variaciones. Lo que hay que lograr es que el código fuente sea legible.

Sin embargo, la regla de que al C no le importan los espacios en blanco tiene una excepción. Dentro de las constantes literales de cadena los tabuladores y espacios no son ignorados, sino que son considerados parte de la cadena. Una *cadena* es un conjunto de caracteres. Las constantes *literales de cadena* son cadenas que se encuentran entre comillas, y son interpretadas literalmente por el compilador, espacio por espacio. Aunque lo siguiente está muy mal, no obstante es legal:

```
printf(
"Hello, world!"
);
```

Sin embargo, lo que viene a continuación no es legal:

```
printf("Hello,
world!");
```

Para partir una línea de una constante literal de cadena se debe usar el carácter de diagonal inversa (\) inmediatamente antes del corte. Así, lo siguiente es legal:

```
printf("Hello,\
 world");
```

Si se pone un punto y coma solo en una línea, se crea un *enunciado nulo*, esto es, un enunciado que no ejecuta ninguna acción. Esto es perfectamente legal en C. Posteriormente en el libro aprenderá la manera en que el enunciado nulo puede ser útil algunas veces.

Enunciados compuestos

Un *enunciado compuesto,* también llamado un *bloque,* es un grupo de dos o más enunciados de C encerrados entre llaves. A continuación se presenta un ejemplo de bloque:

```
{
   printf("Hello, ");
   printf("world!");
}
```

En C, un bloque puede usarse en cualquier lugar donde puede usarse un solo enunciado. Muchos ejemplos de esto aparecen a lo largo del libro. Tome en cuenta que las llaves pueden ser posicionadas en diversas maneras. Lo siguiente es equivalente al ejemplo anterior:

```
{printf("Hello, ");
printf("world!");}
```

Es una buena idea el poner las llaves en su propia línea, haciendo claramente visible el inicio y final del bloque. El hecho de poner las llaves en sus propias líneas también facilita ver dónde faltan.

DEBE	NO DEBE

DEBE Mantener consistencia en la forma en que usa los espacios en blanco en los enunciados.

DEBE Poner llaves de bloque en sus propias líneas. Esto facilita la lectura del código.

DEBE Alinear las llaves del bloque para que sea fácil encontrar el inicio y el fin de un bloque.

NO DEBE Separar un solo enunciado en varios renglones si no hay necesidad. Trate de mantener los enunciados en una línea.

Expresiones

En C, una *expresión* es cualquier cosa que evalúa a un valor numérico. Las expresiones de C se presentan en todos los niveles de complejidad.

Expresiones simples

La expresión más simple de C consiste en un solo concepto: una simple variable, constante literal o constante simbólica. A continuación se presentan cuatro expresiones:

```
PI              /* Una constante simbólica (definida en el programa) */
20              /* Una constante literal. */
velocidad /* Una variable. */
-1.25           /* Otra constante literal. */
```

Una constante literal evalúa a su propio valor. Una constante simbólica evalúa al valor que le fue dado cuando se creó con la directiva `#define`. Una variable evalúa al valor asignado a ella por el programa.

Expresiones complejas

Las *expresiones más complejas* consisten en expresiones simples conectadas por operadores. Por ejemplo,

```
2 + 8
```

es una expresión que consiste de dos subexpresiones 2 y 8 y el operador de suma +. La expresión 2 + 8 evalúa como usted sabe a 10. Se pueden escribir expresiones de C de mayor complejidad:

```
1.25 / 8 + 5 * tasa + tasa * tasa / costo
```

Cuando una expresión contiene varios operadores la evaluación de la expresión depende de la precedencia de los operadores. Este concepto, así como sus detalles acerca de todos los operadores del C, se tratan posteriormente en este capítulo.

Las expresiones de C se ponen todavía más interesantes. Vea el siguiente enunciado de asignación:

```
x = a + 10;
```

Este enunciado evalúa la expresión `a + 10` y asigna el resultado a `x`. Además, el enunciado completo `x = a + 10` es en sí una expresión que evalúa al valor de la variable que se encuentra al lado izquierdo del signo de igual. Esto se muestra en la figura 4.1.

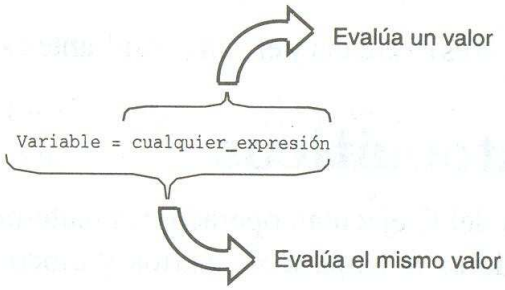


Figura 4.1. *Un enunciado de asignación es en sí mismo una expresión.*

Por lo tanto, se pueden escribir enunciados como

```
y = x = a + 10;
```

el cual asigna el valor de la expresión `a + 10` a ambas variables, `x` y `y`. También se pueden escribir enunciados como

```
x = 6 + (y = 4 + 5);
```

El resultado de este enunciado es que `y` tiene el valor de 9 y `x` tiene el valor de 15. Observe los paréntesis, que son necesarios para que el enunciado pueda ser compilado. El uso de los paréntesis se trata posteriormente, en este capítulo.

Operadores

Un *operador* es un símbolo que le da instrucciones al C para que ejecute alguna operación, o acción, en uno o más operandos. Un *operando* es algo sobre lo cual actúa un operador. En C todos los operandos son expresiones. Los operadores de C se agrupan en varias categorías.

El operador de asignación

El operador de asignación es el signo de igual (`=`). Es usado en programación en una forma ligeramente diferente a su uso en las matemáticas normales. Si se escribe

```
x = y;
```

en un programa C no significa "x es igual a y". En cambio, significa "asigne el valor de y a x". En un enunciado de asignación del C el lado derecho puede ser cualquier expresión, y el lado izquierdo debe ser un nombre de variable. Por lo tanto, la forma es

```
variable = expresión;
```

Cuando se ejecuta, la *expresión* es evaluada y el valor resultante es asignado a la `variable`.

Operadores matemáticos

Los operadores matemáticos del C ejecutan operaciones matemáticas, como la suma y la resta. El C tiene dos operadores matemáticos unarios y cinco operadores matemáticos binarios.

Los operadores matemáticos unarios

Los operadores matemáticos *unarios* son llamados de esta forma debido a que toman un solo operando. El C tiene dos operadores matemáticos unarios, que se listan en la tabla 4.1.

Tabla 4.1. Operadores matemáticos unarios del C.

Operador	Símbolo	Acción	Ejemplo
Incremento	++	Incrementa al operando en 1	++x, x++
Decremento	—	Decrementa al operando en 1	—x, x—

Los operadores de incremento y decremento pueden usarse solamente con variables y no con constantes. La operación ejecutada es el sumar o restar uno del operando. En otras palabras, los enunciados

```
++x;
--y;
```

son equivalentes a

```
x = x + 1;
y = y - 1;
```

Debe observar en la tabla 4.1 que cualquiera de los operadores unarios puede ser puesto antes de su operando (en *modo de prefijo)* o después de su operando (en *modo de posfijo).* Estos dos modos no son equivalentes. Se diferencian en el momento en que se ejecuta el incremento o decremento:

- Cuando se usan en modo de prefijo, los operadores de incremento y decremento modifican a su operando antes de que sea usado.
- Cuando se usan en modo de posfijo los operadores de incremento y decremento modifican a su operando después de que es usado.

Un ejemplo hará esto más claro. Vea los siguientes dos enunciados:

```
x = 10;
y = x++;
```

Después de que estos enunciados se ejecutan `x` tiene el valor de 11 `y` y tiene el valor de 10: el valor de `x` fue asignado a `y`, y luego `x` fue incrementado. Por lo contrario, los enunciados

```
x = 10;
y = ++x;
```

dan como resultado que tanto `y` como `x` tienen el valor de 11: `x` fue incrementado y luego fue asignado su valor a `y`.

Recuerde que `=` es el operador de asignación y no el enunciado de igualdad. Como una analogía piense que el `=` es el operador de "fotocopia". El enunciado `x = y` significa que se copie `x` a `y`. Los cambios subsecuentes de `x`, después de que la copia ha sido hecha, no tienen efecto en `y`.

El programa del listado 4.1 muestra la diferencia entre los modos de prefijo y posfijo.

Captura

Listado 4.1. UNARY.C.

```
1:  /* Demuestra los modos de prefijo y posfijo de operadores unarios */
2:
3:  #include <stdio.h>
4:
5:  int a, b;
6:
7:  main()
8:  {
```

Listado 4.1. continuación

```
9:      /* Pone a y b igual a 5 */
10:
11:     a = b = 5;
12:
13:     /* Los imprime decrementándolos cada vez */
14:      /* Usa modo de prefijo para b y modo de posfijo para a */
15:
16:     printf("\n%d   %d", a–, –b);
17:     printf("\n%d   %d", a–, –b);
18:     printf("\n%d   %d", a–, –b);
19:     printf("\n%d   %d", a–, –b);
20:     printf("\n%d   %d", a–, –b);
21:
22:     return 0;
23: }
```

Salida

La salida del programa es

```
5    4
4    3
3    2
2    1
1    0
```

Análisis

Este programa declara dos variables, `a` y `b`, en la línea 5. En la línea 11 las variables son puestas al valor de 5. Con la ejecución de cada enunciado `printf()` (líneas 16-20) tanto `a` como `b` son decrementados en 1. Después de que es impreso, `a` es decrementado. `b` es decrementado antes de ser impreso.

Los operadores matemáticos binarios

Los *operadores binarios* del C usan dos operandos. Los operadores binarios, que incluyen las operaciones matemáticas comunes que se encuentran en una calculadora, son listados en la tabla 4.2.

Los primeros cuatro operadores de la tabla 4.2 le deben ser familiares y deberá tener poco problema para usarlos. El quinto operador, módulo, tal vez sea nuevo. El *módulo* regresa el residuo cuando el primer operando es dividido entre el segundo operando. Por ejemplo, 11 módulo 4 es igual a 3 (esto es, 4 cabe dos veces en 11 y sobran 3). A continuación se presentan más ejemplos:

```
100 módulo 9 es igual a 1
10  módulo 5 es igual a 0
40  módulo 6 es igual a 4
```

Tabla 4.2. Operadores matemáticos binarios del C.

Operador	Símbolo	Acción	Ejemplo
Suma	+	Suma sus dos operandos	x + y
Resta	-	Resta el segundo operando del primero	x - y
Multiplicación	*	Multiplica sus dos operandos	x * y
División	/	Divide el primer operando entre el segundo	x / y
Módulo	%	Da el residuo cuando el primer operando es dividido entre el segundo	x % y

El programa que se encuentra en el listado 4.2 muestra la manera en que puede usarse el operador de módulo, para convertir un gran número de segundos en horas, minutos y segundos.

Captura

Listado 4.2. SECONDS.C.

```
1:  /* Ilustra el operador de módulo */
2:  /* Recibe un número de segundos y lo convierte a horas, */
3:  /* minutos y segundos */
4:
5:  #include <stdio.h>
6:
7:  /* Define constantes */
8:
9:  #define SECS_PER_MIN 60
10: #define SECS_PER_HOUR 3600
11:
12: unsigned seconds, minutes, hours, secs_left, mins_left;
13:
14: main()
15: {
16:     /* Recibe el número de segundos */
17:
18:     printf("Enter number of seconds (< 65000): ");
19:     scanf("%d", &seconds);
20:
21:     hours = seconds / SECS_PER_HOUR;
22:     minutes = seconds / SECS_PER_MIN;
23:     mins_left = minutes % SECS_PER_MIN;
```

Listado 4.2. continuación

```
24:     secs_left = seconds % SECS_PER_MIN;
25:
26:     printf("%u seconds is equal to ", seconds);
27:     printf("%u h, %u m, and %u s", hours, mins_left, secs_left);
28:
29:     return 0;
30: }
```

Salida

```
E:\>list0402
Enter number of seconds (< 65000): 60
60 seconds is equal to 0 h, 1 m, and 0 s

E:\>list0402
Enter number of seconds (< 65000): 10000
10000 seconds is equal to 2 h, 46 m, and 40 s
```

Análisis

El programa SECONDS.C sigue el mismo formato que han seguido los programas anteriores. Las líneas 1-3 proporcionan algunos comentarios para indicar lo que va a hacer el programa. La línea 4 es espacio en blanco para hacer más legible al programa. De manera similar al espacio en blanco que se encuentra en enunciados y expresiones, las líneas en blanco son ignoradas por el compilador. La línea 5 incluye los archivos de encabezado necesarios para este programa. Las líneas 9 y 10 definen dos constantes, `SECS_PER_MIN` y `SECS_PER_HOUR`, que se usan para facilitar la lectura de los enunciados en el programa. La línea 12 declara todas las variables que serán usadas. A algunas personas les gusta declarar cada variable en su propia línea, en vez de declararlas todas en una sola, como se muestra anteriormente. De manera similar a muchos elementos del C, esto es, cuestión de estilo. Cualquier método es correcto.

La línea 14 es la función `main()`, que contiene la parte medular del programa. Para convertir segundos a horas y minutos, el programa primero debe obtener los valores que necesita para trabajar. Para hacer esto, la línea 18 usa la función `printf()` para desplegar un enunciado en la pantalla, seguido de la línea 19 que usa la función `scanf()` para obtener el número tecleado por el usuario. El enunciado `scanf()` luego guarda la cantidad de segundos que ha de convertirse en la variable seconds. Las funciones `printf()` y `scanf()` se tratan a mayor detalle en el Día 7, "Entrada/salida básica". La línea 21 contiene una expresión para determinar la cantidad de horas, dividiendo la cantidad de segundos por la constante `SECS_PER_HOUR`. Debido a que hours es una variable entera, la parte fraccional es ignorada. La línea 22 usa la misma lógica para determinar la cantidad total de minutos a que corresponden los segundos tecleados. Debido a que el número total de minutos calculado en la línea 22 también contiene los minutos de las horas, la línea 23 usa el operador de módulo para dividir las horas y guardar los minutos restantes. La línea 24 hace un cálculo similar para determinar la cantidad de segundos que quedan. Las líneas 26 y 27 son un reflejo de lo que se ha visto anteriormente. Ellas toman los valores que han sido calculados en las expresiones y los despliegan. La línea 29 termina el programa, regresando 0 al sistema operativo antes de terminar.

Precedencia de operadores y los paréntesis

En una expresión que contiene más de un operador, ¿cuál es el orden en que se ejecutan las operaciones? La importancia de esta pregunta se ilustra con el siguiente enunciado de asignación:

```
x = 4 + 5 * 3;
```

Si primero se ejecuta la suma, se tiene

```
x = 9 * 3;
```

y a x se le asigna el valor 27. Por el contrario, si primero se ejecuta la multiplicación, se tiene

```
x = 4 + 15;
```

y a x se le asigna el valor 19. Es obvio que se necesitan algunas reglas acerca del orden en que se ejecutan las operaciones. Este orden, llamado *precedencia de los operadores,* es indicado estrictamente en C. Cada operador tiene una precedencia específica. Cuando una expresión es evaluada, los operadores que tienen mayor precedencia se ejecutan primero. La precedencia de los operadores matemáticos del C se lista en la tabla 4.3. El número 1 es la mayor precedencia.

Tabla 4.3. La precedencia de los operadores matemáticos del C.

Operadores	Precedencia relativa
++ --	1
* / %	2
+ -	3

En la tabla 4.3 se puede ver que en cualquier expresión del C las operaciones se ejecutan en el siguiente orden:

- Incremento y decremento unario.
- Multiplicación, división y módulo.
- Suma y resta.

Si una expresión contiene más de un operador con el mismo nivel de precedencia, los operadores se ejecutan en orden de izquierda a derecha, como aparecen en la expresión. Por ejemplo, en la expresión

```
12 % 5 * 2
```

los operadores % y * tienen el mismo nivel de precedencia, pero el % es el operador de la izquierda y por esto se ejecuta primero. La expresión evalúa a 4 (12 % 5 evalúa a 2; 2 * 2 da 4).

Regresando al ejemplo anterior, se ve que el enunciado `x = 4 + 5 * 3;` asigna el valor de 19 a `x`, debido a que la multiplicación se ejecuta antes que la suma.

¿Qué pasa si el orden de precedencia no evalúa la expresión como se necesita? Usando el ejemplo anterior, ¿qué habría que hacer si se quiere sumar 4 a 5 y luego multiplicar la suma por 3? En el C se usan paréntesis para modificar el orden de evaluación. Una subexpresión encerrada entre paréntesis es evaluada primero, sin tomar en cuenta la precedencia de los operadores. Por lo tanto, se podría escribir

```
x = (4 + 5) * 3;
```

La expresión `4 + 5` dentro del paréntesis es evaluada primero y, por lo tanto, el valor asignado a `x` es 27.

Se pueden usar varios paréntesis y anidarlos en una expresión. Cuando los paréntesis están anidados la evaluación se ejecuta desde la expresión más interna hacia afuera. Vea la siguiente expresión compleja:

```
x = 25 - (2 * (10 + (8 / 2)))
```

Esta evaluación se ejecuta en los siguientes pasos:

1. La expresión más interna, 8 / 2, es evaluada primero dando el valor 4.

   ```
   25 - (2 * (10 + 4))
   ```

2. Moviéndonos hacia afuera, la siguiente expresión, que ahora es 10 + 4, es evaluada, dando como resultado el valor 14.

   ```
   25 - (2 * 14)
   ```

3. La última expresión, o más externa, es 2 * 14, y es evaluada dando como resultado el valor 28.

   ```
   25 - 28
   ```

4. La expresión final, 25 - 28, es evaluada, asignando el valor -3 a la variable x.

   ```
   x = -3
   ```

Tal vez usted quiera usar paréntesis en algunas expresiones con objeto de tener más claridad, incluso cuando no sean necesarios para modificar la precedencia de los operadores. Los paréntesis deben estar siempre en pares, o en caso contrario el compilador generará un mensaje de error.

Orden para la evaluación de subexpresiones

Como se dijo en la sección anterior, si las expresiones de C contienen más de un operador con el mismo nivel de precedencia son evaluadas de izquierda a derecha. Por ejemplo, en la expresión

```
w * x / y * z
```

`w` primero es multiplicado por `x`, el resultado de la multiplicación es luego dividido entre `y` y el resultado de la división es luego multiplicado por `z`.

Sin embargo, entre los niveles de precedencia no hay garantía de que se siga el orden de izquierda a derecha. Vea esta expresión:

```
w * x / y + z / y
```

Debido a la precedencia, la multiplicación y división son ejecutadas antes que la suma. Sin embargo, el C no especifica si la subexpresión `w * x / y` debe ser evaluada antes o después de `z / y`. Tal vez no le sea claro el porqué de la importancia de esto. Vea otro ejemplo:

```
w * x / ++y + z / y
```

Si la primera subexpresión es evaluada primero, `y` ha sido incrementada cuando es evaluada la segunda expresión. Si la segunda expresión es evaluada primero, `y` no ha sido incrementada y el resultado es diferente. Por lo tanto, se debe evitar este tipo de expresiones indeterminadas en la programación.

El apéndice C, "Precedencia de operadores en C", lista la precedencia de todos los operadores de C.

DEBE	NO DEBE
DEBE Usar paréntesis para hacer claro el orden de evaluación de las expresiones.	
	NO DEBE Sobrecargar una expresión. Por lo general, es más claro partir una expresión en dos o más enunciados. Esto es especialmente cierto cuando se usan los operadores unarios (++) o (--).

Operadores relacionales

Los *operadores relacionales* del C se usan para comparar expresiones, "haciendo preguntas" como "¿es x mayor que 100?" o "¿es y igual a 0?". Una expresión que contiene un operador relacional evalúa a cierto (1) o falso (0). Los seis operadores relacionales del C se encuentran listados en la tabla 4.4.

Véase la tabla 4.5 para algunos ejemplos sobre la manera en que pueden usarse los operadores relacionales. Estos ejemplos usan constantes literales, pero los mismos principios se aplican con las variables.

Tabla 4.4. Operadores relacionales del C.

Operador	Símbolo	Pregunta	Ejemplo
Igual	==	¿Es el operando 1 igual al operando 2?	x == y
Mayor que	>	¿Es el operando 1 mayor que el operando 2?	x > y
Menor que	<	¿Es el operando 1 menor que el operando 2?	x < y
Mayor que o igual a	>=	¿Es el operando 1 mayor que o igual al operando 2?	x >= y
Menor que o igual a	<=	¿Es el operando 1 menor que o igual al operando 2?	x <= y
Diferente	!=	¿Es el operando 1 diferente al operando 2?	x != y

Tabla 4.5. Operadores relacionales en uso.

Expresión	Evalúa a
5 == 1	0 (falso)
5 > 1	1 (cierto)
5 != 1	1 (cierto)
(5 + 10) == (3 * 5)	1 (cierto)

DEBE

DEBE Aprender la manera en que el C interpreta al cierto y falso. Cuando trabaje con operadores relacionales, cierto es igual a 1 y falso es igual a 0.

NO DEBE

NO DEBE Confundir el operador relacional == con el operador de asignación =. ¡Este es uno de los errores más comunes que cometen los programadores de C!

El enunciado *if*

Los operadores relacionales se emplean principalmente para construir las expresiones relacionales que se usan en los enunciados `if` y `while`, tratados a detalle en el Día 6, "Control básico del programa". Por ahora es útil explicar lo básico del enunciado `if`, para mostrar la manera en que se usan los operadores relacionales para hacer enunciados de control de programa.

Tal vez se pregunte qué cosa es un enunciado de control de programa. Los enunciados en un programa en C normalmente ejecutan de arriba hacia abajo, en el mismo orden en que aparecen en el archivo de código fuente. Un *enunciado de control de programa* modifica el orden de ejecución de los enunciados. Los enunciados de control de programa pueden usar otros enunciados de programa para ejecutarlos varias veces o para no ejecutarlos, dependiendo de las circunstancias. El enunciado `if` es uno de los enunciados de control de programa del C. Otros, como `do` y `while` se tratan en el Día 6, "Control básico del programa".

En su forma básica, el enunciado `if` evalúa una expresión, y dirige la ejecución del programa dependiendo del resultado de esa evaluación. La forma de un enunciado `if` es la siguiente:

```
if (expresión)
enunciado;
```

Si la *expresión* evalúa a cierto, se ejecuta el *enunciado*. Si la *expresión* evalúa a falso, el *enunciado* no se ejecuta. En cualquier caso, la ejecución continúa al código que se encuentra a continuación del enunciado `if`. Se puede decir que la ejecución del *enunciado* depende del resultado de la *expresión*. Observe que se considera que tanto la línea de `if` (*expresión*) y la línea de *enunciado,* forman el enunciado `if` completo; no son enunciados separados.

DEBE	NO DEBE

DEBE Recordar que si programa demasiado en un día se enfermará de C.

NO DEBE Cometer el error de poner un punto y coma al final de un enunciado `if`. Un enunciado `if` debe terminar con el enunciado condicional que se encuentra a continuación. En el siguiente ejemplo, *enunciado1* ejecuta sin importar si x es igual a 2 o no, debido a que cada línea es evaluada como un enunciado separado ¡y no juntas como se pretende!

```
if( x == 2);        /* aquí no debe ir el punto y coma */
enunciado1;
```

Un enunciado `if` puede controlar la ejecución de varios enunciados mediante el uso de un enunciado compuesto o bloque. Como se definió anteriormente en este capítulo, un bloque

es un grupo de dos o más enunciados encerrados entre llaves. Un bloque puede usarse en cualquier lugar donde puede usarse un solo enunciado. Por lo tanto, se podría escribir un enunciado `if` de la manera siguiente:

```
if (expresión)
{
    enunciado1;
    enunciado2;
    /* aquí va código adicional */
    enunciadon;
}
```

En la programación encontrará que los enunciados `if` se usan la mayoría de las veces con expresiones relacionales. En otras palabras, "ejecute los siguientes enunciados sólo si tales y cuales condiciones son ciertas". A continuación se presenta un ejemplo:

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

Este código asigna el valor de `x` a `y` solamente si `x` es mayor que `y`. Si `x` no es mayor que `y` no se ejecuta ninguna asignación. El listado 4.3 presenta un programa corto que ilustra el uso de enunciados `if`.

Captura

Listado 4.3. IF.C.

```
1:  /* Demuestra el uso de enunciados if */
2:
3:  #include <stdio.h>
4:
5:  int x, y;
6:
7:  main()
8:  {
9:      /* Recibe los dos valores que se han de probar */
10:
11:     printf("\nInput an integer value for x: ");
12:     scanf("%d", &x);
13:     printf("\nInput an integer value for y: ");
14:     scanf("%d", &y);
15:
16:     /* Prueba los valores e imprime el resultado */
17:
18:     if (x == y)
19:         printf("x is equal to y");
20:
21:     if (x > y)
22:         printf("x is greater than y");
23:
24:     if (x < y)
25:         printf("x is smaller than y");
```

```
26:
27:     return 0;
28: }
```

Salida

```
E:\>list0403
Input an integer value for x: 100
Input an integer value for y: 10
x is greater than y

E:\>list0403
Input an integer value for x: 10
Input an integer value for y: 100
x is smaller than y

E:\>list0403
Input an integer value for x: 10
Input an integer value for y: 10
x is equal to y
```

Análisis

IF.C muestra tres enunciados `if` en acción (líneas 18-25). Muchas de las líneas de este programa le deben ser familiares. La línea 5 declara dos variables, `x` y `y`, y las líneas 10-14 le piden al usuario los valores que deberán ser puestos en estas variables. Las líneas 18-25 usan enunciados `if` para determinar si `x` es mayor que, menor que o igual a `y`. Observe que la línea 18 usa un enunciado `if` para ver si `x` es igual a `y`. Recuerde que `==`, el operador de igualdad, es lo mismo que decir "es igual a", y no debe ser confundido con el operador de asignación `=`. Después de que el programa revisa para ver si las variables son iguales, en la línea 21 revisa para ver si `x` es mayor que `y`, seguido de una revisión en la línea 24 para ver si `x` es menor que `y`. Tal vez piense que esto es ineficiente y tiene usted razón. En el siguiente programa verá cómo evitar esta ineficiencia. Por ahora ejecute el programa con diferentes valores para `x` y `y` y vea los resultados.

Un enunciado `if` puede incluir una cláusula `else`. La cláusula `else` se incluye de la siguiente manera:

```
if (expresión)
    enunciado1;
else
    enunciado2;
```

Si la *expresión* es cierta, se ejecuta el *enunciado1*. Si es falsa, se ejecuta el *enunciado2*. Tanto el *enunciado1* como el *enunciado2* pueden ser enunciados compuestos, o bloques. El listado 4.4 muestra al programa del listado 4.3 reescrito para usar un enunciado `if` con una cláusula `else`.

Captura

Listado 4.4. El enunciado `if` con una cláusula `else`.

```
1:  /* Demuestra el uso del enunciado if con cláusula else */
2:
3:  #include <stdio.h>
4:
5:  int x, y;
6:
7:  main()
8:  {
9:      /* Recibe los dos valores que se han de probar */
10:
11:     printf("\nInput an integer value for x: ");
12:     scanf("%d", &x);
13:     printf("\nInput an integer value for y: ");
14:     scanf("%d", &y);
15:
16:     /* Prueba los valores e imprime el resultado */
17:
18:     if (x == y)
19:         printf("x is equal to y");
20:     else
21:         if (x > y)
22:             printf("x is greater than y");
23:         else
24:             printf("x is smaller than y");
25:
26:     return 0;
27: }
```

Salida

```
E:\>list0404
Input an integer value for x: 99

    Input an integer value for y: 8
    x is greater than y

    E:\>list0404

    Input an integer value for x: 8

    Input an integer value for y: 99
    x is smaller than y

    E:\>list0404

    Input an integer value for x: 99

    Input an integer value for y: 99
    x is equal to y
```

Análisis Las líneas 18-24 son ligeramente diferentes del listado anterior. La línea 18 todavía revisa para ver si `x` es igual a `y`. Si `x` sí es igual a `y` se escribe que `x` es igual a `y`, de manera similar a como se hizo en IF.C. Sin embargo, el programa termina a continuación. Las líneas 20-24 no se ejecutan. La línea 21 se ejecuta solamente en el caso de que `x` no sea igual a `y` o, para decirlo con más precisión, si la expresión "x igual a y" es falsa. Si `x` no es igual a `y`, la línea 21 revisa para ver si `x` es mayor que `y`. Si esto es así, la línea 22 imprime que `x` es mayor que `y` y, en caso contrario `(else)`, ejecuta la línea 24.

Observe que el programa del listado 4.4 usa enunciados `if` *anidados*. El anidado significa poner (anidar) uno o más enunciados de C dentro de otro enunciado de C. En el caso del listado 4.4, un enunciado `if` es parte de la cláusula `else` del primer enunciado `if`.

Sintaxis

El enunciado *if*

Forma 1

```
if( expresión )
    enunciado1
siguiente enunciado
```

Este es el enunciado `if` en su forma más simple. Si la expresión es cierta, entonces se ejecuta el `enunciado1`. Si la `expresión` no es cierta, el `enunciado1` es ignorado.

Forma 2

```
if( expresión )
    enunciado1
else
    enunciado2
siguiente enunciado
```

Esta es la forma más común del enunciado `if`. Si la primera `expresión` es cierta, se ejecuta el `enunciado1` y, en caso contrario, se ejecuta el `enunciado2`.

Forma 3

```
if( expresión )
    enunciado1
else if( expresión )
        enunciado2
     else
        enunciado3
siguiente enunciado
```

Esta forma presenta un `if` anidado. Si la primera *`expresión`* es cierta, se ejecuta el *`enunciado1`* y, en caso contrario, se evalúa la segunda *`expresión`*. Si la primera *`expresión`* no es cierta y la segunda es cierta, se ejecuta el *`enunciado2`*. Si ambas expresiones son falsas se ejecuta el *`enunciado3`*. Solamente uno de los tres enunciados se ejecuta.

Ejemplo 1

```
if( salario > 45,000 )
    impuesto = 0.30
else
    impuesto = 0.25
```

Ejemplo 2

```
if( edad < 18 )
    printf("Menor");
else if( edad < 65 )
          printf("Adulto");
     else          printf( "Anciano");
```

Evaluación de expresiones relacionales

Recuerde que las expresiones que usan operadores relacionales son expresiones del C verdaderas, que evalúan por definición a un valor. Las expresiones relacionales evalúan a un valor que puede ser falso (0) o cierto (1). Aunque el uso más común para las expresiones relacionales se da dentro de los enunciados `if` y otras construcciones condicionales, pueden usarse como valores numéricos puros. Esto es ilustrado por el programa que se encuentra en el listado 4.5.

Captura

Listado 4.5. Demostración de la evaluación de expresiones relacionales.

```
1:  /* Demuestra la evaluación de expresiones relacionales */
2:
3:  #include <stdio.h>
4:
5:  int a;
6:
7:  main()
8:  {
9:       a = (5 == 5);           /* Evalúa a 1 */
10:     printf("\na = (5 == 5)\na = %d", a);
11:
12:      a = (5 != 5);           /* Evalúa a 0 */
13:     printf("\na = (5 != 5)\na = %d", a);
14:
15:     a = (12 == 12) + (5 != 1); /* Evalúa a 1 + 1 */
16:     printf("\na = (12 == 12) + (5 != 1)\na = %d", a);
17:     return 0;
18: }
```

Salida

A continuación se presenta la salida del listado 4.5:

```
a = (5 == 5)
a = 1
a = (5 != 5)
a = 0
a = (12 == 12) + (5 != 1)
a = 2
```

Análisis

La salida de este listado puede parecer algo confusa a primera vista. Recuerde que el error más común que comete la gente cuando usa los operadores relacionales es usar un signo de igual solo, el operador de asignación, en vez de un signo de igual doble.

La expresión

```
x = 5
```

evalúa a 5 (y también asigna el valor de 5 a `x`). Por el contrario, la expresión

```
x == 5
```

evalúa a 0 o a 1 (dependiendo si `x` es igual a 5) y no cambia el valor de `x`. Si por error se escribe

```
if (x = 5)
    printf("x es igual a 5");
```

el mensaje siempre se imprimirá debido a que la expresión que está siendo probada por el enunciado `if` siempre evalúa a cierto, sin importar cuál haya sido el valor original de `x`.

Con el listado 4.5 se puede comenzar a comprender por qué toma los valores que toma. En la línea 9 el valor 5 es igual a 5 y, por lo tanto, se asigna cierto (1) a `a`. En la línea 12 el enunciado "5 no es igual a 5" es falso y por lo tanto se asigna 0 a `a`.

Repitiendo, los operadores relacionales se usan para crear expresiones relacionales, que hacen preguntas acerca de relaciones entre expresiones. La respuesta regresada por una expresión relacional es 1 (representando cierto) o 0 (representando falso).

Precedencia de los operadores relacionales

De manera similar a los operadores matemáticos, tratados anteriormente en este capítulo, los operadores relacionales tienen una precedencia que determina el orden en el que se ejecutan en una expresión que tiene varios operadores. En forma similar, se pueden usar paréntesis para modificar la precedencia en expresiones que usan operadores relacionales.

En primer lugar, todos los operadores relacionales tienen menor precedencia que los operadores matemáticos. Por lo tanto, si se escribe

```
if (x + 2 > y)
```

2 es sumado a x y el resultado es comparado con y. Esto es el equivalente de

```
if ((x + 2) > y)
```

que es un buen ejemplo sobre el uso de paréntesis para dar claridad. Aunque no son requeridos por el compilador C, los paréntesis que rodean a (x + 2) aclaran que es la suma de x y 2 la que va a ser comparada contra y.

También hay una precedencia de dos niveles dentro de los operadores relacionales. Esta se muestra en la tabla 4.6.

Por lo tanto, si se escribe

```
x == y > z
```

es lo mismo que escribir

```
x == (y > z)
```

debido a que el C evalúa primero la expresión y > z, dando como resultado un valor de 0 o 1. A continuación el C determina si x es igual al 1 o al 0 obtenido en el primer paso. Es muy poco probable que se llegue a dar el caso de que se use este tipo de construcción, pero se debe saber acerca de ella.

Recuerde que, el apéndice C, "Precedencia de operadores en C", lista la precedencia de todos los operadores del C.

DEBE NO DEBE

NO DEBE Poner enunciados de asignación en enunciados if. Esto puede dar lugar a confusión si otras personas observan el código. Pueden pensar que es un error y cambiar la asignación al enunciado de igualdad lógica.

NO DEBE Usar el operador "no igual a" (!=) en un enunciado if que contenga un else. Casi siempre es más claro usar el operador "igual a" (==) con un else.

```
if ( x != 5 )
      enunciado1;
else
      enunciado2;
sería mejor como
if ( x == 5 )
      enunciado2;
else
      enunciado1;
```

Tabla 4.6. Orden de precedencia de los operadores relacionales del C.

Operador	Precedencia relativa
< <= > >=	1
!= ==	2

Operadores lógicos

Algunas veces tal vez necesite hacer más de una pregunta relacional al mismo tiempo. Por ejemplo, "si son las 7:00 AM y es un día laboral y no estoy de vacaciones, haz sonar al despertador". Los operadores lógicos del C le permiten combinar dos o más expresiones relacionales en una sola expresión que evalúa a cierto o falso. Los tres operadores lógicos del C se listan en la tabla 4.7.

Tabla 4.7. Operadores lógicos del C.

Operador	Símbolo	Ejemplo
y	`&&`	`exp1 && exp2`
o	`||`	`exp1 || exp2`
no	`!`	`!exp1`

La manera en que funcionan estos operadores lógicos se explica en la tabla 4.8.

Tabla 4.8. Operadores lógicos del C en uso.

Expresión	Evalúa a
`(exp1 && exp2)`	Cierto (1) solamente si ambos exp1 y exp2 son ciertos. En caso contrario, falso (0).
`(exp1 || exp2)`	Cierto (1) si cualquiera de exp1 y exp2 es cierto. En caso contrario, falso (0).
`(!exp1)`	Falso (0) si exp1 es cierto, y cierto (1) si exp1 es falso.

Puede ver que las expresiones que usan los operadores lógicos evalúan a cierto o falso, dependiendo de los valores cierto o falso de sus operandos. La tabla 4.9 muestra ejemplos de código de trabajo.

Tabla 4.9. Ejemplos de código de operadores lógicos del C.

Expresión	Evalúa a
`(5 == 5) && (6 != 2)`	Cierto (1) debido a que ambos operandos son ciertos.
`(5 > 1) || (6 < 1)`	Cierto (1) debido a que un operando es cierto.
`(2 == 1) && (5 == 5)`	Falso (0) debido a que un operando es falso.
`!(5 == 4)`	Cierto (1) debido a que el operando es falso.

Se pueden crear expresiones que usan varios operadores lógicos. Por ejemplo, para hacer la pregunta "¿es x igual a 2, 3 o 4?" se podría escribir

```
(x == 2) || (x == 3) || (x == 4)
```

Los operadores lógicos a menudo proporcionan más de una forma de hacer una pregunta. Si x es una variable entera, la pregunta anterior también pudiera ser escrita en alguna de las siguientes maneras:

```
(x > 1) && (x < 5)
```

o

```
(x >= 2) && (x <= 4)
```

Más sobre valores cierto/falso

Ya ha visto que las expresiones relacionales del C evalúan a 0 para representar falso y a 1 para representar cierto. Sin embargo, es importante estar consciente de que cualquier valor numérico es interpretado como cierto o falso cuando es usado en una expresión o enunciado del C que está esperando un valor lógico (esto es, cierto o falso). Las reglas para esto son las siguientes:

Un valor de 0 representa falso.

Cualquier valor diferente de 0 representa cierto.

Esto es ilustrado por el siguiente ejemplo:

```
x = 125;
if (x)
      printf("%d", x);
```

En este caso, el valor de x es impreso.

Como x tiene un valor diferente de cero, la expresión (x) es interpretada como cierta por el enunciado if. Se puede generalizar esto todavía más, debido a que cualquier expresión C escrita

(expresión)

es equivalente a escribir

(expresión != 0)

Ambas evalúan a cierto cuando la expresión no es igual a cero, y a falso cuando la expresión es 0. Usando al operador no (!) también se puede escribir:

(!expresión)

que es equivalente a

(expresión == 0)

Precedencia de los operadores lógicos

Tal como usted se imagina, los operadores lógicos también tienen un orden de precedencia, tanto entre ellos como en relación a otros operadores. El operador ! tiene una precedencia igual a los operadores matemáticos unarios ++ y --. Por lo tanto, ! tiene una precedencia mayor que todos los operadores relacionales y todos los operadores matemáticos binarios.

Por el contrario, los operadores && y || tienen una precedencia mucho menor, menor que todos los operadores matemáticos y relacionales, aunque && tiene una mayor precedencia que ||. De manera similar a todos los operadores del C, se pueden utilizar paréntesis para modificar el orden de evaluación cuando se usan los operadores lógicos. Vea el siguiente ejemplo.

Se quiere escribir una expresión lógica que haga tres comparaciones individuales:

1. ¿Es a menor que b?
2. ¿Es a menor que c?
3. ¿Es c menor que d?

Se quiere que la expresión lógica completa evalúe a cierto si la condición 3 es cierta y cualquiera de las condiciones 1 o 2 sea cierta. Podría escribir

```
a < b || a < c && c < d
```

Sin embargo, esto no hace lo que se pretende. Debido a que el operador && tiene mayor precedencia que ||, la expresión es equivalente a

```
a < b || (a < c && c < d)
```

y evalúa a cierto si `(a < b)` es cierto, sin importar que las relaciones `(a < c)` y `(c < d)` sean ciertas. Se necesita escribir

```
(a < b || a < c) && c < d
```

lo que fuerza que el `||` sea evaluado antes que el `&&`. Esto se muestra en el listado 4.6, que evalúa la expresión escrita en ambas formas. Las variables están puestas en tal forma que si se escriben correctamente la expresión debe evaluar a falso (0).

Captura

Listado 4.6. Precedencia de los operadores lógicos.

```
1:  #include <stdio.h>
2:
3:  /* Inicializa variables. Observe que c no es menor que d, */
4:  /* que es una de las condiciones que se han de probar. */
5:  /* Por lo tanto, la expresión completa debe evaluar a falso */
6:
7:  int a = 5, b = 6, c = 5, d = 1;
8:  int x;
9:
10: main()
11: {
12:     /* Evalúa la expresión sin paréntesis */
13:
14:     x = a < b || a < c && c < d;
15:     printf"\nWithout parentheses the expression evaluates as %d", x),
16:
17:     /* Evalúa la expresión con paréntesis */
18:
19:     x = (a < b || a < c) && c < d;
20:     printf("\nWith parentheses the expression evaluates as \
            %d", x);
21:     return 0;
22: }
```

Salida

```
Without parentheses the expression evaluates as 1
With parentheses the expression evaluates as 0
```

Análisis

Teclee y corra este listado. Observe que los dos valores impresos para la expresión son diferentes. Este programa inicializa cuatro variables, en la línea 7, con valores que serán usados en las comparaciones. La línea 8 declara `x` para que sea usada para guardar e imprimir los resultados. Las líneas 14 y 19 usan los operadores lógicos. La línea 14 no usa los paréntesis, por lo que los resultados son determinados por la precedencia de operadores. En este caso los resultados no son los deseados. La línea 19 usa paréntesis para cambiar el orden en que son evaluadas las expresiones.

Operadores de asignación compuestos

Los *operadores de asignación compuestos* del C proporcionan un método abreviado para combinar una operación matemática binaria con una operación de asignación. Por ejemplo, digamos que se quiere incrementar el valor de `x` en 5, o, en otras palabras, sumar 5 a `x` y asignar el resultado a `x`. Se podría escribir

```
x = x + 5;
```

Con un operador de asignación compuesto, del cual se puede pensar como un método abreviado de asignación, se podría escribir:

```
x += 5;
```

En una notación más general, los operadores de asignación compuestos tienen la siguiente sintaxis (donde *op* representa un operador binario):

```
exp1 op= exp2
```

que es equivalente a escribir:

```
exp1 = exp1 op exp2;
```

Se pueden crear operadores de asignación compuestos con los cinco operadores matemáticos binarios tratados anteriormente en este capítulo. La tabla 4.10 lista algunos ejemplos.

Tabla 4.10. Ejemplos de operadores de asignación compuestos.

Si se escribe	Es equivalente a
`x *= y`	`x = x * y`
`y -= z + 1`	`y = y - z + 1`
`a /= b`	`a = a / b`
`x += y / 8`	`x = x + y / 8`
`y %= 3`	`y = y % 3`

Los operadores compuestos proporcionan un método abreviado conveniente, cuyas ventajas son particularmente evidentes cuando la variable del lado izquierdo del operador de asignación es compleja. De manera similar a todos los otros enunciados de asignación, un enunciado de asignación compuesto es una asignación, y evalúa al valor asignado del lado izquierdo. Por lo tanto, ejecutando los enunciados

```
x = 12;
z = x += 2;
```

da como resultado que tanto `x` como `z` tengan el valor de 14.

El operador condicional

El *operador condicional* es el único *operador ternario* del C, significando esto que usa tres operandos. Su sintaxis es

```
exp1 ? exp2 : exp3
```

Si *exp1* evalúa a cierto (esto es, diferente de 0), la expresión completa evalúa al valor de *exp2*. Si *exp1* evalúa a falso (esto es, cero) la expresión completa evalúa al valor de *exp3*. Por ejemplo, el enunciado

```
x = y ? 1 : 100;
```

asigna el valor de 1 a `x` si `y` es cierta, y asigna 100 a `x` si `y` es falsa. De manera similar, para hacer que `z` sea igual al mayor de `x` y `y`, se podría escribir

```
z = (x > y) ? x : y;
```

Tal vez se haya dado cuenta de que el operador condicional funciona de manera parecida a un enunciado `if`. El enunciado anterior también podría ser escrito:

```
if (x > y)
    z = x;
else
    z = y;
```

El operador condicional no puede usarse en todas las situaciones en vez de una construcción `if...else`, pero el operador condicional es más conciso. El operador condicional también puede usarse en lugares donde no se puede usar un enunciado `if`, como el interior de un solo enunciado `printf()`.

El operador coma

La *coma* es frecuentemente usada en C como una simple marca de puntuación, sirviendo para separar declaraciones de variables, argumentos de función, etc. En algunas situaciones la coma actúa como un operador, en vez de ser solamente un separador. Se puede formar una expresión separando dos subexpresiones con una coma. El resultado es el siguiente:

- ❑ Ambas expresiones son evaluadas (primero la expresión de la izquierda).
- ❑ La expresión completa evalúa al valor de la expresión de la derecha.

Por ejemplo, el enunciado

```
x = (a++ , b++);
```

asigna el valor de `b` a `x`, luego incrementa a `a` y luego incrementa a `b`. Debido a que el operador `++` es usado en modo de posfijo, el valor de `b`, antes de ser incrementado, es asignado a `x`. Es necesario usar paréntesis, ya que el operador de coma tiene una precedencia menor, incluso al operador de asignación.

Como le enseñará el siguiente capítulo, el uso más común del operador de coma es en los enunciados `for`.

DEBE	NO DEBE

DEBE Usar (*expresión* == 0) en vez de (!*expresión*). Cuando se compila, estas dos expresiones se evalúan de la misma forma, más sin embargo, la primera es más legible.

DEBE Usar los operadores lógicos && y || en vez de anidar enunciados `if`.

NO DEBE Confundir el operador de asignación (=) con el operador de igualdad (==).

Resumen

Este capítulo ha tratado mucho material. Se ha aprendido lo que es un enunciado de C, que los espacios en blanco no le importan al compilador C y que los enunciados siempre terminan con un punto y coma. También se aprendió que un enunciado compuesto (o bloque), que consiste en dos o más enunciados encerrados entre llaves, puede usarse en cualquier lugar donde puede usarse un solo enunciado.

Muchos enunciados están compuestos de alguna combinación de expresiones y operadores. Recuerde que una expresión es algo que evalúa a un valor numérico. Las expresiones complejas pueden contener muchas expresiones más simples, llamadas subexpresiones.

Los operadores son símbolos del C que le dan instrucciones a la computadora para que ejecute una operación en una o más expresiones. Algunos operadores son unarios, lo que significa que operan en un solo operando. Sin embargo, la mayoría de los operadores del C son binarios, operando en dos operandos. Un operador, el condicional, es ternario. Los operadores del C tienen una jerarquía definida de precedencia, que determina el orden en el cual se ejecutan las operaciones en una expresión que contiene varios operadores.

Los operadores del C tratados en este capítulo se agrupan en tres categorías, indicando que

- Los operadores matemáticos ejecutan operaciones aritméticas sobre sus operandos (por ejemplo, suma).

- Los operadores relacionales ejecutan comparaciones entre sus operandos (por ejemplo, mayor que).
- Los operadores lógicos operan sobre expresiones cierto/falso. Recuerde que el C usa al 0 y al 1 para representar falso y cierto, respectivamente, y que cualquier valor diferente de cero es interpretado como cierto.

También se presentó el enunciado `if` del C, que le permite controlar la ejecución del programa basándose en la evaluación de expresiones relacionales.

Preguntas y respuestas

1. ¿Qué efecto tienen los espacios y las líneas en blanco sobre la ejecución del programa?

   Los espacios en blanco (líneas, espacios, tabuladores) hacen que el listado de código sea más legible. Cuando el programa es compilado, los espacios en blanco son quitados, y por lo tanto no tienen efecto sobre el programa ejecutado. Por esta razón los espacios en blanco deben usarse para hacer que el programa sea fácil de leer.

2. ¿Qué es mejor, codificar un enunciado `if` compuesto o anidar varios enunciados `if`?

   Se debe hacer que el código sea fácil de entender. Si se anidan enunciados `if`, son evaluados como se vio en el capítulo. Si se usa un solo enunciado compuesto, las expresiones son evaluadas solamente hasta que el enunciado completo es evaluado como falso.

3. ¿Cuál es la diferencia entre operadores unarios y binarios?

   Como su nombre lo indica, los operadores unarios trabajan con una variable y los binarios con dos.

4. ¿Es el operador de resta (-) unario o binario?

   ¡Es ambos! El compilador es lo suficientemente listo como para saber cuál se está usando. El sabe cuál forma usar basándose en la cantidad de variables en la expresión que se está usando. En el siguiente enunciado, es unario

   ```
   x = -y;
   ```

   contra el uso binario que se muestra:

   ```
   x = a - b;
   ```

5. ¿Los números negativos son considerados ciertos o falsos?

   Recuerde que 0 es falso y cualquier otro valor es cierto. Esto incluye los números negativos.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado y ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Cómo se le llama al siguiente enunciado C y cuál es su significado?

   ```
   x = 5 + 8;
   ```

2. ¿Qué es una expresión?
3. En una expresión que contiene varios operadores, ¿qué es lo que determina el orden en el que se ejecutan las operaciones?
4. Si la variable `x` tiene el valor de 10, ¿cuáles son los valores de `x` y `a` después de que cada uno de los siguientes enunciados se ejecuta por separado?

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

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

5. ¿Cuál es el resultado de la expresión 10 % 3?
6. ¿Cuál es el resultado de la expresión 5 + 3 * 8 / 2?
7. Reescriba la expresión de la pregunta 6, añadiendo paréntesis de tal forma que dé como resultado 16.
8. Si una expresión evalúa a falso, ¿qué valor tiene la expresión?
9. ¿Cuál tiene mayor precedencia?

   a. == o <

   b. * o +

   c. != o ==

   d. >= o >

10. ¿Qué son los operadores de asignación compuestos y para qué son útiles?

Ejercicios

1. El siguiente código no está bien escrito. Tecléelo y compílelo para ver si funciona.

```
#include <stdio.h>
int x,y;main(){ printf(
"\nEnter two numbers");scanf(
"%d %d",&x,&y);printf(
"\n\n%d is bigger",(x>y)?x:y);return 0;}
```

2. Vuelva a escribir el código del ejercicio 1 para que sea más legible.
3. Cambie el listado 4.1 para que cuente hacia arriba en vez de hacia abajo.
4. Escriba un enunciado `if` que asigne el valor de `x` a la variable y solamente si `x` se encuentra entre 1 y 20. Deje a `y` sin cambio cuando `x` no se encuentre en ese rango.
5. Use el operador condicional para ejecutar la misma tarea que en el ejercicio 4.
6. Vuelva a escribir los siguientes enunciados `if` anidados, usando un solo enunciado `if` y operadores compuestos.

```
if (x < 1)
   if (x > 10)
     enunciado;
```

7. ¿Cuál es el resultado de cada una de las siguientes expresiones?

```
a. (1 + 2 * 3)
b. 10 % 3 * 3 - (1 + 2)
c. ((1 + 2) * 3)
d. (5 == 5)
e. (x = 5)
```

8. Si `x` = 4, `y` = 6 y `z` = 2, determine si cada uno de los siguientes evalúa a cierto o falso.

```
a. if (x == 4)
b. if (x != y - z)
c. if (z = 1)
d. if (y)
```

9. Escriba un enunciado `if` que determine si alguien es legalmente un adulto (edad 21 años), pero no un anciano (edad 65 años).

10. BUSQUEDA DE ERRORES: Componga el siguiente programa para que ejecute correctamente.

```
/* Un programa con problemas...*/

#include <stdio.h>
int x= 1:
main()
{
    if( x = 1);
        printf(" x equals 1" );
    otherwise
        printf(" x does not equal 1");

    return
}
```

DIA 5

Funciones: lo básico

Las funciones son la parte central de la programación en C y de la filosofía de diseño de programas en C. Ya le han sido presentadas algunas funciones de biblioteca del C, que son funciones completas proporcionadas como parte del compilador. Este capítulo trata las funciones definidas por el usuario, las cuales, como su nombre lo indica, son funciones que uno, el programador, crea.

Hoy aprenderá

- ❑ Lo que es una función y cuáles son sus partes.
- ❑ Acerca de las ventajas de la programación estructurada con funciones.
- ❑ Cómo crear una función.
- ❑ Acerca de la declaración de variables locales en una función.
- ❑ La manera de regresar un valor desde una función al programa.
- ❑ La manera de pasar argumentos a una función.

¿Qué es una función?

Este capítulo responde a la pregunta: "¿Qué es una función?", de dos maneras. Primero le dice lo que son las funciones y luego le muestra la manera en que se usan.

La definición de una función

Primero la definición: una *función* es una sección de código de C que tiene un nombre, independiente, que ejecuta una tarea específica y que opcionalmente regresa un valor al programa que la llama. Ahora veamos las partes de esta definición.

Una función tiene nombre. Cada función tiene un nombre único. Con ese nombre, en cualquier otra parte del programa, se pueden ejecutar los enunciados contenidos en la función. A esto se le conoce como la llamada de la función. Una función puede ser llamada desde el interior de otra función.

Una función es independiente. Una función puede ejecutar su trabajo sin interferencias de, y sin interferir con, otras partes del programa.

Una función ejecuta una tarea específica. Esta es la parte fácil de la definición. Una tarea es un trabajo concreto que un programa debe ejecutar como parte de su operación general, como enviar una línea de texto a la impresora, ordenar un arreglo en orden numérico o calcular una raíz cúbica.

Una función puede regresar un valor al programa que la llama. Cuando el programa llama a una función, se ejecutan los enunciados que contiene. Estos, en caso de que se desee, pueden pasar información de regreso al programa que la llama.

Esto es todo por lo que se refiere a la definición. Téngalo presente mientras ve la siguiente sección.

La ilustración de una función

El programa que se presenta en el listado 5.1 contiene una función definida por el usuario. Los números de renglón no son parte del programa.

Captura

Listado 5.1. Un programa que usa una función para calcular el cubo de un número.

```
1:  /* Demuestra una función simple */
2:  #include <stdio.h>
3:
4:  long cube(long x);
5:
6:  long input, answer;
7:
8:  main()
9:  {
10:    printf("Enter an integer value: ");
11:    scanf("%d", &input);
12:    answer = cube(input);
13:    /* Nota: %ld es el especificador de conversión para */
14:    /* un entero largo */
15:    printf("\n\nThe cube of %ld is %ld.", input, answer);
16: }
17:
18: long cube(long x)
19: {
20:    long x_cubed;
21:
22:    x_cubed = x * x * x;
23:    return x_cubed;
24: }
```

A continuación se muestra la salida que produce la ejecución de este programa tres veces:

Salida

```
E:\>list0501
Enter an integer value: 100
The cube of 100 is 1000000.
E:\>list0501
Enter an integer value: 9
The cube of 9 is 729.
E:\>list0501
Enter an integer value: 3
The cube of 3 is 27.
```



Tome en cuenta que nos vamos a concentrar en los componentes del programa que se relacionan directamente con la función, en vez de explicar el programa completo.

Análisis La línea 4 contiene el *prototipo de función,* un modelo para una función que aparecerá posteriormente en el programa. Un prototipo de función contiene el nombre de la función, una lista de variables que se le deben pasar y el tipo de variable que regresa, en caso de haberla. Viendo la línea 4, se puede decir que la función es llamada cube, que requiere una variable de tipo `long` y que regresará un valor de tipo `long`. La lista de variables que serán pasadas a la función son llamadas argumentos, y aparecen entre los paréntesis que se encuentran a continuación del nombre de la función. En este ejemplo el argumento de la función es `long x`. La palabra clave antes del nombre de la función indica el tipo de variable que regresa la función. En este caso es regresada una variable de tipo `long`.

La línea 12 llama a la función cube y le pasa la variable input como argumento de la función. El valor de retorno de la función es asignado a la variable `answer`. Observe que tanto input como answer son declaradas en la línea 6 como variables `long`, ajustándose al prototipo de función que se encuentra en la línea 4.

La función propiamente dicha es llamada la *definición de función.* En este caso es llamada cube, y está contenida en las líneas de programa 18 a 24. De manera similar al prototipo, la definición de función tiene varias partes. La función comienza con un encabezado de función en la línea 18. El encabezado de función se encuentra al inicio de la función y le da su nombre a la función (en este caso, el nombre `es cube`). El encabezado también da el tipo de retorno de la función y describe sus argumentos. Observe que el encabezado de función es idéntico al prototipo de función (a excepción del punto y coma).

El cuerpo de la función, líneas 19 a 24, se encuentra encerrado entre llaves. El cuerpo contiene enunciados, como el que se muestra en la línea 22, que se ejecutan cada vez que es llamada la función. La línea 20 es una declaración de variable, que se parece a las declaraciones que se han visto anteriormente, pero con una diferencia: es local. Las variables *locales* son aquellas que son declaradas dentro del cuerpo de una función. (Las declaraciones locales se tratan a mayor detalle en el Día 12, "Alcance de las variables".) Por último, la función termina con un enunciado `return` en la línea 23 que marca el fin de la función. Un enunciado `return` también regresa un valor al programa que la llamó. En este caso es regresado el valor de la variable `x_cube`.

Si se compara la estructura de la función `cube()` con la de la función `main()` se verá que son la misma. `main()` es también una función. Otras funciones que ya se han usado son `printf()` y `scanf()`. Aunque `printf()` y `scanf()` son funciones de biblioteca (en vez de ser funciones definidas por el usuario), pueden recibir argumentos y regresar valores, de manera similar a las funciones que uno crea.

La manera en que trabaja una función

Un programa en C no ejecuta los enunciados de una función sino hasta que ella es llamada por otra parte del programa. Cuando una función es llamada, el programa puede enviar la información para la función en forma de uno o más argumentos. Un argumento es un dato del programa que es necesario para que la función ejecute su tarea. Luego los enunciados de la función ejecutan, realizando la tarea para la cual fueron diseñados. Cuando terminan los enunciados de la función, la ejecución regresa a la misma posición en el programa de donde fue llamada la función. Las funciones pueden enviar información de regreso al programa en forma de un valor de retorno.

La figura 5.1 muestra un programa con tres funciones, y cada una de ellas es llamada una vez. Cada vez que es llamada una función, la ejecución pasa a esa función. Cuando termina la función, la ejecución regresa al lugar de donde fue llamada la función. Una función puede ser llamada tantas veces como se necesite y las funciones pueden ser llamadas en cualquier orden.

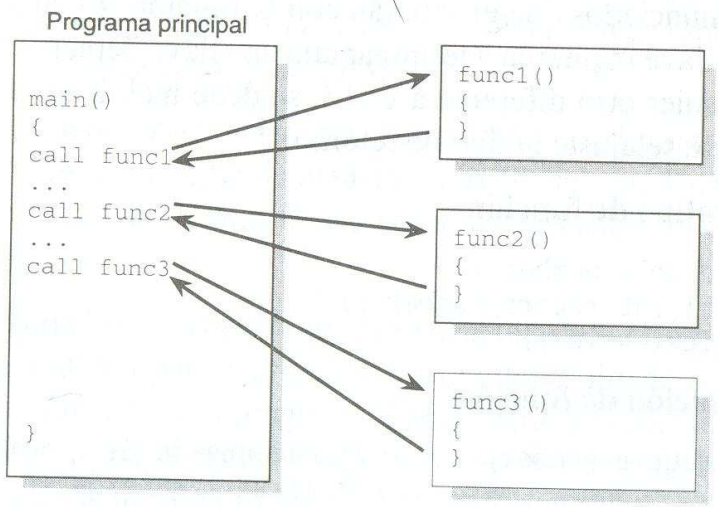


Figura 5.1. *Cuando un programa llama a una función, la ejecución pasa a la función y luego regresa al programa que la llamó.*

Ahora usted sabe lo que es una función y la importancia de las funciones. A continuación se presentan lecciones sobre la manera de crear y usar sus propias funciones.

Sintaxis

Funciones

Prototipo de función

```
tipo_de_retorno nombre_de_función (tipo-de-argumento nombre-1,..., tipo-de-argumento nombre-n);
```

Definición de función

```
tipo_de_retorno nombre_de_función (tipo-de-argumento nombre-1,..., tipo-de-
argumento nombre-n)

{          enunciados;}
```

Un *prototipo de función* proporciona al compilador la descripción de una función que será definida más adelante en el programa. El prototipo incluye un tipo de retorno, que indica el tipo de variable que regresará la función. También incluye el nombre de la función, que deberá describir lo que hace la función. El prototipo también contiene el tipo de las variables de los argumentos (`tipo-de-argumento`) que serán pasadas a la función. Opcionalmente puede contener los nombres de las variables que serán pasadas. Un prototipo siempre termina con un punto y coma.

Una *definición de función* es, de hecho, la función. La definición contiene el código que será ejecutado. La primera línea de la definición de función, llamada el *encabezado de función*, debe ser idéntico al prototipo de función, a excepción del punto y coma. Un encabezado de función no debe terminar con un punto y coma. Adicionalmente, aunque los nombres de variable de los argumentos son opcionales en el prototipo, deben ser incluidos en el encabezado de función. A continuación del encabezado se encuentra el cuerpo de la función, que contiene los enunciados que ejecutarán con la función. El cuerpo de la función debe comenzar con una llave izquierda y terminar con una llave derecha. Si el tipo de retorno de la función es cualquier otro diferente a `void`, se debe incluir un enunciado `return` que regrese un valor que se ajuste al tipo de retorno.

Ejemplos de prototipo de función

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

Ejemplos de definición de función

```
double squared( double number)          /* encabezado de función */
{                                       /* llave izquierda */
       return(  number * number );      /* cuerpo de la función */
}                                       /* llave derecha */
print_report( int report_number )
{
       if( report_number == 1 )
           puts( "Printing Report 1" );
       else
           puts( "Not printing Report 1 );
}
```

Las funciones y la programación estructurada

Usando funciones en los programas en C se puede practicar la *programación estructurada,* en la cual las tareas individuales del programa se ejecutan por secciones independientes de código de programa. "Secciones independientes de código de programa" se oye como parte de la definición de función que se dio anteriormente, ¿no es así? Las funciones y la programación estructurada están íntimamente relacionadas.

Las ventajas de la programación estructurada

¿Por qué es tan importante la programación estructurada? Hay dos razones importantes:

- Es más fácil escribir un programa estructurado, ya que los problemas complejos de programación son divididos en varias tareas más pequeñas y más simples. Cada tarea se ejecuta por una función, en la cual el código y las variables se encuentran aislados del resto del programa. Se puede avanzar más rápido resolviendo uno a la vez con estas tareas relativamente simples.
- Es mas fácil depurar un programa estructurado. Si el programa tiene un *error* (algo que hace que trabaje de manera equivocada), un diseño estructurado facilita el aislamiento del problema en una sección específica del código (una función específica).

Una ventaja relacionada con la programación estructurada es el tiempo que se puede ahorrar. Si se escribe una función para ejecutar una tarea determinada en un programa, rápida y fácilmente puede usarse en otro programa que necesite ejecutar la misma tarea. Incluso si el nuevo programa necesita realizar una tarea ligeramente diferente, muchas veces se encuentra que modificar una función que se ha creado anteriormente es más fácil que escribir una nueva a partir de cero. Considere qué tanto ha usado las funciones `printf()` y `scanf()`, aun cuando usted no ha visto el código que contienen. Si las funciones han sido creadas para hacer una sola tarea, es muy fácil usarlas en otros programas.

La planeación de un programa estructurado

Si se va a escribir un programa estructurado, primero se necesita hacer algo de planeación. Esta debe realizarse antes de comenzar a escribir una sola línea de código y, para hacerla, por lo general no se necesita más que lápiz y papel. El plan debe consistir en una lista de las

tareas específicas que ejecutará el programa. Comience con una idea global del objetivo del programa. Si se está planeando un programa para manejar una lista de nombres y direcciones ¿qué quiere que haga el programa? Estas son algunas cosas obvias:

Teclear nuevos nombres y direcciones.
Modificar entradas existentes.
Ordenar las entradas por apellido.
Escribir etiquetas para el correo.

Con esta lista se ha dividido al programa en cuatro tareas principales, y cada una de ellas puede ser asignada a una función. Ahora se puede dar un paso más, dividiendo estas tareas en subtareas. Por ejemplo, la tarea de "teclear nuevos nombres y direcciones" puede ser subdividida en estas subtareas:

Leer del disco la lista de direcciones existente.
Pedirle al usuario que teclee una o más entradas.
Añadir los nuevos datos a la lista.
Guardar la lista actualizada en el disco.

De manera similar, la tarea de "modificar entradas existentes" puede ser subdividida de la manera siguiente:

Leer del disco la lista de direcciones existente.
Modificar una o más entradas.
Guardar la lista actualizada en el disco.

Puede haberse dado cuenta de que estas dos listas tienen dos subtareas en común, las que se refieren a la lectura y el guardado de la lista en el disco. Se puede escribir una función para "leer del disco la lista de direcciones existente", y que esa función sea llamada tanto por la función "teclear nuevos nombres y direcciones" como por la de "modificar entradas existentes". Lo mismo se aplica para "guardar la lista actualizada en disco".

Por lo anterior, usted ya debe ver por lo menos una ventaja de la programación estructurada. Dividiendo cuidadosamente el programa en tareas se pueden identificar las partes del programa que comparten tareas comunes. Se pueden escribir funciones de acceso a disco de "doble trabajo", ahorrándose tiempo y haciendo que el programa sea más pequeño y más eficiente.

Este método de programación da como resultado una estructura de programa jerárquica o en capas. La figura 5.2 ilustra la programación jerárquica para el programa de lista de direcciones.

Cuando se sigue este enfoque planificado, rápidamente se hace una lista de las tareas concretas que necesita ejecutar el programa. Luego se pueden resolver las tareas de una en una, poniendo toda la atención en una tarea relativamente simple. Cuando esa función está escrita y funciona adecuadamente, se puede pasar a la siguiente tarea. Antes de que lo note, el programa comienza a tomar forma.

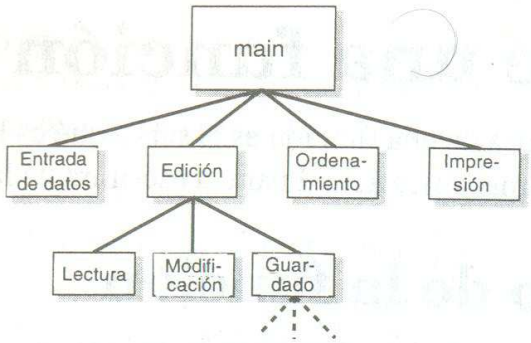


Figura 5.2. *Un programa estructurado está organizado jerárquicamente.*

El enfoque descendente

Usando la programación estructurada los programadores en C toman el *enfoque descendente*. Se vio esto ilustrado en la figura 5.2, donde la estructura del programa se parece a un árbol invertido. Muchas veces la mayoría del trabajo real del programa se ejecuta por las funciones que se encuentran "en la punta de las ramas". Las funciones más cercanas al tronco dirigen la ejecución del programa primeramente entre esas funciones.

Como resultado, muchos programas en C tienen una pequeña cantidad de código en el cuerpo principal del programa, esto es, en `main()`. El grueso del código de programa se encuentra en las funciones. En `main()` todo lo que se puede encontrar son unas cuantas líneas de código que dirigen la ejecución del programa entre las funciones. Por lo general se presenta un menú a la persona que usa el programa, con ramificaciones de la ejecución del programa de acuerdo a la selección del usuario.

Este es un buen método para diseñar programas. El Día 13, "Más sobre el control de programa", le muestra la manera en que puede usar al enunciado `switch` para crear un sistema versátil manejado por menús.

Ahora que ya sabe lo que son las funciones y por qué son tan importantes, ha llegado el momento de que aprenda cómo escribir sus propias funciones.

DEBE	NO DEBE
DEBE Planificar antes de comenzar a escribir el código. Comenzando con la determinación de la estructura del programa, se puede ahorrar tiempo en la escritura y depuración del código.	**NO DEBE** Tratar de hacer todo en una sola función. Una sola función debe hacer una sola tarea, como la lectura de información de un archivo.

Escritura de una función

El primer paso en la escritura de una función es el saber qué es lo que se quiere que haga la función. Luego de esto, la mecánica actual para la escritura de la función no es muy difícil.

El encabezado de la función

La primera línea de cada función es el encabezado de función, que tiene tres componentes, sirviendo cada uno a una función específica. Se encuentran diagramados en la figura 5.3 y explicados en el siguiente texto.

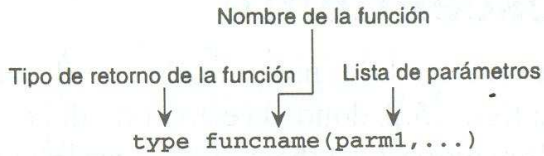


Figura 5.3. *Los tres componentes de un encabezado de función.*

El tipo de retorno de la función

El tipo de retorno de la función especifica el tipo de dato que regresa la función al programa que la llama. El tipo de retorno puede ser cualquiera de los tipos de datos del C: `char`, `int`, `long`, `float` o `double`. También se puede definir una función que no regrese un valor, teniendo un tipo de retorno `void`. A continuación se presentan algunos ejemplos:

```
int func1(...)          /* regresa un tipo int.   */
float func2(...)        /* regresa un tipo float. */
void func3(...)         /* no regresa nada.       */
```

El nombre de la función

Se le puede dar a la función cualquier nombre que se desee, siempre y cuando se sigan las reglas para nombres de variables del C (véase el Día 3, "Variables y constantes numéricas"). Un nombre de función debe ser único (no estar asignado a ninguna otra función o variable). Es una buena idea asignar un nombre que refleje lo que hace la función.

La lista de parámetros

Muchas funciones usan *argumentos*, que son valores pasados a la función cuando es llamada. Una función necesita saber qué tipos de argumentos espera, el tipo de dato de cada argumento. Se le puede pasar a una función cualquiera de los tipos de datos del C. La información sobre el tipo de argumentos es proporcionada en el encabezado de función por la lista de parámetros.

Para cada argumento que es pasado a la función debe contener una entrada la lista de parámetros. Esta entrada especifica el tipo de dato y el nombre del parámetro. Por ejemplo, a continuación se presenta el encabezado para la función que se encuentra en el listado 5.1:

```
long cube(long x)
```

La lista de parámetros dice `long x`, especificando que esta función toma un argumento de tipo `long` representado por el parámetro `x`. Si hay más de un parámetro, cada uno debe estar separado por una coma. El encabezado de función

```
void func1(int x, float y, char z)
```

especifica una función con tres argumentos: uno tipo `int` llamado `x`, otro tipo `float` llamado `y` y otro de tipo `char` llamado `z`. Algunas funciones no usan argumentos, y en esos casos la lista de parámetros debe decir `void`:

```
void func2(void)
```

No se pone un punto y coma al final del encabezado de función. Si por error se le pone, el compilador genera un mensaje de error.

Algunas veces se presentan confusiones acerca de la distinción entre un parámetro y un argumento. Un *parámetro* es una entrada en un encabezado de función y sirve como un relleno para un argumento. Los parámetros de la función son fijos y no cambian durante la ejecución del programa.

Un argumento es un valor actual, pasado a la función por el programa que la llama. Cada vez que la función es llamada se le pueden pasar argumentos diferentes. A una función se le deben pasar la misma cantidad y tipo de argumentos cada vez que es llamada, pero los valores de los argumentos pueden ser diferentes. En la función el argumento es accesado usando el nombre de parámetro correspondiente.

Un ejemplo puede aclarar esto. El listado 5.2 presenta un programa muy simple, con una función que es llamada dos veces.

Listado 5.2. La diferencia entre argumentos y parámetros.

```
1:  /* Ilustra la diferencia entre argumentos y parámetros. */
2:
3:  #include <stdio.h>
4:
5:  float x = 3.5, y = 65.11, z;
6:
7:  float half_of(float k);
8:
9:  main()
10: {
```

Listado 5.2. continuación

```
11:     /* En esta llamada x es el argumento para half_of(). */
12:     z = half_of(x);
13:     printf("The value of z = %f\n", z);
14:
15:     /* En esta llamada y es el argumento para half_of(). */
16:     z = half_of(y);
17:     printf("The value of z = %f\n", z);
18: }
19:
20: float half_of(float k)
21: {
22:     /* k es el parámetro. Cada vez que half_of() es llamado, */
23:     /* k tiene el valor que fue pasado como argumento. */
24:
25:     return (k/2);
26: }
```

Salida

```
The value of z = 1.750000
The value of z = 32.555000
```

La figura 5.4 muestra esquemáticamente la relación entre argumentos y parámetros.

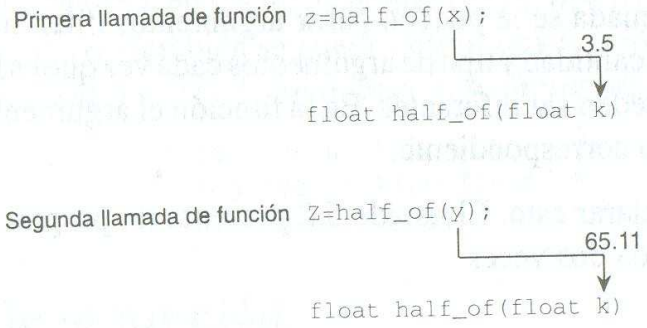


Figura 5.4. *Cada vez que es llamada una función, los argumentos son pasados a los parámetros de la función.*

Análisis

En el listado 5.2 se puede ver que el prototipo de la función `half_of()` está declarado en la línea 7. Las líneas 12 y 16 llaman a `half_of()`, y las líneas 20-26 contienen la función actual. Las líneas 12 y 16 envían cada una un argumento diferente a `half_of()`. La línea 12 envía `x`, que contiene un valor de 3.5, y la línea 16 envía `y`, que contiene un valor de 65.11. Cuando el programa ejecuta, imprime el número correcto para cada una de ellas. Los valores que se encuentran en `x` y `y` son pasados al argumento `k` de `half_of()`. Esto es como copiar el valor de `x` a `k`, y luego el de `y` a `k`. `Half_of()` regresa luego este valor después de haberlo dividido entre 2 (línea 25).

DEBE	NO DEBE

DEBE Usar un nombre de función que describa el objeto de la función.

NO DEBE Pasar valores innecesarios a una función.

NO DEBE ¡Tratar de pasar menos (o más) argumentos que parámetros a una función!

El cuerpo de la función

El *cuerpo de la función* se pone entre llaves, y se encuentra inmediatamente después del encabezado de función. Aquí es donde se hace el trabajo real. Cuando una función es llamada, la ejecución comienza en el inicio del cuerpo de la función, y termina (regresa al programa que la llamó) cuando se encuentra un enunciado `return` o cuando la ejecución llega a la llave derecha.

Variables locales

Se pueden declarar variables dentro del cuerpo de la función. Las variables declaradas en una función son llamadas *variables locales*. El término *local* significa que las variables son privadas de esa función particular, y distintas de otras variables que tengan el mismo nombre y que hayan sido declaradas en cualquier otro lugar del programa. Esta es una explicación breve. Por ahora, usted debe aprender la manera de declarar variables locales.

Una variable local se declara en la misma forma que cualquier otra variable, con los mismos tipos de variable y reglas para los nombres que se aprendieron en el Día 3, "Variables y constantes numéricas". Las variables locales también pueden ser inicializadas cuando son declaradas. Se pueden declarar variables de cualquier tipo en una función. A continuación se presentan algunos ejemplos:

```
int func1(int y)
{
    int a, b = 10;
    float tasa;
    double costo = 12.55;
...
}
```

Las declaraciones anteriores crean las variables locales `a, b, tasa` y `costo`, que pueden usarse por el código de la función. Note que los parámetros de la función son considerados como declaraciones de variables, por lo que, en caso de haberlas, las variables que se encuentren en la lista de parámetros también están disponibles.

Cuando se declara y usa una variable en una función está totalmente separada, y es distinta de cualquier otra variable que se haya declarado en cualquier otro lugar del programa. Esto es cierto incluso si las variables tienen el mismo nombre. El programa que se encuentra en el listado 5.3 muestra esta independencia.

Captura

Listado 5.3. Demostración de las variables locales.

```
1:  /* Demuestra las variables locales */
2:
3:  #include <stdio.h>
4:
5:  int x = 1, y = 2;
6:
7:  void demo(void);
8:
9:  main()
10: {
11:   printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
12:   demo();
13:   printf("\nAfter calling demo(), x = %d and y = %d.", x, y);
14: }
15:
16: void demo(void)
17: {
18:     /* Declara e inicializa dos variables locales */
19:
20:     int x = 88, y = 99;
21:
22:     /* Despliega sus valores. */
23:
24:     printf("\nWithin demo(), x = %d and y = %d.", x, y);
25: }
```

Salida

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

Análisis

El listado 5.3 es similar a los programas anteriores de este capítulo. La línea 5 declara a las variables `x` y `y`. Ellas son declaradas fuera de cualquier función y, por lo tanto, consideradas globales. La línea 7 contiene el prototipo para nuestra función de muestra, llamada `demo()`. Es una función que no toma ningún parámetro, y por lo tanto tiene `void` en el prototipo. Tampoco regresa ningún valor, por lo que se le da un tipo de `void`. La línea 9 inicia la función `main()` que es muy simple. En primer lugar es llamada `printf()` en la línea 11 para desplegar los valores de `x` y `y`, y luego es llamada la función `demo()`. Observe que `demo()` declara su propia versión local de `x` y `y` en la línea 20. La línea 24 muestra que las variables locales tienen precedencia sobre cualquier otra. Después

de que es llamada la función demo, la línea 13 nuevamente imprime los valores de `x` y `y`. Debido a que ya no se está en `demo()`, son impresos los valores globales originales.

Como puede ver, las variables locales `x` y `y` de la función son totalmente independientes de las variables globales `x` y `y` que están declaradas fuera de la función. Tres reglas gobiernan el uso de las variables en las funciones.

- Para usar una variable en una función se le debe declarar en el encabezado de función o en el cuerpo de la función (a excepción de las variables globales, tratadas en el Día 12, "Alcance de las variables").
- Para que una función obtenga un valor del programa que la llama, el valor debe ser pasado como argumento.
- Para que un programa que llama obtenga un valor de una función, el valor debe ser regresado explícitamente por la función.

Para ser honestos estas "reglas" no se aplican estrictamente, debido a que posteriormente en el libro aprenderá cómo evadirlas. ¡Sin embargo, por el momento siga estas reglas y se evitará problemas!

Mantener separadas las variables de la función de las otras variables del programa es una de las formas en que las funciones son independientes. Una función puede ejecutar cualquier tipo de manipulación de datos que se desee usando su propio juego de variables locales. No hay por qué preocuparse de que estas manipulaciones tengan efectos secundarios en otra parte del programa.

Enunciados de función

Esencialmente no hay limitación sobre los enunciados que pueden ser incluidos dentro de una función. Lo único que no se puede hacer en el interior de una función es definir otra función. Sin embargo, se pueden usar todos los otros enunciados del C, incluidos los ciclos (tratados en el Día 6, "Control básico del programa"), enunciados `if` y enunciados de asignación. Se pueden llamar funciones de biblioteca y otras funciones definidas por el usuario.

¿Qué hay acerca del largo de la función? El C no pone restricciones de longitud a las funciones, pero para fines prácticos se deben mantener las funciones relativamente cortas. Recuerde que en la programación estructurada se supone que cada función va a ejecutar una tarea relativamente simple. Si se da cuenta de que una función se está haciendo muy larga, probablemente se deba a que está tratando de ejecutar una tarea demasiado compleja para ser realizada por una sola función. Probablemente puede ser dividida en dos o más funciones más pequeñas.

¿Qué tanto es demasiado largo? No hay una respuesta definitiva a esta pregunta, pero en la práctica es raro encontrar una función que sea más larga de 25 o 30 renglones de código. Se

tiene que usar el criterio propio. Algunas tareas de programación requieren funciones más largas, aunque muchas funciones son sólo de unas cuantas líneas. Conforme obtenga experiencia en la programación podrá decidir fácilmente lo que debe cortarse o no en funciones más pequeñas.

Regreso de un valor

Para regresar un valor de una función se usa la palabra clave `return` seguida por una expresión de C. Cuando la ejecución llega al enunciado `return` la expresión es evaluada, y la ejecución pasa el valor de regreso al programa que hizo la llamada. El valor de retorno de la función es el valor de la expresión. Vea la siguiente función:

```
int func1(int var)
{
    int x;
    ...
    ...
    return x;
}
```

Cuando esta función es llamada, ejecutan los enunciados de la función hasta llegar al enunciado `return`. El `return` termina la función y regresa el valor de `x` al programa que la llamó. Las expresiones que se encuentran a continuación de la palabra clave `return` pueden ser cualquier expresión válida del C.

Una función puede contener varios enunciados `return`. El primer `return` que ejecuta es el único que tiene efecto. Usar varios enunciados `return` es una manera eficiente de regresar diferentes valores de una función. Véase el ejemplo del listado 5.4.



Listado 5.4. Demostración del uso de varios enunciados return en una función.

```
1:  /* Demuestra el uso de varios enunciados return en una función. */
2:
3:  #include <stdio.h>
4:
5:  int x, y, z;
6:
7:  int larger_of( int a, int b);
8:
9:  main()
10: {
11:     puts("Enter two different integer values: ");
12:     scanf("%d%d", &x, &y);
13:
14:     z = larger_of(x,y);
15:
```

```
16:     printf("\nThe larger value is %d.", z);
17: }
18:
19: int larger_of( int a, int b)
20: {
21:     if (a > b)
22:         return a;
23:     else
24:         return b;
25: }
```

Salida

```
E:\>list0504
Enter two different integer values:
200 300

The larger value is 300.

E:\>list0504
Enter two different integer values:
300
200

The larger value is 300.
```

Análisis

De manera similar a otros ejemplos, el listado 5.4 se inicia con un comentario para describir lo que hace el programa (línea 1). El archivo de encabezado STDIO.H es incluido para las funciones de entrada/salida estándar, que le permiten al programa desplegar información en la pantalla y obtener entrada de datos del usuario. La línea 7 es el prototipo de función para `larger_of()`. Observe que usa dos variables `int` como parámetros y regresa un `int`. La línea 14 llama a `larger_of()` con `x` y `y`. La función `larger_of()` tiene varios enunciados `return`. Con un enunciado `if`, la función revisa en la línea 21 para ver si `a` es mayor que `b`. Si es así, la línea 22 ejecuta un enunciado `return` y la función termina inmediatamente. Las líneas 23 y 24 son ignoradas en este caso. Si `a` no es mayor que `b`, es saltada la línea 22, se ejecuta la cláusula `else` y se ejecuta el `return` que se encuentra en la línea 24. Como puede ver, dependiendo de los argumentos que se le pasen a la función `larger_of()`, será ejecutado el primero o el segundo enunciado `return`, y el valor apropiado será pasado de regreso a la función que la llamó.

Una nota final sobre este programa. La línea 11 es una nueva función que no se había visto antes. `puts()` (lea *put string*) es una función simple que despliega una cadena en la salida estándar, que por lo general es la pantalla de la computadora. (Las cadenas se tratan en el Día 10, "Caracteres y cadenas". Por ahora, simplemente sepa que son el texto entre comillas.)

Recuerde que el valor de retorno de una función tiene un tipo que es especificado en el encabezado de función y en el prototipo de función. El valor regresado por la función debe ser del mismo tipo, ya que si no el compilador genera un mensaje de error.

El prototipo de la función

Un programa debe incluir un prototipo para cada función que use. Se vio un ejemplo de prototipo de función en la línea 4 del listado 5.1, y ha habido prototipos de función también en los otros listados. ¿Qué es un prototipo de función y para qué se necesita?

Puede ver en los ejemplos anteriores que el prototipo de una función es idéntico al encabezado de la función, con un punto y coma añadido al final. De manera similar al encabezado de función, el prototipo de función incluye información acerca del tipo de retorno, el nombre y los parámetros de la función. El objeto del prototipo es darle información al compilador sobre el tipo de retorno, el nombre y los parámetros de la función. Con esta información el compilador puede hacer una revisión cada vez que el código fuente llame a la función, y verificar que se están pasando la cantidad y tipo correctos de argumentos a la función y se está usando correctamente el valor de retorno. Si hay alguna discordancia, el compilador generará un mensaje de error.

Hablando estrictamente, un prototipo de función no necesita ser exactamente igual que el encabezado de función. Los nombres de parámetros pueden ser diferentes, siempre y cuando sean del mismo tipo, cantidad y estén en el mismo orden. No hay razón para que el encabezado y el prototipo no concuerden. Al tenerlos idénticos se facilita la comprensión del código fuente, y también facilita la escritura del programa. Cuando se completa una definición de función, use la característica de cortar y pegar del editor, para copiar el encabezado de función y crear el prototipo. Asegúrese de añadir un punto y coma al final.

¿Dónde deben ponerse los prototipos de función en el código fuente? Deben ser puestos antes del inicio de `main()` o antes de que la función sea definida por primera vez. Para mejorar la legibilidad, lo mejor es agrupar todos los prototipos en una sola posición.

DEBE	NO DEBE

NO DEBE Tratar de regresar un valor que tiene un tipo diferente al tipo de la función.

DEBE Usar variables locales siempre que sea posible.

NO DEBE Dejar que las funciones se hagan muy largas. Si una función comienza a hacerse muy larga, trate de partirla en tareas más pequeñas.

DEBE Limitar cada función a una sola tarea.

NO DEBE Poner varios enunciados `return` si no son necesarios. Se debe tratar de tener un solo return cuando sea posible. Sin embargo, algunas veces el tener varios enunciados `return` es más fácil y claro.

Paso de argumentos a una función

Para pasar argumentos a una función se les lista entre paréntesis a continuación del nombre de la función. La cantidad de argumentos y el tipo de cada uno de ellos debe coincidir con los parámetros del encabezado y prototipo de función. Por ejemplo, si una función está definida para que tome dos argumentos de tipo `int`, se le deben pasar exactamente dos argumentos `int`, ni más ni menos, y no de otro tipo. Si se trata de pasar a una función una cantidad y/o tipos de argumentos, el compilador lo detecta basado en la información que se encuentra en el prototipo de función.

Si la función usa varios argumentos, los argumentos listados en la llamada a la función son asignados a los parámetros de la función en orden: el primer argumento con el primer parámetro, el segundo argumento con el segundo parámetro y así sucesivamente, como se ilustra en la figura 5.5.

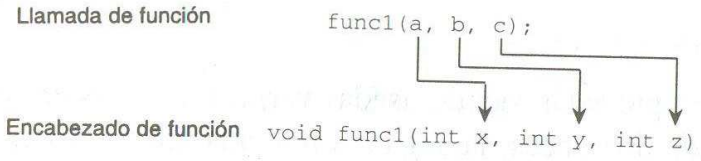


Figura 5.5. *Varios argumentos son asignados a los parámetros de la función en orden.*

Cada argumento puede ser una expresión válida del C: una constante, una variable, una expresión matemática o lógica, o incluso otra función (una que tenga un valor de retorno). Por ejemplo, si `half()`, `square()` y `third()` son funciones con valores de retorno, se podría escribir

```
x = half(third(square(half(y))));
```

El programa primero llama a `half()`, pasándole `y` como argumento. Cuando la ejecución regresa de `half()`, el programa llama a `square()`, pasándole el valor de retorno de `half()` como su argumento. A continuación es llamado `third()`, con el valor de retorno de `square()` como argumento. Luego es vuelto a llamar `half()`, y esta vez con el valor de retorno de `third()` como argumento. Por último, el valor de retorno de `half()` es asignado a la variable `x`. El siguiente es un fragmento de código equivalente:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

Llamado de funciones

Hay dos maneras de llamar una función. Cualquier función puede ser llamada simplemente con su nombre y lista de argumentos en un enunciado. Si la función tiene un valor de retorno es descartado. Por ejemplo,

```
wait(12);
```

El segundo método puede usarse solamente con funciones que tienen un valor de retorno. Como estas funciones dan como resultado un valor (esto es, su valor de retorno) son expresiones válidas del C, y pueden usarse en cualquier lugar donde pueda usarse una expresión de C. Ya ha visto una expresión con un valor de retorno usada en el lado derecho de un enunciado de asignación. A continuación se presentan otros ejemplos:

```
printf("Half of %d is %d. ", x, half_of(x));
```

En este ejemplo, `half_of()` es un parámetro de una función. Primero es llamada la función `half_of()` con el valor de `x` y luego es llamada `printf()` usando los valores `x` y `half_of(x)`.

```
y = half_of(x) + half_of(z);
```

En este segundo ejemplo están siendo usadas varias funciones en una expresión. Aunque `half_of()` es usada dos veces, la segunda llamada puediera haber sido cualquier otra función. El siguiente código muestra los mismos enunciados, pero sin estar todos en una línea.

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

Los dos ejemplos finales muestran maneras efectivas de usar los valores de retorno de las funciones:

```
if ( half_of(x) > 10 )
{
    enunciados              /* éste puede ser cualquier enunciado */
}
```

Aquí una función se está usando con el enunciado `if`. Si el valor de retorno de la función satisface el criterio (en este caso, si `half_of()` regresa un valor mayor que 10), el enunciado `if` es cierto y los enunciados se ejecutan. Si el valor regresado no satisface el criterio, los enunciados del `if` no se ejecutan. El siguiente ejemplo es todavía mejor:

```
if ( ejecuta_un_proceso() != OKAY )
{
   enunciados              /* ejecuta rutina de error */
}
```

Nuevamente no he dado los enunciados actuales ni `ejecuta_un_proceso()` es una

función real. Sin embargo, es un ejemplo importante que revisa el valor de retorno de un proceso para ver si ejecutó correctamente. Si lo hizo, los enunciados se encargan de cualquier manejo de errores o de limpieza. Esto es usado comúnmente cuando se accesa información en archivos, se comparan valores y se ubica memoria.

Si trata de usar una función con un tipo de retorno `void` en una expresión, el compilador genera un mensaje de error.

DEBE	NO DEBE
DEBE ¡Pasar parámetros a las funciones en orden para hacer a las funciones genéricas, y por lo tanto, reutilizables!	
DEBE Aprovechar la capacidad de poner funciones en expresiones.	
	NO DEBE Hacer confuso un enunciado individual poniendo muchas funciones en él. Sólo ponga funciones en los enunciados cuando no hagan el código más confuso.

Recursión

El término *recursión* se refiere a la situación en la que una función se llama a sí misma, directa o indirectamente. La *recursión indirecta* sucede cuando una función llama a otra función que a su vez llama a la primera función. El C permite las funciones recursivas y pueden ser útiles en algunas situaciones.

Por ejemplo, la recursión puede usarse para calcular el factorial de un número. El factorial de un número `x` es escrito `x!`, y calculado de la manera siguiente:

```
x! = x * (x - 1) * (x - 2) * (x - 3) ... * (2) * 1
```

Sin embargo, también se puede calcular `x!` de la manera siguiente:

```
x! = x * (x - 1)!
```

Yendo un paso más adelante, se puede calcular `(x-1)!` con el mismo procedimiento:

```
(x-1)! = (x - 1) * (x - 2)!
```

Se puede continuar calculando en forma recursiva hasta que se llega al valor de 1, y cuando éste es el caso, se ha terminado. El programa en el listado 5.5 usa una función recursiva para calcular factoriales. Como el programa usa enteros sin signo, está limitado a un valor inicial de 8. El factorial de 9 y de valores más grandes está fuera del rango permitido para los enteros.

Captura

Listado 5.5. El uso de una función recursiva para calcular factoriales.

```
1:  /* Demuestra la recursión de una función. Calcula el */
2:  /* Factorial de un número. */
3:
4:  #include <stdio.h>
5:
6:  unsigned int f, x;
7:  unsigned int factorial(unsigned int a);
8:
9:  main()
10: {
11:     puts("Enter an integer value between 1 and 8: ");
12:     scanf("%d", &x);
13:
14:     if( x > 8 || x < 1)
15:     {
16:         printf("Only values from 1 to 8 are acceptable!");
17:     }
18:     else
19:     {
20:         f = factorial(x);
21:         printf("%u factorial equals %u", x, f);
22:     }
23: }
24:
25: unsigned int factorial(unsigned int a)
26: {
27:     if (a == 1)
28:         return 1;
29:     else
30:     {
31:         a *= factorial(a-1);
32:         return a;
33:     }
34: }
```

Salida

```
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

Análisis

La primera parte de este programa es similar a muchos de los otros programas que ya se han visto. Comienza con comentarios en las líneas 1 y 2. En la línea 4 se incluye el archivo de encabezado adecuado para las rutinas de entrada/salida. La línea 6 declara un par de valores enteros sin signo. La línea 7 es un prototipo de función para la función de factorial. Observe que toma un `unsigned int` como parámetro y regresa un `unsigned int`. Las líneas 9 a 23 son la función `main()`. Las líneas 11 y 12 imprimen un mensaje que pide un valor del 1 al 8 y luego acepta el valor tecleado.

Las líneas 14 a 22 muestran un enunciado `if` interesante. Como un valor mayor que 8 causa problemas, este enunciado `if` revisa el valor. Si es mayor que 8, imprime un mensaje de error y, en caso contrario, el programa calcula el factorial en la línea 20 e imprime el resultado en la línea 21. Cuando sepa que puede haber problemas, como el límite en el tamaño de una cifra, añada código para detectar el problema y prevenirlo.

Nuestra función recursiva, `factorial()`, se encuentra en las líneas 14 a 22. El valor pasado es asignado a `a`. En la línea 27 es revisado el valor de `a`. Si es 1, el programa regresa el valor 1. Si el valor no es 1, es puesto `a` igual a sí mismo multiplicado por el valor de `factorial(a-1)`. El programa vuelve a llamar a la función factorial, pero esta vez el valor de `a` es `(a - 1)`. Si `(a - 1)` no es igual a 1, es vuelto a llamar `factorial()` con `((a - 1)-1)`, que es lo mismo que `(a - 2)`. Este proceso continúa hasta que el enunciado `if` de la línea 27 es cierto. Si el valor del factorial es 3, el factorial es evaluado a lo siguiente:

```
3 * (3 - 1) * ((3 - 1)-1)
```

DEBE	NO DEBE
DEBE Comprender y trabajar con la recursión antes de usarla.	**NO DEBE** Usar recursión si va a haber varias iteraciones. (Una iteración es la repetición de un enunciado de programa.) La recursión usa muchos recursos, ya que la función tiene que recordar dónde está.

Dónde se ponen las funciones

Tal vez se pregunte en qué parte del código fuente debe poner las definiciones de función. Por ahora deben ir en el mismo archivo de código fuente que `main()` y después del final de `main()`. La estructura básica de un programa que usa funciones se muestra en la figura 5.6.

Se pueden guardar las funciones definidas por el usuario en un archivo del código fuente separado, separado de `main()`. Esta técnica es útil con programas grandes y cuando se quiere usar el mismo juego de funciones en más de un programa. Esta técnica se trata en el Día 21, "Aprovechando las directivas del preprocesador y más".

```
/* start of source code */
   ...
   prototypes here
   ...
   main()
   {
   ...
   ...
   }
   func1()
   {
   ...
   }
   func2()
   {
   ...
   }
/* end of source code */
```

Figura 5.6. *Ponga los prototipos de función antes de* `main()` *y las definiciones de función después de* `main()`.

Resumen

Este capítulo le presentó las funciones, que son una parte importante de la programación en C. Las funciones son secciones independientes de código que ejecutan tareas específicas. Cuando el programa necesita que se ejecute una tarea llama a la función que ejecuta esa tarea. El uso de funciones es esencial para la programación estructurada, un método de diseño de programa que enfatiza el enfoque modular descendente. La programación estructurada crea programas más eficientes y también más fáciles de usarse por uno, el programador.

También se aprendió que una función consiste en encabezado y cuerpo. Aquél incluye información acerca del tipo de retorno, nombre y parámetros de la función; éste, declaraciones de variables locales y los enunciados del C que se ejecutan cuando es llamada la función. Por último, se vio que las variables locales, aquellas declaradas dentro de una función, son completamente independientes de cualquier otra variable de programa declarada en cualquier otro lado.

Preguntas y respuestas

1. ¿Qué pasa si necesito regresar más de un valor de una función?

   Muchas veces necesitará regresar más de un valor de una función, o lo que es más común, deseará cambiar un valor que le es enviado a la función y guardar el

cambio después de que termine la función. Esto se trata en el Día 18, "Obteniendo más de las funciones".

2. ¿Cómo sé qué tan bueno es el nombre de una función?

   Un buen nombre de función describe lo más específicamente posible lo que hace la función.

3. Cuando se declaran variables al principio del listado, antes de `main()`, pueden usarse en cualquier lugar, pero las variables locales sólo pueden usarse en la función específica. ¿Por qué no declarar todo antes de `main()`?

   En el Día 12, "Alcance de las variables" se trata a mayor detalle el alcance de las variables.

4. ¿Qué otras formas hay de usar la recursión?

   La función factorial es un primer ejemplo sobre el uso de la recursión. En muchos cálculos estadísticos se necesita el número del factorial. La recursión es simplemente un ciclo. Sin embargo, tiene una diferencia con respecto a otros ciclos. Con la recursión cada vez que es llamada una función recursiva se crea un nuevo juego de variables. Esto no es cierto en los otros ciclos que verá en el siguiente capítulo.

5. ¿Tiene que ser `main()` la primera función en un programa?

   No. Es un estándar en C que la función `main()` sea la primera función que ejecute. Sin embargo, puede ser puesta en cualquier lugar del archivo fuente. La mayoría de la gente la pone primero para que sea fácil de localizar.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado y ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Va a usar programación estructurada cuando escriba sus programas en C?
2. ¿Cómo funciona la programación estructurada?
3. ¿Dónde entran las funciones del C en la programación estructurada?
4. ¿Cuál debe ser la primera línea de una definición de función y qué información contiene?
5. ¿Qué tantos valores puede regresar una función?

6. Si una función no regresa un valor, ¿con qué tipo debe ser declarada?
7. ¿Cuál es la diferencia entre una definición de función y un prototipo de función?
8. ¿Qué es una variable local?
9. ¿En qué son especiales las variables locales?

Ejercicios

1. Escriba un encabezado para una función llamada `hazlo()`, que tome tres argumentos de tipo `char` y regrese un `tipo float` al programa que la llama.
2. Escriba un encabezado para una función llamada `imprime_un_número()`, que tome un solo argumento de tipo `int` y no regrese nada al programa que la llama.
3. ¿Qué tipo de valor regresan las siguientes funciones?

```
a. int imprime_error( float num_error);
b. long lee_registro( int num_reg, int longitud );
```

4. BUSQUEDA DE ERRORES: ¿Cuál es el error en el siguiente listado?

```
#include <stdio.h>
void print_msg( void );
main()
{
     print_msg( "This is a message to print" );
}

void print_msg( void )
{
     puts( "This is a message to print" );
     return 0;
}
```

5. BUSQUEDA DE ERRORES: ¿Cuál es el error en la siguiente definición de función?

```
int twice(int y);
{
    return (2 * y);
}
```

6. Vuelva a escribir el listado 5.4 de tal forma que sólo necesite un enunciado `return`.

7. Escriba una función que reciba dos números como argumentos y regrese el valor de su producto.
8. Escriba una función que reciba dos números como argumentos. La función debe dividir el primer número entre el segundo. No divida cuando el segundo número sea cero. (Consejo: Use un enunciado `if`.)
9. Escriba una función que llame a las funciones de los ejercicios 7 y 8.
10. Escriba un programa que use una función para encontrar el promedio de cinco valores `tipo float` tecleados por el usuario.
11. Escriba una función recursiva que calcule el valor de 3 a la potencia de otro número. Por ejemplo, si se le pasa 4, la función regresará 81.

DIA 6

Control básico del programa

En el Día 4, "Enunciados, expresiones y operadores", se trató al enunciado `if`, que da algo de control sobre el flujo de los programas. Sin embargo, muchas veces se necesita algo más que la simple habilidad de tomar decisiones sobre cierto o falso. Este capítulo presenta tres nuevas maneras de controlar el flujo del programa. Hoy aprenderá

- La manera de usar arreglos simples.
- La manera de usar ciclos `for`, `while` y `do...while` para ejecutar enunciados varias veces.
- Cómo se pueden anidar enunciados de control de programa.

Este capítulo no pretende dar un tratamiento completo de estos temas, pero sí suficiente información para que usted sea capaz de comenzar a escribir programas reales. Estos temas se tratan a mayor detalle en el Día 13, "Más sobre el control de programa".

Arreglos: lo básico

Antes de que tratemos al enunciado `for`, hagamos una pausa y aprendamos lo básico de los arreglos. (Véase el Día 8, "Arreglos numéricos" para una explicación a fondo de los arreglos.) El enunciado `for` y los arreglos están íntimamente relacionados en C, por lo que es díficil definir uno sin explicar el otro. Para ayudarle a comprender los arreglos, que se usan en los ejemplos del enunciado `for` que se presentan a continuación, se da una rápida explicación de los arreglos.

Un *arreglo* es un grupo indexado de ubicaciones de almacenamiento de datos que tienen el mismo nombre y se distinguen entre ellas por un *subíndice* o *índice,* un número que se pone a continuación del nombre de la variable encerrado entre corchetes. (Esto le quedará más claro conforme avance.) De manera similar a otras variables del C, los arreglos deben ser declarados. Una declaración de arreglo incluye tanto el tipo de dato como el tamaño de arreglo (la cantidad de elementos en el arreglo). Por ejemplo, el enunciado

```
int datos [1000];
```

declara a un arreglo llamado `datos` que tiene el tipo `int` y contiene 1,000 elementos. A los elementos individuales se hace referencia mediante subíndices, como `datos[0]` hasta `datos[999]`. El primer elemento es `datos[0]`, y no `datos[1]`. En otros lenguajes, como el BASIC, el primer elemento de un arreglo es 1, pero esto no es cierto en C.

Cada elemento de este arreglo es equivalente a una variable entera normal y puede ser usado en la misma forma. El subíndice de un arreglo puede ser otra variable del C, como en este ejemplo:

```
int datos[1000];
int contador;
contador = 100;
datos[contador] = 12;          /* Que es igual a datos[100] = 12  */
```

Esta ha sido una rápida introducción a los arreglos. Sin embargo, con esto debe ser capaz de comprender la manera en que se usan los arreglos en los ejemplos de programa que se encuentran posteriormente en este capítulo. Si todos los detalles de los arreglos no le han quedado claros, no se preocupe. Ya verá más acerca de los arreglos en el Día 8, "Arreglos numéricos".

DEBE	NO DEBE

NO DEBE Declarar arreglos con subíndices mayores de lo que necesita, ya que se desperdicia memoria.

NO DEBE Olvidar que en el C a los arreglos se les hace referencia comenzando con el subíndice 0 y no con 1.

Control de la ejecución del programa

El orden por omisión de ejecución en un programa de C es descendente. La ejecución comienza al principio de la función `main()`, y avanza enunciado por enunciado hasta que se llega al final de `main()`. Sin embargo, este orden rara vez se encuentra en los programas de C reales. El lenguaje C incluye una variedad de enunciados para el control de programa, que le permiten controlar el orden de la ejecución del programa. Ya ha aprendido la manera de usar el operador fundamental de decisiones del C, el enunciado `if`, por lo que exploraremos tres enunciados de control adicionales que encontrará útiles.

El enunciado *for*

El enunciado `for` es una construcción de programación del C que ejecuta un bloque de uno o más enunciados una determinada cantidad de veces. A veces es llamado el *ciclo for,* debido a que la ejecución del programa por lo general hace ciclos por los enunciados más de una vez. Ya ha visto unos cuantos enunciados `for`, que han sido usados en los ejemplos de programación anteriormente en este libro. Ahora se encuentra listo para ver la manera en que funciona el enunciado `for`.

Un enunciado `for` tiene la siguiente estructura:

```
for(inicial; condición; incremento)
    enunciado
```

`inicial`, `condición` e `incremento` son expresiones del C, y `enunciado` es un enunciado simple o compuesto del C. Cuando se encuentra un enunciado `for` durante la ejecución del programa, suceden los siguientes eventos:

1. La expresión *inicial* es evaluada. Lo *inicial* es por lo general un enunciado de asignación que pone una variable a un valor determinado.
2. La expresión de *condición* es evaluada. La *condición* es típicamente una expresión relacional.
3. Si la *condición* evalúa a falso (esto es, a cero), el enunciado `for` termina, y la ejecución pasa al primer enunciado que se encuentra a continuación del *enunciado* del `for`.
4. Si la *condición* evalúa a cierto (esto es, a diferente de cero), se ejecutan los *enunciados* que se encuentran dentro del `for`.
5. La expresión de *incremento* es evaluada y la ejecución regresa al paso dos.

La operación de un enunciado `for` se muestra esquemáticamente en la figura 6.1. Observe que el *enunciado* nunca ejecuta si la *condición* es falsa la primera vez que es evaluada.

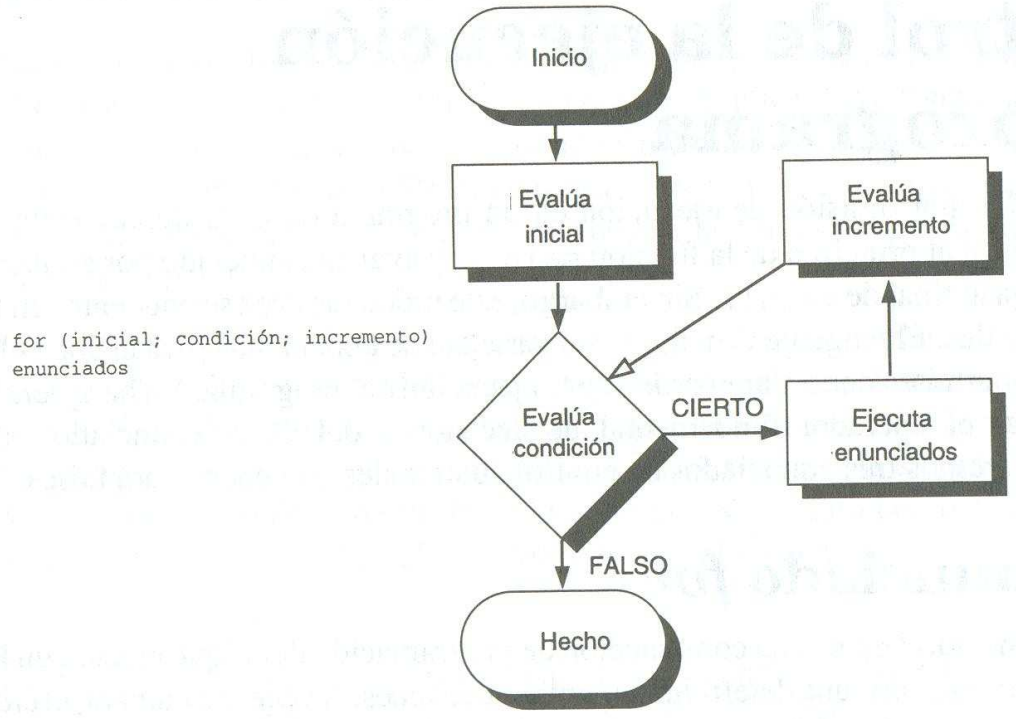


Figura 6.1. *Representación esquemática de un enunciado* `for`.

Aquí tenemos un ejemplo simple. El programa en el listado 6.1 usa un enunciado `for` para imprimir los números del 1 al 20. Puede ver que el código resultante es más compacto que como lo sería si fuera usado un enunciado `printf()` para cada uno de los 20 valores.

Captura

Listado 6.1. Demostración de un enunciado `for` simple.

```
1:  /* Demuestra un enunciado for simple */
2:
3:  #include <stdio.h>
4:
5:  int count;
6:
7:  main()
8:  {
9:      /* Imprime los números del 1 al 20 */
10:
11:     for (count = 1; count <= 20; count++)
12:         printf("\n%d", count);
13: }
```

Salida

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

El diagrama en la figura 6.2 ilustra la operación del ciclo `for` en el listado 6.1.

Análisis

La línea 3 incluye el archivo de encabezado de entrada/salida estándar. La línea 5 declara una variable de tipo `int` llamada `count`, que será usada en el ciclo `for`. Las líneas 11 y 12 son el ciclo `for`. Cuando se llega al enunciado `for`, se ejecuta primero el enunciado inicial. En este listado el enunciado inicial es `count=1`. Esto inicializa a `count` para que de esta forma pueda ser usado en el resto del ciclo. El segundo paso en la ejecución de este enunciado `for` es la evaluación de la condición `count <= 20`. Debido a que `count` acaba de ser inicializado a 1, se sabe que es menor que 20, por lo que el enunciado del comando `for`, `printf()`, se ejecuta. Después de ejecutar la función de impresión es evaluada la expresión de incremento, `count++`. Esto añade 1 a `count` haciendo que sea 2.

Ahora el programa regresa y revisa nuevamente la condición. Si es cierta, vuelve a ejecutar el `printf()`, y el incremento suma a `count` (haciendo que sea 3) y la condición es revisada. Este ciclo continúa hasta que la condición evalúa a falso, y en este punto el programa sale del ciclo y continúa en la siguiente línea (línea 13), que en este listado da por terminado al programa.

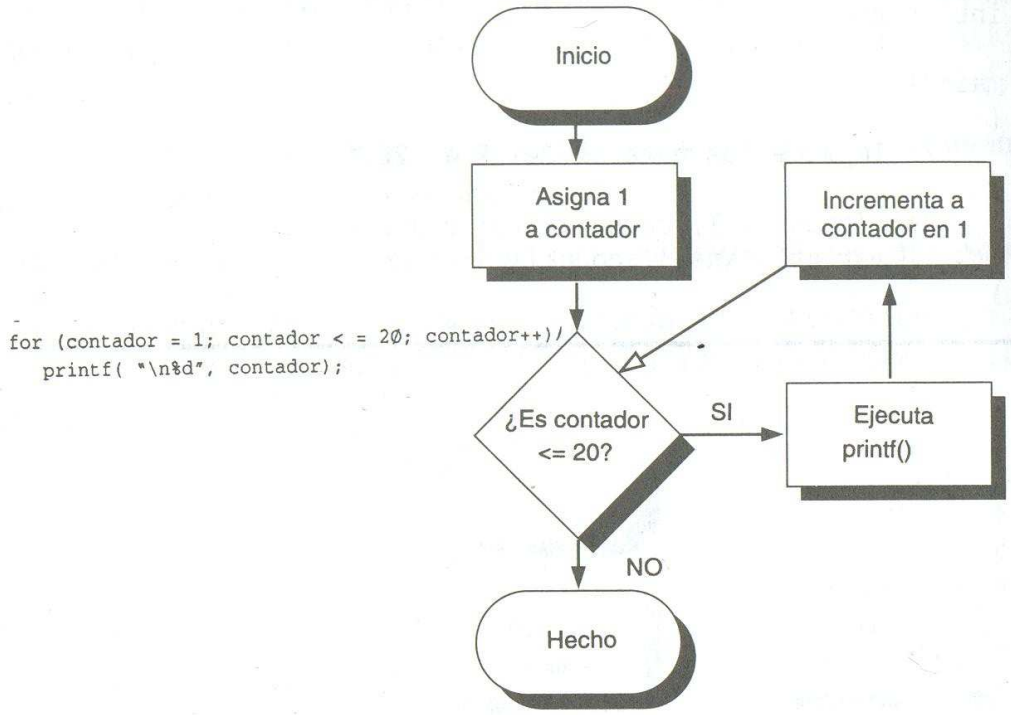


Figura 6.2. *La manera en que funciona el ciclo* `for` *del listado* 6.1.

El enunciado `for` es usado frecuentemente, como en el ejemplo anterior, para contar, incrementando un contador de un valor a otro. También se le puede usar para "contar al revés", disminuyendo en vez de incrementar la variable del contador.

```
for (contador = 100; contador > 0; contador--)
```

También se puede incrementar en un valor diferente de 1.

```
for (contador = 0; contador < 1000; contador += 5)
```

El enunciado `for` es bastante flexible. Por ejemplo, se puede omitir la expresión de inicialización si la variable que se ha de probar ha sido inicializada anterioremente en el programa. (Sin embargo, todavía se debe usar el separador de punto y coma, como se muestra.)

```
contador = 1;
for ( ; contador < 1000; contador++)
```

La expresión de inicialización no necesita ser de hecho una inicialización, sino que puede ser cualquier expresión válida del C. Sin importar lo que sea, se ejecuta una sola vez, cuando el enunciado `for` se ejecuta por primera vez. Por ejemplo, lo siguiente imprime "Ahora se ordena el arreglo..."

```
contador = 1;
for ( printf("Ahora se ordena el arreglo...") ; contador < 1000; contador++)
    /* Aquí van los enunciados para el ordenamiento */
```

También se puede omitir la expresión de incremento, ejecutando la actualización en el cuerpo del enunciado `for`. Nuevamente debe ser incluido el punto y coma. Por ejemplo, para imprimir los números del 0 al 99, se puede escribir

```
for (contador = 0; contador < 100; )
printf("%d", contador++);
```

La expresión de prueba que hace que termine el ciclo puede ser cualquier expresión de C. Mientras evalúe a cierto (diferente de cero) el enunciado `for` continúa ejecutando. Se pueden usar los operadores lógicos del C para construir expresiones de prueba complejas. Por ejemplo, el siguiente enunciado `for` imprime los elementos de un arreglo llamado `arreglo[]`, deteniéndose cuando todos los elementos han sido impresos o se ha encontrado un elemento con un valor de 0.

```
for (contador = 0; contador < 1000 && arreglo[contador] != 0; contador++)
    printf("%d", arreglo[contador]);
```

Se podría simplificar todavía más el ciclo `for` anterior, escribiéndolo de la manera siguiente. (Si no entiende los cambios hechos a las expresiones de prueba, necesita revisar el Día 4, "Enunciados, expresiones y operadores".)

```
for (contador = 0; contador < 1000 && arreglo[contador]; )
printf("%d", arreglo[contador++]);
```

Se puede poner a continuación del enunciado `for` un enunciado nulo, haciendo que todo el trabajo se ejecute en el mismo enunciado `for`. Recuerde que el enunciado nulo es un punto y coma sólo en una línea. Por ejemplo, para inicializar todos los elementos de un arreglo de 1,000 elementos al valor 50, se podría escribir

```
for (contador = 0; contador < 1000; arreglo[contador++] = 50)
;
```

En este enunciado `for` el valor de 50 es asignado a cada miembro del arreglo por la parte de incremento del enunciado.

En el Día 4, "Enunciados, expresiones y operadores", se mencionó que el operador de coma del C es usado a veces en los enunciados `for`. Se puede crear una expresión separando dos subexpresiones con el operador de coma. Las dos subexpresiones son evaluadas (en orden de izquierda a derecha) y la expresión completa evalúa al valor de la subexpresión que se encuentra a la derecha. Usando al operador de coma se puede hacer que cada parte de un enunciado `for` ejecute varias tareas.

Imagine que tiene dos arreglos de 1,000 elementos, `a[]` y `b[]`. Se quiere copiar el contenido de `a[]` a `b[]` en orden inverso, de forma tal que después de la operación de copia `b[0] = a[999], b[1] = a[998]` y así sucesivamente. El siguiente enunciado `for` hace el truco:

```
for ( i = 0, j = 999; i < 1000; i++, j--)
     b[j] = a[i];
```

El operador de coma es usado para inicializar dos variables, `i` y `j`. También es usado en la parte de incremento para modificar las dos variables dentro de cada ciclo.

Sintaxis

El enunciado *for*

`for(`*inicial*`;` *condición*`;` *incremento*`)` *enunciado(s)*

Lo `inicial` es cualquier expresión válida del C. Por lo general, es un enunciado de asignación, que pone una variable a un valor determinado.

La `condición` es cualquier expresión válida del C. Por lo general, es una expresión relacional. Cuando la `condición` evalúa a falso (0) termina el enunciado `for`, y la ejecución pasa al primer enunciado que se encuentra después del `enunciado` del `for`. En caso contrario se ejecutan los `enunciados` del `for`.

El `incremento` es cualquier expresión válida del C. Por lo general es una expresión que incrementa una variable que ha sido inicializada por la expresión inicial.

Los `enunciados` son los enunciados que se ejecutan mientras la condición permanezca cierta.

Un enunciado `for` es un enunciado de ciclo. Puede tener una inicialización, una prueba de condición y un incremento como parte del comando. El enunciado `for` ejecuta primero la expresión inicial. Luego revisa la condición, y en caso de que sea cierta se ejecutan los enunciados. Una vez que los enunciados se terminan, es evaluada la expresión de incremento. El enunciado `for` vuelve a revisar entonces la condición, y continúa haciendo ciclo hasta que la condición es falsa.

Ejemplo 1

```
/ * Imprime el valor de x al tiempo en que cuenta de 0 a 9 */
int x;
for( x = 0; x < 10; x++ )
     printf( "\nEl valor de x es %d", x );
```

Ejemplo 2

```
/* Pide cifras al usuario hasta que se teclea 99 */
int num = 0;
for( ; num != 99; )
      scanf( "%d", &num );
```

Ejemplo 3

```
/* Permite que el usuario teclee hasta 10 valores enteros.    */
/* Los valores son guardados en un arreglo llamado valor.     */
/* Si se teclea 99 el ciclo se detiene                        */
```

```
int valor[10];
int contador, numero=0;
for ( contador = 0; contador < 10 && numero != 99; contador++)
{
      puts ( "Teclee un número, 99 para terminar ");
      scanf ( "%d", &número);
      valor[contador] = número;}
```

Enunciados *for* anidados

Un enunciado `for` puede ser ejecutado dentro de otro enunciado `for`. A esto se le llama *anidado*. (Ya se vio esto en el Día 4, "Enunciados, expresiones y operadores", dentro del enunciado `if`.) Anidando enunciados `for` se puede hacer programación compleja. El listado 6.2 no es un programa complejo, pero ilustra el anidado de dos enunciados `for`.

Captura

Listado 6.2. Demostración de enunciados for anidados.

```
1:  /* Demuestra el anidado de dos enunciados for */
2:
3:  #include <stdio.h>
4:
5:  void draw_box( int row, int column);
6:
7:  main()
8:  {
9:      draw_box( 8, 35 );
10: }
11:
12: void draw_box( int row, int column )
13: {
14:     int col;
15:     for( ; row > 0; row-- )
16:     {
17:         for(col = column; col > 0; col-- )
18:             printf( "X" );
19:
20:         printf( "\n" );
21:     }
22: }
```

Salida

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Análisis El trabajo principal de este programa se realiza en la línea 18. Cuando se ejecuta este programa se imprimen 280 X en la pantalla, formando un cuadro de 8 por 35. El programa tiene solamente un comando para imprimir una X, pero se encuentra anidado en dos ciclos.

En este listado se declara en la línea 5 el prototipo de función para `draw_box()`. Esta función toma dos variables de tipo `int`, `row` y `column`, que contienen las dimensiones del cuadro de X que será trazado. En la línea 9 `main()` llama a `draw_box()`, y le pasa 8 como el valor de `row` y 35 como el valor de `column`.

Viendo detalladamente la función `draw_box()` se pueden ver unas cuantas cosas que no se entienden fácilmente. La primera es por qué se declara la variable local `col`. La segunda es por qué se usa el segundo `printf()` en la línea 20. Ambas cosas se verán más claras después de observar los dos ciclos `for`.

En la línea 15 comienza el primer ciclo `for`. No se hace la inicialización, debido a que el valor inicial de `row` fue pasado a la función. Viendo la condición se ve que este ciclo `for` se ejecuta hasta que `row` es igual a 0. Al ejecutar por primera vez la línea 15 `row` vale 8, y por lo tanto el programa continúa a la línea 17.

La línea 17 contiene el segundo enunciado `for`. Aquí el parámetro que se ha pasado, `column`, es copiado a una variable local, `col`, de tipo `int`. El valor de `col` es inicialmente 35 (el valor pasado por medio de `column`) y `column` conserva su valor original. Debido a que `col` es mayor que cero, se ejecuta la línea 18, imprimiendo una X. Luego `col` es decrementado y el ciclo continúa. Cuando `col` es 0 el ciclo `for` termina y el control pasa a la línea 20. La línea 20 hace que se imprima en la pantalla el comienzo de una nueva línea. (En el Día 7, "Entrada/salida básica", se trata a detalle la impresión). Después de moverse a una nueva línea en la pantalla el control llega al final de los enunciados del primer ciclo `for`, y por lo tanto se ejecuta la expresión de incremento que resta 1 de `row`, haciendo que sea 7. Esto regresa el control a la línea 17. Observe que el valor de `col` fue 0 la última vez que se usó. Si se hubiera usado `column` en vez de `col`, hubiera fallado la prueba de la condición, debido a que nunca sería mayor que 0. Solamente la primera línea sería impresa. Quite la inicialización de la línea 17 y cambie las dos variables `col` a `column` para ver lo que pasa realmente.

DEBE	NO DEBE

NO DEBE Poner mucho procesamiento en el enunciado `for`. Aunque se puede usar el separador coma, por lo general es más claro poner algo del funcionamiento en el cuerpo del ciclo.

DEBE Recordar usar el punto y coma si se usa un `for` con un enunciado nulo. Ponga el punto y coma en una línea aparte, o ponga un espacio entre él y el final del enunciado `for`.

```
for( contador = 0; contador < 1000; arreglo[contador] = 50) ;
/* ¡observe el espacio! */
```

El enunciado *while*

El enunciado `while`, también llamado el *ciclo while*, ejecuta un bloque de enunciados en tanto una condición específica sea cierta. El enunciado *while* tiene la siguiente forma:

```
while (condición)
    enunciado
```

Esta *condición* es cualquier expresión de C y el *enunciado* es un enunciado del C, simple o compuesto. Cuando la ejecución del programa llega al enunciado `while` suceden los siguientes eventos:

1. Es evaluada la expresión de la *condición*.
2. Si la *condición* evalúa a falso (esto es, a cero), el enunciado `while` termina, y la ejecución pasa al primer enunciado que se encuentre a continuación de los *enunciados* del `while`.
3. Si la `condición` evalúa a cierto (esto es, diferente de cero), se ejecutan los *enunciados* C del `while`.
4. La ejecución regresa al paso uno.

La operación de un enunciado `while` se encuentra diagramada en la figura 6.3.

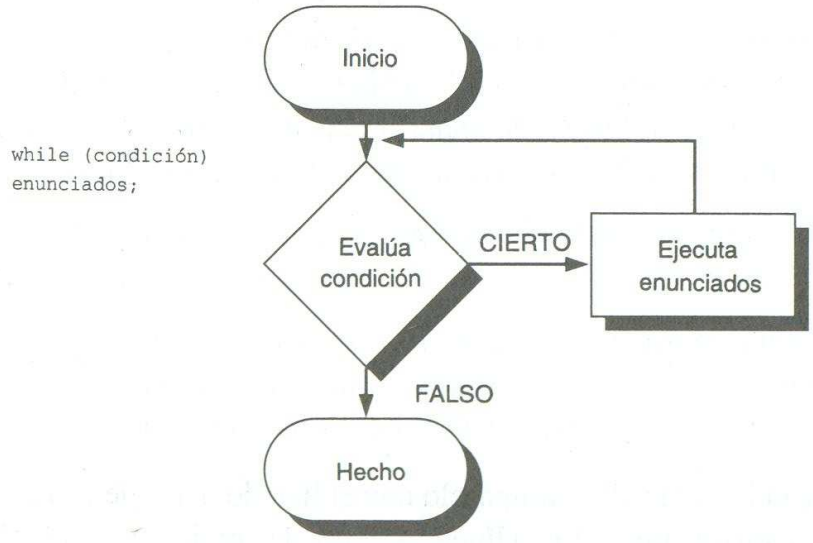


Figura 6.3. *Representación esquemática de la operación de un enunciado* `while`.

El listado 6.3 es un programa simple que usa un enunciado `while` para imprimir los números del 1 al 20. (Esta es la misma tarea que se ejecuta por un enunciado `for` en el listado 6.1.)

Captura

Listado 6.3. Demostración de un enunciado `while` simple.

```
1:  /* Demuestra un enunciado while simple */
2:
3:  #include <stdio.h>
4:
5:  int count;
6:
7:  main()
8:  {
9:      /* Imprime los números del 1 al 20 */
10:
11:     count = 1;
12:
13:     while (count <= 20)
14:     {
15:         printf("\n%d", count);
16:         count++;
17:     }
18: }
```

Salida

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Análisis

Examine el listado 6.3 y compárelo con el listado 6.1, que usa un enunciado `for` para ejecutar la misma tarea. En la línea 11 `count` es inicializado a 1. Debido a que el enunciado `while` no contiene una sección de inicialización, uno debe hacerse cargo de la inicialización de cualquier variable antes de comenzar el `while`. La línea 13 es el enunciado

while actual, y contiene el mismo enunciado condicional que el listado 6.1, count <= 20. En el ciclo while la línea 16 se ocupa de incrementar a count. ¿Qué cree que pasaría si se le olvida poner la línea 16 en el programa? El programa no sabría cuándo parar, debido a que count siempre sería 1, que es siempre menor que 20.

Tal vez se haya dado cuenta de que un enunciado while es esencialmente un enunciado for sin los componentes de inicialización e incremento. Por lo tanto,

```
for ( ; condición ; )
```

es equivalente a

```
while (condición)
```

Debido a esta equivalencia, cualquier cosa que pueda ser hecha con un enunciado for también puede ser hecha con un enunciado while. Cuando se usa un enunciado while se debe hacer primero cualquier inicialización que se necesite en un enunciado separado, y la actualización debe ser realizada por un enunciado que sea parte del ciclo while.

Cuando son requeridas la inicialización y la actualización, la mayoría de los programadores de C con experiencia prefieren usar un enunciado for en vez de un enunciado while. Esta preferencia se basa, en primer lugar, sobre la legibilidad del código fuente. Cuando se usa un enunciado for las expresiones de inicialización e incremento se ubican juntas, y son fáciles de encontrar y modificar. En un enunciado while las expresiones de inicialización y de actualización se encuentran ubicadas por separado y pueden ser menos obvias.

Sintaxis

El enunciado *while*

```
while( condición )
    enunciado(s)
```

La *condición* es cualquier expresión válida del C y por lo general es una expresión relacional. Cuando la *condición* evalúa a falso (cero) el enunciado while termina, y la ejecución pasa al primer enunciado que se encuentra a continuación de los *enunciados* del while. En caso contrario se ejecutan los *enunciados* C que se encuentran en el while.

Los *enunciados* son los enunciados C que se ejecutan siempre y cuando la *condición* sea cierta.

Un enunciado while es un enunciado de ciclo del C. El permite la ejecución repetida de un enunciado o de un bloque de enunciados en tanto la condición permanezca cierta (diferente de cero). Si la condición no es cierta cuando el comando while se ejecuta por primera vez, los *enunciados* nunca se ejecutan.

Ejemplo 1

```
int x = 0;
while( x < 10 )
```

```
{
    printf( "\nEl valor de x es %d", x );
    x++;
}
```

Ejemplo 2

```
/* Pide cifras al usuario hasta que se teclea 99 */
int núm = 0;
while( núm <= 99 )
      scanf( "%d", &núm );
```

Ejemplo 3

```
/* Permite que el usuario teclee hasta 10 valores enteros.    */
/* Los valores son guardados en un arreglo llamado valor.     */
/* Si se teclea 99 el ciclo se detiene.                       */
int valor[10];
int contador = 0;
int número;
while ( contador < 10 && número != 99)
{
     puts ( "Teclee un número, 99 para terminar ");
     scanf ( "%d", &número);
     valor[contador] = número;
     contador++;
}
```

Enunciados *while* anidados

De manera similar a los enunciados `for` e `if`, los enunciados `while` también pueden anidarse. El listado 6.4 muestra un ejemplo de enunciados `while` anidados. Aunque éste no es el mejor uso de un enunciado `while`, el ejemplo presenta algunas ideas nuevas.

Captura

Listado 6.4. Demostración de enunciados `while` anidados.

```
1:  /* Demuestra enunciados while anidados */
2:
3:  #include <stdio.h>
4:
5:  int array[5];
6:
7:  main()
8:  {
9:     int ctr = 0,
10:        nbr = 0;
11:
12:     printf( "This program prompts you to enter 5 numbers\n");
13:     printf( "Each number should be from 1 to 10\n");
14:
15:     while ( ctr < 5 )
```

```
16:     {
17:         nbr = 0;
18:         while ( nbr < 1 || nbr > 10 )
19:         {
20:             printf( "\nEnter number %d of 5: ", ctr + 1 );
21:             scanf( "%d", &nbr );
22:         }
23:
24:         array[ctr] = nbr;
25:         ctr++;
26:     }
27:
28:     for( ctr = 0; ctr < 5; ctr++ )
29:        printf( "\nValue %d is %d", ctr + 1, array[ctr] );
30: }
```

Salida

```
This program prompts you to enter 5 numbers
Each number should be from 1 to 10
Enter number 1 of 5: 3
Enter number 2 of 5: 6
Enter number 3 of 5: 3
Enter number 4 of 5: 9
Enter number 5 of 5: 2
Value 1 is 3
Value 2 is 6
Value 3 is 3
Value 4 is 9
Value 5 is 2
```

Análisis De manera similar a los listados anteriores, la línea 1 contiene un comentario con una descripción del programa, y la línea 3 contiene un enunciado `#include` para el archivo de encabezado de entrada/salida estándar. La línea 5 contiene una declaración para un arreglo (llamado array) que puede guardar cinco valores enteros. La función `main()` contiene dos variables locales adicionales, `ctr` y `nbr` (líneas 9 y 10). Observe que estas variables son inicializadas a cero al mismo tiempo que son declaradas. También observe que es usado el operador coma como separador al final de la línea 9, permitiendo que `nbr` sea declarado como `int` sin volver a poner el comando de tipo `int`. Dar declaraciones de esta forma es una práctica común de muchos programadores de C. Las líneas 11 y 12 imprimen mensajes, diciendo lo que el programa hace y lo que se espera del usuario. Las líneas 15 a 26 contienen el primer comando `while` y sus enunciados. Las líneas 18 a 22 también contienen un ciclo `while` anidado con sus propios enunciados, que son todos parte del `while` exterior.

Este ciclo exterior continúa ejecutando mientras `ctr` sea menor a 5 (línea 15). En tanto `ctr` sea menor que 5, la línea 17 pone a `nbr` a 0, con las líneas 18 a 22 (el enunciado `while` anidado) se obtiene un número en la variable `nbr`, la línea 24 pone el número en el arreglo y la línea 25 incrementa a `ctr`. Luego el ciclo vuelve a comenzar. Por lo tanto, el ciclo exterior obtiene cinco números, y pone a cada uno en array indexado por `ctr`.

El ciclo interno es un buen uso del enunciado `while`. Sólo son válidos los números del 1 al 10, por lo que mientras que el usuario teclee un número válido no tiene caso continuar con el programa. Las líneas 18 a 22 previenen la continuación. Este enunciado `while` establece que mientras el número sea menor que 1 o mientras sea mayor que 10, el programa debe imprimir un mensaje para pedir el número y luego obtenerlo.

Las línea 28 y 29 imprimen los valores que se encuentran guardados en `array`. Observe que debido a que los enunciados `while` han terminado de usar la variable `ctr`, el comando `for` puede reutilizarla. Comenzando en cero e incrementándola de uno en uno, el `for` hace ciclo cinco veces, imprimiendo el valor de `ctr` más uno (debido a que el contador comenzó en cero), e imprimiendo el valor correspondiente de `array`.

Como práctica adicional hay dos cosas que se pueden cambiar en este programa. La primera son los valores que son aceptados por el programa. En vez de 1 a 10 trate de hacer que acepte de 1 a 100. También puede cambiar la cantidad de valores que acepta. Actualmente acepta 5 números. Trate de que acepte 10.

DEBE	NO DEBE

NO DEBE Usar la siguiente convención si no es necesario.

```
while( x )
```

En vez de ello, use:

```
while( x != 0)
```

Aunque ambas funcionan, la segunda es más clara cuando se está depurando el código. Cuando se compila producen virtualmente el mismo código.

DEBE Usar el enunciado `for` en vez del enunciado `while` si necesita inicializar e incrementar dentro del ciclo. El enunciado `for` mantiene juntos los enunciados de inicialización, condición e incremento. El enunciado `while` no lo hace.

El ciclo *do...while*

La tercera construcción de ciclo del C es el ciclo `do...while`, que ejecuta un bloque de enunciados mientras una condición específica sea cierta. El ciclo `do...while` prueba la condición al final del ciclo en vez de hacerlo al principio, como es hecho por el ciclo `for` y por el ciclo `while`.

La estructura del ciclo `do...while` es la siguiente:

```
do
   enunciado
while (condición);
```

La *condición* es cualquier expresión del C, y el *enunciado* es un enunciado simple o compuesto del C.

Cuando la ejecución del programa llega a un enunciado `do...while` suceden los siguientes eventos:

1. Se ejecutan los enunciados que se encuentran en *enunciado*.
2. La *condición* es evaluada. Si es cierta, la ejecución regresa al paso 1. Si es falsa el ciclo termina.

La operación de un ciclo `do...while` se muestra esquemáticamente en la figura 6.4.

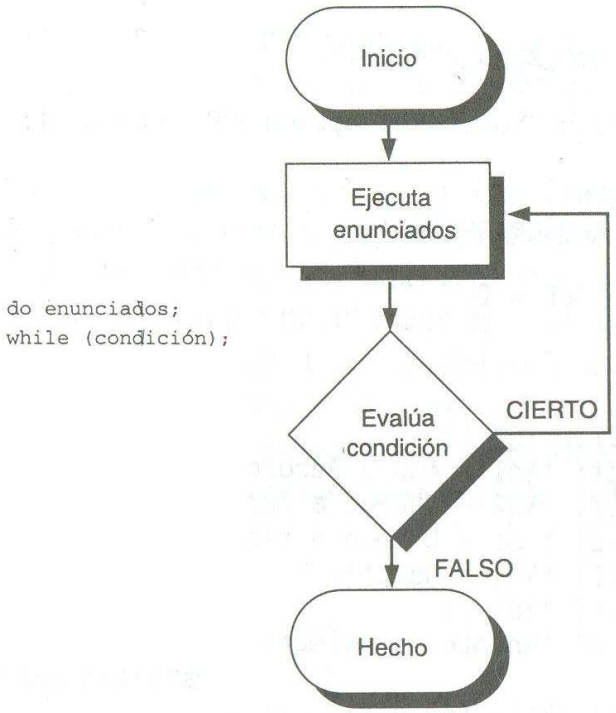


Figura 6.4. *La operación de un ciclo* `do...while`.

Los enunciados asociados con un ciclo `do...while` son siempre ejecutados por lo menos una vez. Esto se debe a que la condición es evaluada al final, en vez de al principio del ciclo. Por el contrario, los ciclos `for` y `while` evalúan la condición al inicio del ciclo y, por lo tanto, los enunciados asociados no se ejecutan si la condición es falsa al inicio.

El ciclo `do...while` es usado menos frecuentemente que los ciclos `while` y `for`. Es el más adecuado cuando los enunciados asociados con el ciclo deben ser ejecutados por lo menos una vez. Por supuesto que es posible lograr lo mismo con un ciclo `while`, asegurándose de que la condición sea cierta cuando la ejecución llegue al ciclo por primera vez. Sin embargo, probablemente sea más directo un ciclo `do...while`.

El listado 6.5 muestra un ejemplo de un ciclo `do...while`.



Listado 6.5. Demostración de un enunciado `do...while` simple.

```
1:  /* Demuestra un enunciado do...while simple */
2:
3:  #include <stdio.h>
4:
5:  int get_menu_choice( void );
6:
7:  main()
8:  {
9:      int choice;
10:
11:     choice = get_menu_choice();
12:
13:     printf( "You chose Menu Option %d", choice );
14: }
15:
16: int get_menu_choice( void )
17: {
18:     int selection = 0;
19:
20:     do
21:     {
22:         printf( "\n" );
23:         printf( "\n1 - Add a Record" );
24:         printf( "\n2 - Change a record");
25:         printf( "\n3 - Delete a record");
26:         printf( "\n4 - Quit");
27:         printf( "\n" );
28:         printf( "\nEnter a selection:" );
29:
30:         scanf( "%d", &selection );
31:
32:      }while ( selection < 1 || selection > 4 );
33:
34:      return selection;
35: }
```



```
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit

Enter a selection:8
```

```
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit

Enter a selection:4
You chose Menu Option 4
```

Análisis Este programa proporciona un menú con cuatro alternativas. El usuario selecciona una de ellas, y luego el programa imprime el número seleccionado. Algunos programas que se encuentran posteriormente en este libro usan y amplían este concepto. Por ahora deberá ser capaz de seguir la mayor parte del listado. La función `main()` (líneas 7-14) no añaden nada a lo que ya sabe. Todo lo de `main()` pudiera haber sido escrito en una sola línea.

```
printf( "You chose Menu Option %d", get_menu_option() );
```

Si se fuera a ampliar este programa y actuar sobre la selección, tal vez necesitaría el valor regresado por `get_menu_choice()`, por lo que conviene asignar el valor a una variable (como `choice`).

Las líneas 16 a 35 contienen a `get_menu_choice()`. Esta función despliega un menú en la pantalla (líneas 22 a 28) y luego obtiene una selección. Debido a que se tiene que desplegar un menú por lo menos una vez para obtener una respuesta, es adecuado usar un ciclo `do...while`. En el caso de este programa, el menú es desplegado hasta que se da una selección válida. La línea 32 contiene la parte `while` del enunciado `do...while` y valida el valor de la selección que, adecuadamente, es llamado `selection`. Si el valor dado no se encuentra entre 1 y 4, el menú se vuelve a desplegar y se le pide al usuario un nuevo valor. Cuando se da una selección válida el programa continúa a la línea 34, la cual regresa el valor en la variable selection.

Sintaxis

El enunciado *do...while*

```
do
{
    enunciado(s)
}while( condición );
```

La *condición* es cualquier expresión de C válida y, por lo general, es una expresión relacional. Cuando la *condición* evalúa a falso (cero) el enunciado `while` termina, y la ejecución pasa al primer enunciado que se encuentra a continuación del enunciado `while`. En caso contrario el programa regresa al `do` para repetir el ciclo, y se ejecutan los enunciados del C que se encuentran en *enunciado(s)*.

Los *enunciado(s)* son un enunciado simple del C o un bloque de enunciados, que se ejecutan la primera vez que se pasa por el ciclo y luego mientras la *condición* permanece cierta.

Un enunciado `do...while` es un enunciado de ciclo del C. El permite la ejecución repetida de un enunciado o de un bloque de enunciados mientras la condición se mantenga cierta (diferente a cero). A diferencia del enunciado `while`, un ciclo `do...while` ejecuta sus enunciados por lo menos una vez.

Ejemplo 1

```
/* imprime aunque la condición falle */
int x = 10;
do
{
     printf( "\nEl valor de x es %d", x );
}while( x != 10 );
```

Ejemplo 2

```
/* obtiene números hasta que el número es mayor que 99 */
int núm;
do
{
    scanf( "%d", &núm );
}while( núm <= 99 );
```

Ejemplo 3

```
/* Le permite al usuario dar hasta 10 valores enteros.    */
/* Los valores son guardados en un arreglo llamado valor. */
/* Si se da 99 el ciclo termina                           */
int valor[10];
int contador = 0;
int núm;
do
{
   puts( "Teclee un número, 99 para terminar ");
   scanf( "%d", &núm);
   valor[contador] = núm;
   contador++;
}while( contador < 10 && núm != 99);
```

Ciclos anidados

El término *ciclo anidado* se refiere a un ciclo que se encuentra dentro de otro ciclo. Ya ha visto ejemplos de algunos enunciados anidados. El C no pone limitación sobre el anidado de los ciclos, a excepción de que cada ciclo interno debe estar encerrado completamente en el ciclo externo. No se pueden tener ciclos traslapados. Por lo tanto, lo siguiente no es permitido:

```
for ( contador = 1; contador < 100; contador++)
{
    do
```

```
    {
    /* el ciclo do...while */
}  /* fin del ciclo for */
   } while (x != 0);
```

Si el ciclo do...while es puesto completamente dentro del ciclo for, no hay problemas.

```
for ( contador = 1; contador < 100; contador++)
{
    do
    {
    /* el ciclo do...while */
    }while (x != 0);
}  /* fin del ciclo for */
```

Cuando use ciclos anidados recuerde que los cambios que se hacen en el ciclo interior también pueden afectar al ciclo exterior. En el ejemplo anterior, si el ciclo interior `do...while` modifica el valor de contador, la cantidad de veces que ejecuta el ciclo `for` exterior es afectada. Sin embargo, observe que el ciclo interior puede ser independiente de cualquier variable del ciclo exterior. En este ejemplo no lo son.

El buen estilo de sangrado hace que el código con ciclos anidados sea fácil de leer. Cada nivel de ciclo debe ser indentado un paso más que el nivel anterior. Esto etiqueta claramente al código asociado con cada ciclo.

DEBE	NO DEBE
NO DEBE Tratar de traslapar ciclos. Los puede anidar pero deben estar completamente dentro del otro.	
DEBE Usar el ciclo `do...while` cuando sepa que un ciclo debe ejecutar por lo menos una vez.	

Resumen

Ahora está casi listo para comenzar a escribir programas en C reales por su cuenta.

El C tiene tres enunciados de ciclo que controlan la ejecución del programa: `for`, `while` y `do...while`. Cada una de estas construcciones le permiten al programa ejecutar un bloque de enunciados cero, una o más veces, basado en la condición de determinada variable de programa. Muchas tareas de programación se adaptan a la ejecución repetitiva que permiten estos enunciados de ciclo.

Aunque los tres pueden usarse para realizar la misma tarea, cada uno es diferente. El enunciado `for` le permite inicializar, evaluar e incrementar en un solo comando. El enunciado

`while` funciona mientras una condición es cierta. El enunciado `do...while` siempre ejecuta sus instrucciones por lo menos una vez, y continúa ejecutándolas hasta que una condición es falsa.

El anidado es poner un comando dentro de otro. El C le permite el anidado de cualquiera de sus comandos. El anidado de enunciados `if` fue demostrado en el Día 4, "Enunciados, expresiones y operadores". En este capítulo fueron anidados los enunciados `for`, `while` y `do...while`.

Preguntas y respuestas

1. ¿Cómo sé qué enunciado de control de programa debo usar, el `for`, el `while` o el `do...while`?

   Si observa los cuadros de sintaxis que se proporcionan, podrá ver que cualquiera de los tres puede ser usado para resolver un problema de ciclo. Sin embargo, cada uno de ellos tiene una forma particular de hacerlo. El enunciado `for` es el mejor cuando se sabe que se necesita inicializar e incrementar en el ciclo. Si solamente se tiene una condición que se quiere satisfacer, y no se está manejando una cantidad específica de ciclos, el `while` es una buena alternativa. Si se sabe que un juego de enunciados necesita ser ejecutado por lo menos una vez, un `do...while` puede ser lo mejor. Debido a que los tres pueden usarse para la mayoría de los problemas, lo mejor es aprenderlos todos y luego evaluar cada situación de programación para determinar cuál es el más adecuado.

2. ¿A qué tanta profundidad puedo anidar mis ciclos?

   Puede anidar tantos ciclos como desee. Sin embargo, si el programa requiere más de dos ciclos, considere el uso de una función en vez de ellos. Tal vez encuentre difícil seguir la pista por todas estas llaves, y una función es más fácil de seguir en el código.

3. ¿Puedo anidar diferentes comandos de ciclo?

   Se pueden anidar comandos `if`, `for`, `while`, `do...while` o cualquier otro. Encontrará que muchos de los programas que trate de escribir requerirán que anide por lo menos unos cuantos de éstos.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado y ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Cuál es el valor de índice del primer elemento de un arreglo?
2. ¿Cuál es la diferencia entre un enunciado `for` y un enunciado `while`?
3. ¿Cuál es la diferencia entre un enunciado `while` y un enunciado `do...while`?
4. ¿Es cierto que un enunciado `while` puede ser usado y obtener todavía los mismos resultados que al codificar un enunciado `for`?
5. ¿Qué debe recordarse cuando se anidan enunciados?
6. ¿Puede un enunciado `while` ser anidado en un enunciado `do...while`?

Ejercicios

1. Escriba una declaración para un arreglo que guarde 50 valores de tipo `long`.
2. Muestre un enunciado que asigne el valor de 123.456 al quincuagésimo elemento del arreglo del ejercicio 1.
3. ¿Cuál es el valor de x cuando se termina el siguiente enunciado?

```
for( x = 0; x < 100, x++ );
```

4. ¿Cuál es el valor de contador cuando se termina el siguiente enunciado?

```
for( contador = 2; contador < 10; contador +=3) ;
```

5. ¿Qué tantas X imprime lo siguiente?

```
for( x = 0; x < 10; x++ )

   for( y = 5; y > 0; y-- )

      puts( "X" );
```

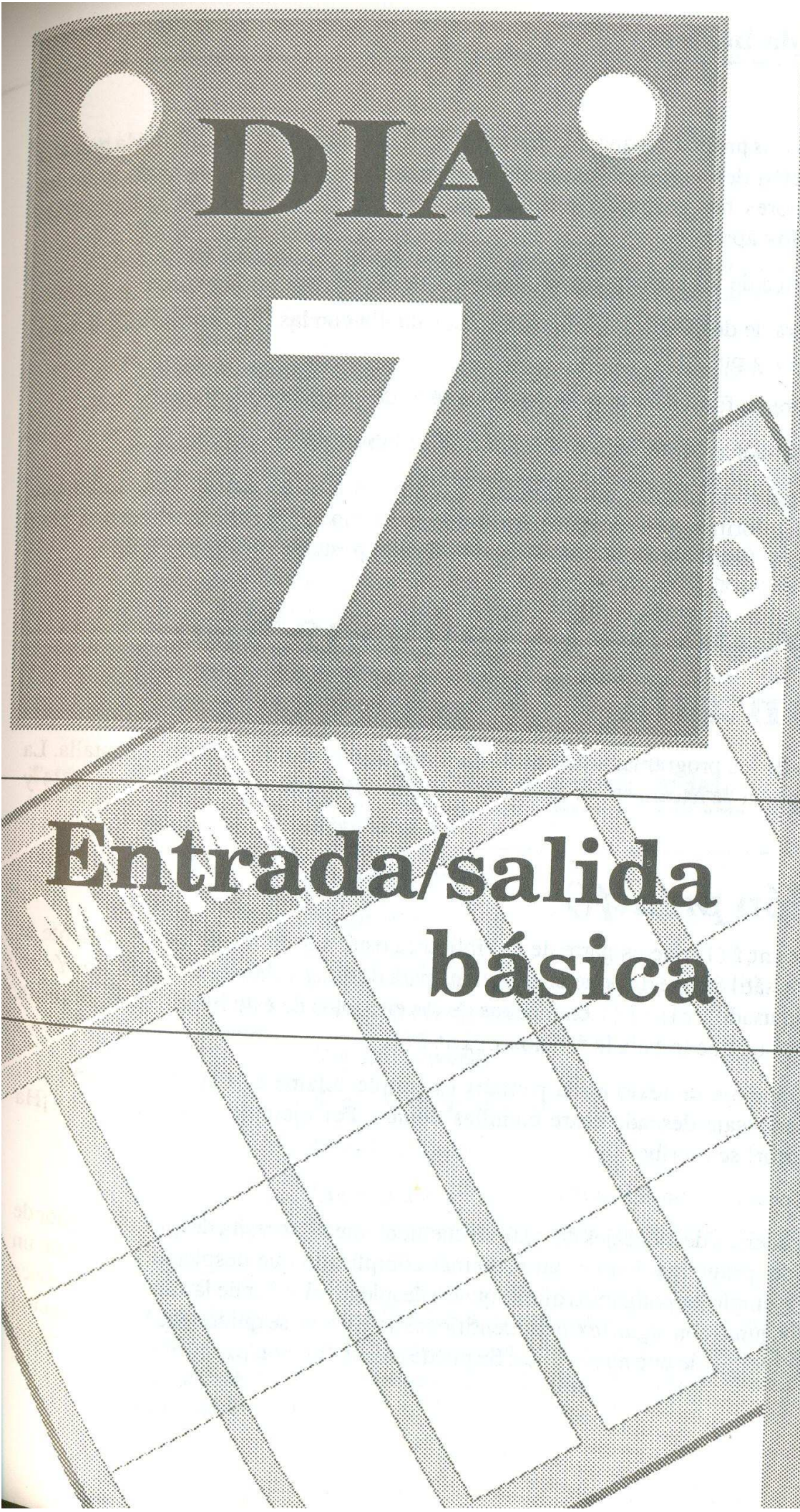
6. Escriba un enunciado `for` para contar de 1 a 100 en incrementos de 3.
7. Escriba un enunciado `while` para contar de 1 a 100 en incrementos de 3.
8. Escriba un enunciado `do...while` para contar de 1 a 100 en incrementos de 3.
9. BUSQUEDA DE ERRORES: ¿Qué está equivocado en el siguiente fragmento de código?

```
record = 0;
while( record < 100 )
{
```

```
        printf( "\nRecord %d ", record );
        printf( "\nGetting next number..." );
    }
```

10. BUSQUEDA DE ERRORES: ¿Qué está equivocado en el siguiente fragmento de código? (¡MAXVALUES no es el problema!)

```
for ( counter = 1; counter < MAXVALUES; counter++ );
   printf( "\nCounter = %d", counter );
```



Entrada/salida básica

En la mayoría de los programas que se crean se necesita desplegar información en la pantalla o leer información del teclado. Muchos de los programas que se han presentado en los capítulos anteriores han realizado estas tareas, pero tal vez no lo haya comprendido exactamente. Hoy aprenderá

- Algo acerca de los enunciados de entrada y salida del C.
- La manera de desplegar información en la pantalla con las funciones de biblioteca `printf()` y `puts()`.
- La manera de formatear la información que es desplegada en la pantalla.
- Cómo leer datos del teclado con la función de biblioteca `scanf()`.

Este capítulo no pretende dar un tratamiento completo a estos temas, sino solamente proporcionar la información suficiente para que de esta forma se pueda comenzar a escribir programas reales. Estos temas se tratan a mayor detalle posteriormente en el libro.

Desplegado de la información en la pantalla

En la mayoría de los programas usted necesitará desplegar información en la pantalla. La manera más común de hacer esto es usando las funciones de biblioteca del C, `printf()` y `puts()`.

La función *printf()*

La función `printf()`, que es parte de la biblioteca estándar del C, es probablemente la manera más versátil de que dispone un programa para desplegar datos en la pantalla. Ya ha visto que se ha usado `printf()` en muchos de los ejemplos de este libro. Ahora necesita saber la manera en que trabaja la función `printf()`.

Imprimir un mensaje de texto en la pantalla es simple. Llame a la función `printf()`, pasándole el mensaje deseado entre comillas dobles. Por ejemplo, para desplegar ¡Ha ocurrido un error! se escribe

```
printf("¡Ha ocurrido un error!");
```

Sin embargo, además de mensajes de texto frecuentemente se necesita desplegar el valor de las variables del programa. Esto es un poco más complicado que desplegar solamente un mensaje. Por ejemplo, supongamos que se quiere desplegar el valor de la variable numérica `x` en la pantalla, junto con algún texto de identificación. Es más, se quiere que la información comience al principio de una nueva línea. Se puede usar la función `printf()` de la manera siguiente:

```
printf("\nEl valor de x es %d", x);
```

y el desplegado en la pantalla, suponiendo que el valor de `x` sea 12, sería

```
El valor de x es 12
```

En este ejemplo se le pasan dos argumentos a `printf()`. El primer argumento se encuentra encerrado entre comillas dobles y es llamado el *formato*. El segundo argumento es el nombre de la variable `(x)` que contiene el valor que va a ser impreso.

Un formato de `printf()` especifica la manera en que se formatea la salida. Los tres posibles componentes de un formato son los siguientes:

- El texto literal es desplegado exactamente igual a como se dio en el formato. En el ejemplo, los caracteres que comienzan con la E (de El) y hasta, pero sin incluir el %, comprenden el texto literal.
- Una *secuencia de escape* proporciona control especial del formateo. Una secuencia de escape consiste en la diagonal inversa (\) seguida de un solo carácter. En el ejemplo anterior \n es la secuencia de escape. Este es llamado el carácter de *nueva línea* y significa "moverse al inicio de la nueva línea". Las secuencias de escape también se usan para imprimir determinados caracteres. En la tabla 7.1 se listan más secuencias de escape.

Tabla 7.1. Las secuencias de escape más frecuentemente usadas.

Secuencia	Significado
\a	Campana (alerta)
\b	Retroceso
\n	Nueva línea
\t	Tabulador horizontal
\\	Diagonal inversa
\?	Signo de interrogación
\'	Comilla simple
\"	Comilla doble

- Un *especificador de conversión* consiste en el signo de % seguido de un solo carácter. En el ejemplo, el especificador de conversión es %d. Un especificador de conversión le dice a `printf()` la manera en que debe interpretar a la(s) variable(s) que ha(n) de ser impresa(s). El %d le dice a `printf()` que interprete la variable x como un entero decimal con signo.

Las secuencias de escape de *printf()*

Veamos ahora los componentes del formato a mayor detalle. Las secuencias de escape se usan para controlar la posición de la salida moviendo el cursor de la pantalla, así como para imprimir caracteres que, de no hacerlo así, podrían tener un significado especial para `printf()`. Por ejemplo, para imprimir un solo carácter de diagonal inversa se incluye una doble diagonal inversa (\\) en el formato. La primera diagonal inversa le dice a `printf()` que la segunda diagonal inversa debe ser interpretada como un carácter literal y no como el comienzo de una secuencia de escape. En general, la diagonal inversa le dice a `printf()` que interprete el siguiente carácter de una manera especial. A continuación se presentan algunos ejemplos:

Carácter	Descripción
n	El carácter *n*
\n	Nueva línea
\"	El carácter comillas dobles
S"	El comienzo o final de un texto

La tabla 7.1 lista las secuencias de escape del C más comúnmente usadas. Una lista completa puede encontrarla en el Día 15, "Más sobre apuntadores".

El listado 7.1 es un programa que muestra algunas de las secuencias de escape frecuentemente usadas.



Listado 7.1. Uso de las secuencias de escape de `printf()`.

```
1:  /* Demostración de las secuencias de escape usadas frecuentemente */
2:
3:  #include <stdio.h>
4:
5:  #define QUIT  3
6:
7:  int  get_menu_choice( void );
8:  void print_report( void );
9:
10: main()
11: {
12:     int choice = 0;
13:
14:     while( choice != QUIT )
15:     {
16:        choice = get_menu_choice();
17:
18:        if( choice == 1 )
19:           printf( "\nBeeping the computer\a\a\a" );
20:        else
```

```
21:         {
22:             if( choice == 2 )
23:                 print_report();
24:         }
25:     }
26:     printf( "You chose to quit!" );
27: }
28:
29: int get_menu_choice( void )
30: {
31:     int selection = 0;
32:
33:     do
34:     {
35:         printf( "\n" );
36:         printf( "\n1 - Beep Computer" );
37:         printf( "\n2 - Display Report");
38:         printf( "\n3 - Quit");
39:         printf( "\n" );
40:         printf( "\nEnter a selection:" );
41:
42:         scanf( "%d", &selection );
43:
44:     }while ( selection < 1 || selection > 3 );
45:
46:     return selection;
47: }
48:
49: void print_report( void )
50: {
51:     printf( "\nSAMPLE REPORT" );
52:     printf( "\n\nSequence\tMeaning" );
53:     printf( "\n=========\t=======" );
54:     printf( "\n\\a\t\tbell (alert)" );
55:     printf( "\n\\b\t\tbackspace" );
56:     printf( "\n...\t\t...");
57: }
```

Salida

```
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence        Meaning
=========       =======
\a              bell (alert)
\b              backspace
```

```
...                    ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

Análisis El listado 7.1 parece largo en comparación con los ejemplos anteriores, pero proporciona algunas adiciones que vale la pena mencionar. El archivo de encabezado, STDIO.H, fue incluido en la línea 3 debido a que se usa `printf()` en este listado. En la línea 5 se define una constante llamada `QUIT`. En el Día 3, "Variables y constantes numéricas", se aprendió que `#define` hace que el uso de la constante `QUIT` sea equivalente a usar el valor 3. Las líneas 7 y 8 son prototipos de función. Este programa tiene dos funciones, `get_menu_choice()` y `print_report()`. `get_menu_choice()` está definida en las líneas 29 a 47. Esto es similar a la función de menú que se encuentra en el listado 6.5. Las líneas 35 y 39 contienen llamadas a `printf()` que imprimen la secuencia de escape de nueva línea. Las líneas 36, 37, 38 y 40 también usan el carácter de escape de nueva línea e imprimen texto. La línea 35 pudiera haber sido eliminada, cambiando la línea 36 para que fuera de la manera siguiente:

```
printf( "\n\n1 - Beep Computer" );
```

Sin embargo, dejar la línea 35 hace que el programa sea más fácil de leer.

Observando a la función `main()` se ve el comienzo de un ciclo `while` en la línea 14. Los enunciados del `while` se mantendrán haciendo ciclo mientras la selección no sea igual a `QUIT`. Debido a que `QUIT` es una constante se le podría haber reemplazado con el número 3. Mas sin embargo, de haber hecho esto el programa no sería tan claro. La línea 16 obtiene la variable choice, que luego es analizada en las líneas 18 a 25 en un enunciado `if`. Si el usuario escoge 1, la línea 19 imprime el carácter de nueva línea, un mensaje y luego da tres pitidos. Si el usuario selecciona 2 en el menú, la línea 23 llama a la función `print_report()`.

`print_report()` está definida en las líneas 49 a 57. Esta función simple muestra lo fácil que es usar a `printf()` y a las secuencias de escape para imprimir información formateada en la pantalla. Ya ha visto el carácter de nueva línea. Las líneas 52 a 56 también usan el carácter de escape tabulador, `\t`. Con él se alinean verticalmente las columnas del reporte. Las líneas 54 y 55 pueden parecer confusas al principio, pero si se comienza a la izquierda y se empieza a analizar yendo hacia la derecha, toman sentido. La línea 54 imprime una nueva línea (`\n`), luego una diagonal inversa (`\`) y luego la letra `a`, seguida de dos tabuladores (`\t\t`). La línea termina con un texto descriptivo (`bell (alert)`). La línea 55 sigue el mismo formato.

Este programa imprime las primeras dos líneas de la tabla 7.1, junto con un título de reporte y encabezados de columnas. En el ejercicio nueve se completará este programa, haciéndolo que imprima el resto de la tabla.

Los especificadores de conversión de *printf()*

El formato debe contener un especificador de conversión para cada variable a ser impresa. Luego `printf()` despliega cada variable, como lo indica su especificador de conversión correspondiente. Se aprenderá más acerca de este proceso en el Día 15, "Más sobre apuntadores". Por ahora, asegúrese de usar el especificador de conversión que corresponda al tipo de la variable que vaya a imprimirse.

¿Qué significa esto exactamente? Si va a imprimir una variable que es un *entero decimal con signo* (`tipos int y long`), use el especificador de conversión `%d`. Para un *entero decimal sin signo* (`tipos unsigned int y unsigned long`), use `%u`. Para una *variable de punto flotante* (`tipos float y double`), use el especificador `%f`. Los especificadores de conversión que se necesitan más frecuentemente se encuentran listados en la tabla 7.2.

Tabla 7.2. Los especificadores de conversión más comúnmente necesitados.

Especificador	Significado
%c	Un solo carácter
%d	Entero decimal con signo
%f	Número decimal de punto flotante
%s	Cadena de caracteres
%u	Entero decimal sin signo

El texto literal de un especificador de formato es cualquier cosa que no califica como secuencia de escape o especificador de conversión. El texto literal es impreso simplemente como es, incluyendo todos los espacios.

¿Qué hay acerca de la impresión de valores de más de una variable? Un solo enunciado `printf()` puede imprimir una cantidad ilimitada de variables, pero el formato debe contener un especificador de conversión para cada variable. Los especificadores de conversión son apareados con las variables en orden de izquierda a derecha. Si se escribe

```
printf("Tasa = %f, cantidad = %d", tasa, cantidad);
```

la variable tasa es apareada con el especificador `%f` y la variable cantidad es apareada con el especificador `%d`. Las posiciones de los especificadores de conversión en el formato determinan la posición de la salida. Si hay más variables pasadas a la función `printf()` que especificadores de conversión, las variables que no hacen par no se imprimen. Si hay más especificadores de conversión que variables, los especificadores sin aparear imprimen "basura".

No se está limitado a imprimir los valores de variables con `printf()`. Los argumentos pueden ser cualquier expresión del C válida. Por ejemplo, para imprimir la suma de `x` y `y`, se podría escribir:

```
z = x + y;
printf("%d", z);
```

También se podría escribir:

```
printf("%d", x + y);
```

Cualquier programa que use a `printf()` debe incluir el archivo de encabezado STDIO.H. El listado 7.2 muestra el uso de `printf()`. El Día 15, "Más sobre apuntadores", da mayores detalles sobre `printf()`.

Captura

Listado 7.2. Uso de `printf()` para desplegar valores numéricos.

```
1:  /* Demostración del uso de printf() para desplegar valores numéricos. */
2:
3:  #include <stdio.h>
4:
5:  int a = 2, b = 10, c = 50;
6:  float f = 1.05, g = 25.5, h = -0.1;
7:
8:  main()
9:  {
10:     printf("\nDecimal values without tabs: %d %d %d", a, b, c);
11:     printf("\nDecimal values with tabs: \t%d \t%d \t%d", a, b, c);
12:
13:     printf("\nThree floats on 1 line: \t%f\t%f\t%f", f, g, h);
14:     printf("\nThree floats on 3 lines: \n\t%f\n\t%f\n\t%f", f,
            g,h);
15:
16:     printf("\nThe rate is %f%%", f);
17:     printf("\nThe result of %f/%f = %f", g, f, g / f);
18: }
```

Salida

```
Decimal values without tabs: 2 10 50
Decimal values with tabs:       2       10      50
Three floats on 1 line: 1.050000 25.500000 -0.100000
Three floats on 3 lines:
1.050000
25.500000
-0.100000
The rate is 1.050000%
The result of 25.500000/1.050000 = 24.285715
```

Análisis

El listado 7.2 imprime seis líneas de información. Las líneas 10 y 11 imprimen cada una tres valores decimales, `a`, `b` y `c`. La línea 10 los imprime sin tabuladores y la línea 11 los imprime con tabuladores. Las líneas 13 y 14 imprimen cada una tres variables

float, f, g y h. La línea 14 las imprime en una línea y la línea 13 las imprime en tres líneas. La línea 16 imprime una variable float, f, seguida por un signo de porcentaje. Debido a que el signo de porcentaje normalmente es un mensaje para imprimir una variable, se debe poner dos de ellos juntos para imprimir un solo signo de porcentaje. Esto es exactamente similar al carácter de escape de diagonal inversa. La línea 17 muestra un concepto final. Cuando se imprimen valores en los especificadores de conversión no se tienen que usar variables. También se pueden usar expresiones, como g / f, o hasta constantes.

DEBE	NO DEBE

NO DEBE Tratar de poner varias líneas de texto en un solo enunciado printf(). La mayoría de las veces es más claro imprimir varias líneas con varios enunciados de impresión que hacerlo con uno solo y con varios caracteres de escape de nueva línea (\n).

NO DEBE Olvidar usar el carácter de escape de nueva línea cuando imprima varias líneas de información en enunciados printf() separados.

Sintaxis

La función *printf()*

```
#include <stdio.h>
printf( formato[,argumentos,...]);
```

printf() es una función que acepta una serie de *argumentos,* aplicándose cada uno de ellos a un *especificador de conversión* en el formato dado. printf() imprime la información formateada en el dispositivo de salida estándar que, por lo general, es la pantalla. Cuando se usa printf() se necesita incluir el archivo de encabezado de entrada/salida estándar, STDIO.H.

El formato es requerido. Sin embargo, los argumentos son opcionales. Para cada argumento debe haber un especificador de conversión. La tabla 7.2 lista los especificadores de conversión necesarios más comunes.

El formato también puede contener secuencias de escape. La tabla 7.1 lista las secuencias de escape más frecuentemente usadas.

Los siguientes son ejemplos de llamadas a printf() y su salida:

Ejemplo 1

```
#include <stdio.h>
main()
{
   printf( "¡Este es un ejemplo de algo impreso!");
}
```

Despliega

```
¡Este es un ejemplo de algo impreso!
```

Ejemplo 2

```
printf( "Esto imprime un carácter, %c\nun número, %d\nun punto flotante, %f",
'z', 123, 456.789 );
```

Despliega

```
Esto imprime un carácter, z
un número, 123
un punto flotante, 456.789
```

Desplegado de mensajes con *puts()*

La función `puts()` también puede ser usada para desplegar mensajes de texto en la pantalla, pero no puede desplegar variables numéricas. `puts()` toma un sola cadena como su argumento y la despliega, añadiendo automáticamente una nueva línea al final. Por ejemplo, el enunciado

```
puts("Hola.");
```

ejecuta la misma acción que

```
printf("Hola.\n");
```

Se pueden incluir secuencias de escape (incluyendo `\n`) en una cadena que se le pasa a `puts()`. Tienen el mismo efecto que cuando se usan con `printf()` (véase la tabla 7.1).

Cualquier programa que use `puts()` debe incluir al archivo de encabezado STDIO.H. Tome en cuenta de que STDIO.H debe ser incluido solamente una vez en cualquier programa

DEBE	NO DEBE

DEBE Usar la función `puts()` en vez de la función `printf()` cada vez que se quiera imprimir texto, pero que no se necesite imprimir variables.

NO DEBE Tratar de usar especificadores de conversión con los enunciados `puts()`.

Sintaxis

La función *puts()*

```
#include <stdio.h>
puts( cadena );
```

`puts()` es una función que copia una cadena al dispositivo de salida estándar, que por lo general es la pantalla. Cuando use `puts()` incluya el archivo de encabezado de entrada/salida estándar (STDIO.H). `puts()` también añade un carácter de nueva línea al final de

la cadena que es impresa. La cadena puede contener secuencias de escape. La tabla 7.1, que se mostró anteriormente, lista las secuencias de escape más frecuentemente usadas.

Los siguientes son ejemplos de llamadas a `puts()` y su salida:

Ejemplo 1

```
puts( "¡Esto es impreso con la función puts()!");
```

Despliega

```
¡Esto es impreso con la función puts()!
```

Ejemplo 2

```
puts( "Esto se imprime en la primera línea. \nEsto se imprime en la
segunda línea.");
puts( "Esto se imprime en la tercera línea.");
puts( "¡Si se usara printf(), las cuatro líneas estarían en dos
líneas!");
```

Despliega

```
Esto se imprime en la primera línea.
Esto se imprime en la segunda línea.
Esto se imprime en la tercera línea.
¡Si se usara printf(), las cuatro líneas estarían en dos líneas!
```

Entrada de datos numéricos con *scanf()*

Así como muchos programas necesitan la salida de datos a pantalla, también necesitan la entrada de datos del teclado. La manera más flexible para que el programa pueda leer datos numéricos del teclado es el uso de la función de biblioteca `scanf()`.

La función `scanf()` lee datos del teclado de acuerdo con un formato especificado, y asigna los datos de entrada a una o más variables del programa. De manera similar a `printf()`, `scanf()` usa un formato para describir el formato de la entrada. El formato utiliza los mismos especificadores de conversión que la función `printf()`. Por ejemplo, el enunciado

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

lee un entero decimal del teclado y lo asigna a la variable entera `x`. De manera similar el enunciado

```
scanf("%f", &tasa);
```

lee un valor de punto flotante del teclado y lo asigna a la variable tasa.

¿Qué hace el & antes del nombre de la variable? El símbolo & es el operador de *dirección de* C, que es explicado a detalle en el Día 9, "Apuntadores". Por ahora, todo lo que necesita recordar es que `scanf()` requiere el símbolo & antes de cada nombre de variable numérica en su lista de argumentos (a menos que la variable sea un *apuntador,* lo que también se explica en el Día 9).

Un solo `scanf()` puede aceptar la entrada de más de un valor si se incluyen varios especificadores de conversión en el formato y varias variables (nuevamente cada una de ellas precedida por un & en la lista de argumentos). El enunciado

```
scanf("%d %f", &x, &tasa);
```

da entrada a un valor entero y a un valor de punto flotante, y los asigna a las variables `x` y `tasa`, respectivamente. Cuando se da entrada a diversas variables, `scanf()` usa al espacio en blanco para separar la entrada en campos. El espacio en blanco pueden ser blancos, tabuladores o nuevas líneas. Cada especificador de conversión en el formato del `scanf()` es apareado con un campo de entrada. El final de cada campo de entrada es identificado por el espacio en blanco.

Esto le da bastante flexibilidad. En respuesta al `scanf()` anterior se podría teclear

```
10 12.45
```

También se podría teclear

```
10                    12.45
```

o

```
10
12.45
```

Mientras haya algún espacio en blanco entre los valores, `scanf()` puede asignar cada valor a su variable.

De manera similar a las otras funciones tratadas en este capítulo, los programas que usan a `scanf()` deben incluir al archivo de encabezado STDIO.H. Aunque el listado 7.3 le da un ejemplo sobre el uso de `scanf()`, una descripción más completa se presenta en el Día 15, "Más sobre apuntadores".

Captura

Listado 7.3. Uso de `scanf()` para obtener valores numéricos.

```
1:  /* Demostración del uso de scanf() */
2:
3:  #include <stdio.h>
4:
5:  #define QUIT 4
6:
7:  int get_menu_choice( void );
8:
```

```
9:  main()
10: {
11:      int   choice    = 0;
12:      int   int_var   = 0;
13:      float float_var = 0.0;
14:      unsigned unsigned_var = 0;
15:
16:      while( choice != QUIT )
17:      {
18:          choice = get_menu_choice();
19:
20:          if( choice == 1 )
21:          {
22:             puts( "\nEnter a signed decimal integer (i.e. -123)" );
23:             scanf( "%d", &int_var );
24:          }
25:          if ( choice == 2 )
26:          {
27:              puts( "\nEnter a decimal floating-point number
                       (i.e. 1.23)" );
28:              scanf( "%f", &float_var );
29:          }
30:          if ( choice == 3 )
31:          {
32:               puts( "\nEnter an unsigned decimal integer \
                      (i.e. 123)" );
33:               scanf( "%u", &unsigned_var );
34:          }
35:      }
36:      printf( "\nYour values are: int: %d  float: %f  unsigned: %u ",
37:                                  int_var, float_var, unsigned_var );
38: }
39:
40: int get_menu_choice( void )
41: {
42:      int selection = 0;
43:
44:      do
45:      {
46:          puts( "\n1 - Get a signed decimal integer" );
47:          puts( "2 - Get a decimal floating-point number" );
48:          puts( "3 - Get an unsigned decimal integer" );
49:          puts( "4 - Quit" );
50:          puts( "\nEnter a selection:" );
51:
52:          scanf( "%d", &selection );
53:
54:      }while ( selection < 1 || selection > 4 );
55:
56:      return selection;
57: }
```

Salida

```
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123  float: 1231.123047 unsigned: 321
```

Análisis

El listado 7.3 usa los mismos conceptos de menú que se usaron en el listado 7.1. Las diferencias en `get_menu_choice()` (líneas 40 a 57) son menores, pero deben ser mencionadas. En primer lugar se usa `puts()` en vez de `printf()`. Debido a que no se imprimen variables no hay necesidad de usar `printf()`. Debido a que se usa `puts()` se han quitado los caracteres de escape de nueva línea de las líneas 47 a 49. La línea 54 también fue cambiada para permitir valores del 1 al 4, debido a que ahora hay cuatro opciones de menú. Observe que la línea 52 no ha cambiado; sin embargo, ahora debe entenderla mejor. `scanf()` obtiene un valor decimal y lo pone en la variable de selección. En la línea 56 la función regresa `selection` al programa que la llama.

Los listados 7.1 y 7.3 usan la misma estructura de `main()`. Un enunciado `if` evalúa a `choice`, que es el valor de retorno de `get_menu_choice()`. Basándose en el valor de `choice`, el programa imprime un mensaje, pide que se teclee un número y lee el valor con `scanf()`. Observe la diferencia entre las líneas 22, 28 y 32. Cada una está puesta para obtener un tipo diferente de variable. Las líneas 12 a 14 declaran variables para los tipos apropiados.

Cuando el usuario selecciona quit, el programa imprime el último número tecleado para cada uno de los tres tipos. Si el usuario no teclea un valor se imprime 0, debido a que las líneas 12 y 13 inicializan a los tres tipos. Una nota final sobre las líneas 20 a 34: Los enunciados `if` que se usan aquí no están bien estructurados. Si usted piensa que una estructura `if...else` hubiera sido mejor, está en lo correcto. En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se presenta un nuevo enunciado de control, `switch`. Este enunciado hubiera sido la mejor opción.

DEBE	NO DEBE
NO DEBE Olvidar incluir al operador *de dirección* de (&) cuando use variables en `scanf()`.	
DEBE Usar `printf()` o `puts()` junto con `scanf()`. Use las funciones de impresión para desplegar un mensaje donde pida los datos que quiere obtener con `scanf()`.	

Sintaxis

La función *scanf()*

```
#include <stdio.h>
scanf( formato[,argumentos,...]);
```

`scanf()` es una función que usa un *especificador de conversión* en un `formato` dado para poner valores en las variables de argumento. Los argumentos deben ser las direcciones de las variables, en vez de las variables actuales. Para las variables numéricas se puede pasar la dirección, añadiendo el operador de *dirección de* (&) al inicio del nombre de variable. Cuando se use `scanf()` se debe incluir el archivo de encabezado STDIO.H.

`scanf()` lee campos de entrada del flujo de entrada estándar que, por lo general, es el teclado. Pone cada uno de estos campos leídos en un argumento. Cuando pone la información la convierte al formato del especificador de conversión correspondiente que se encuentra en el formato. Para cada argumento debe haber un especificador de conversión. La tabla 7.2, mostrada anteriormente, lista los especificadores de conversión necesarios más comunes.

Los siguientes son ejemplos de llamadas a `scanf()`:

Ejemplo 1

```
int x, y, z;
scanf( "%d %d %d", &x, &y, &z);
```

Ejemplo 2

```
#include <stdio.h>
main()
```

```
{
    float y;
    int x;

    puts( "Teclee un punto flotante, luego un entero" );
    scanf( "%f %d", &y, &x);
    printf( "\nSe tecleó %f y %d ", y, x );
}
```

Resumen

Al terminar este capítulo ya está listo para escribir sus propios programas en C. Combinando las funciones `printf()`, `puts()` y `scanf()` y el control de programación que se aprendió en los capítulos anteriores, se tienen las herramientas necesarias para escribir programas simples.

El desplegado en pantalla se ejecuta con las funciones `printf()` y `puts()`. La función `puts()` puede desplegar solamente mensajes de texto, mientras que `printf()` puede desplegar mensajes de texto y variables. Ambas funciones usan secuencias de escape para los caracteres especiales y control de la impresión.

La función `scanf()` lee uno o más valores numéricos del teclado e interpreta a cada uno de acuerdo con un especificador de conversión. Cada valor es asignado a una variable de programa.

Preguntas y respuestas

1. ¿Por qué debo usar `puts()` si `printf()` hace todo lo que hace `puts()` y más?

   Debido a que `printf()` hace más, tiene una sobrecarga adicional. Cuando se esté tratando de escribir un programa pequeño eficiente o cuando los programas se hacen grandes y los recursos son valiosos, se querrá tomar ventaja de la pequeña sobrecarga de `puts()`. Por lo general, use el recurso disponible más sencillo.

2. ¿Por qué necesito incluir STDIO.H cuando uso `printf()`, `puts()` o `scanf()`?

   STDIO.H contiene los prototipos para las funciones de entrada/salida estándares. `printf()`, `puts()` y `scanf()` son tres de estas funciones estándares. Trate de ejecutar un programa sin el archivo de encabezado STDIO.H y vea los errores y avisos que obtiene.

3. ¿Qué pasa si no pongo el operador (&) en una variable de `scanf()`?

Este es un error fácil de cometer. Pueden producirse resultados impredecibles si olvida la dirección del operador. Cuando lea acerca de los apuntadores en los Días 9 y 13, "Apuntadores" y "Más sobre el control de programa", comprenderá mejor esto. Por ahora, simplemente sepa que si omite la dirección del operador, `scanf()` no pone la información tecleada en la variable sino en algún otro lugar de la memoria. Esto puede no producir efecto visible alguno, o bien, por lo contrario, dar lugar a cualquier efecto, incluyendo el de que su computadora se trabe y tenga que rearrancarla.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado y ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Cuál es la diferencia entre `puts()` y `printf()`?
2. ¿Qué archivo de encabezado debe ser incluido cuando se usa a `printf()`?
3. ¿Qué hacen las siguientes secuencias de escape?

   a. `\\`

   b. `\b`

   c. `\n`

   d. `\t`

   e. `\a`

4. ¿Qué especificadores de conversión deben ser usados para imprimir lo siguiente?

   a. una cadena.

   b. un entero decimal con signo.

   c. un número decimal de punto flotante.

5. ¿Cuál es la diferencia entre usar cada uno de los siguientes en el texto literal de `puts()`?

   a. `b`

   b. `\b`

   c. `\`

   d. `\\`

Ejercicios



Nota: Comenzando con este capítulo, algunos de estos ejercicios le piden que escriba programas completos que ejecuten una tarea particular. Siempre hay más de una manera de hacer las cosas en C, por lo que las respuestas que se proporcionan al final del libro no deben ser interpretadas como las únicas correctas. Si usted puede escribir su propio código que ejecute lo que se desea, ¡está muy bien! Si tiene problemas, vea el ejemplo de respuesta como ayuda. Estas respuestas son presentadas con un mínimo de comentarios. ¡Es buena práctica para usted imaginar la manera en que trabajan!

1. Escriba un enunciado con `printf()` y otro con `puts()` que inicien una nueva línea.
2. Escriba un enunciado `scanf()` que pueda ser usado para obtener un carácter, un entero decimal sin signo y otro carácter solo.
3. Escriba los enunciados para obtener un valor entero e imprimirlo.
4. Modifique el ejercicio 3 para que acepte solamente valores pares (2, 4, 6, etcétera).
5. Modifique el ejercicio 4 para que regrese valores hasta que se teclee el número 99 o hasta que se hayan tecleado seis valores pares. Guarde los números en un arreglo. (Consejo: ¡Se necesita un ciclo!)
6. Cambie el ejercicio 5 en un programa ejecutable. Añada una función que imprima los valores del arreglo, separados con tabuladores, en una sola línea. (Sólo imprima los valores que fueron guardados en el arreglo.)
7. BUSQUEDA DE ERRORES: Encuentre el error en el siguiente fragmento de código:

```
printf( "Jack said, "Peter Piper picked a peck of pickled
peppers."");
```

8. BUSQUEDA DE ERRORES: Encuentre los errores en el siguiente programa:

```
int get_1_or_2( void )
{
    int answer = 0;

    while( answer < 1 || answer > 2 )
    {
```

```
        printf(Enter 1 for Yes, 2 for No);
        scanf( "%f", answer );
    }
    return answer;
}
```

9. Usando el listado 7.1 complete la función `print_report()` para que imprima el resto de la tabla 7.1.

10. Escriba un programa que reciba del teclado dos valores de punto flotante y luego despliegue su producto.

11. Escriba un programa que reciba del teclado 10 valores enteros y luego despliegue su suma.

12. Escriba un programa que reciba del teclado enteros, guardándolos en un arreglo. La entrada debe parar cuando se teclee un cero o cuando se llegue al fin del arreglo. Luego, encuentre y despliegue los valores mayor y menor del arreglo. (Nota: Este es un problema difícil debido a que todavía no han sido tratados los arreglos completamente. Si tiene dificultades, trate de resolverlos nuevamente después de haber leído el Día 8, "Arreglos numéricos".)

Revisión de la semana 1

Después de que haya acabado su primera semana de aprendizaje sobre la manera de programar en C, deberá sentirse a gusto tecleando programas y usando el compilador y el editor. El siguiente programa reúne muchos de los temas de la semana anterior.

Nota: Los números a la izquierda de los números de renglón indican el capítulo donde se trata el concepto presentado en esa línea. Si no entiende bien la línea, vea el capítulo a que se hace referencia para mayor información.

Listado R1.1. Listado de revisión de la semana 1.

```
CH02    1:  /* Nombre del programa: week1.c                                    */
        2:  /*                      programa para teclear las edades e ingresos */
        3:  /*                      de hasta 100 gentes. El programa imprime un */
        4:  /*                      reporte basado en las cantidades tecleadas. */
        5:  /*-------------------------------------*/
        6:  /*-----------*/
        7:  /* Archivos de inclusión          */
        8:  /*-----------*/
CH02    9:  #include <stdio.h>
        10:
CH02    11: /*-----------*/
        12: /* Constantes definidas          */
        13:  /*-----------*/
        14:
CH03    15: #define MAX   100
        16: #define YES   1
        17: #define NO    0
        18:
CH02    19:  /*-----------*/
        20: /* Variables                     */
        21:  /*-----------*/
        22:
CH03    23: long   income[MAX]; /* Para guardar los ingresos     */
        24: int    month[MAX], day[MAX], year[MAX]; /* Para guardar fechas de
                   nacimiento */
        25: int     x, y, ctr;   /* Para contadores                */
        26: int     cont;  /* Para control de programa       */
        27: long   month_total, grand_total;  /* Para totales                */
        28:
CH02    29:  /*-----------*/
        30: /* Prototipos de función         */
        31:  /*-----------*/
        32:
CH05    33: void main(void)
        34: int display_instructions(void);
        35: void get_data(void);
        36: void display_report();
        37: int continue_function(void);
        38:
        39:  /*-----------*/
        40: /* Inicio del programa           */
```

```
/*————————*/

CH02 void main()
CH05 {
CH04     cont = display_instructions();

CH05     if ( cont == YES )
CH05     {
CH04         get_data();
CH07         display_report();
CH02     }
CH05     else
CH07         printf( "\nProgram Aborted by User!\n\n");
CH02 }
CH05 /*————————————————————————*
CH07 * Función: display_instructions()                                  *
CH02 * Objetivo: esta función despliega información sobre la manera     *
CH05 *           de usar este programa y pide al usuario teclear 0      *
CH04 *           para terminar o 1 para continuar.                      *
CH06 * Regresa: NO - si el usuario teclea 0                             *
CH04 *   YES - si el usuario teclea cualquier número diferente de 0     *
CH04 *————————————————————————*/

CH05 int display_instructions( void )
CH07 {
CH07     printf("\n\n");
CH07     printf("\nThis program enables you to enter up to 99 \
                people\'s ");
CH07     printf("\nincomes and birthdays.  It then prints the \
                incomes by");
CH02     printf("\nmonth along with the overall income and \
                overall  average.");
CH05     printf("\n");

CH05     cont = continue_function();

CH05     return( cont );
CH02 }
CH02 /*————————————————————————*
CH02  * Función: get_data() *
```

Listado R1.1. continuación

```
78:   * Objetivo: esta función obtiene datos del usuario.  *
79:   *          continúa pidiendo datos hasta que se hayan tecleado      *
80:   *          100 gentes o hasta que el usuario teclee 0 en el mes     *
81:   * Regresa: nada  *
82:   * Nota:    Se permite que se teclee 0/0/0 para la fecha de nacimiento *
83:   *          en caso de que el usuario no esté seguro. También permite  *
84:   *          que todos los meses sean de 31 días.                       *
85:   *-------------------------------------*/
86:
CH05  87:  void get_data(void)
88:  {
CH06  89:     for ( cont = YES, ctr = 0; ctr < MAX && cont == YES; ctr++ )
90:     {
CH07  91:        printf("\nEnter information for Person %d.", ctr+1 );
92:        printf("\n\tEnter Birthday:");
93:
CH06  94:        do
95:        {
CH07  96:           printf("\n\tMonth (0 - 12): ");
CH07  97:           scanf("%d", &month[ctr]);
CH06  98:    }while (month[ctr] < 0 || month[ctr] > 12 );
99:
CH06 100:        do
101:        {
CH07 102:           printf("\n\tDay (0 - 31): ");
CH07 103:           scanf("%d", &day[ctr]);
CH06 104:        }while ( day[ctr] <  0 || day[ctr] > 31 );
105:
CH06 106:        do
107:        {
CH07 108:           printf("\n\tYear (0 - 1994): ");
CH07 109:           scanf("%d", &year[ctr]);
CH06 110:        }while ( year[ctr] < 0 || year[ctr] > 1994 );
111:
CH07 112:        printf("\nEnter Yearly Income (whole dollars): ");
CH07 113:        scanf("%ld", &income[ctr]);
114:
CH05 115:        cont = continue_function();
116:     }
CH07 117:     /* ctr es igual a la cantidad de gente que se ha tecleado.   */
```

```
118: }
CH02 119: /*————————————————————————————*
120:  * Función: display_report()                                    *
121:  * Objetivo: esta función despliega un reporte en la pantalla   *
122:  * Regresa: nada                                                *
123:  * Notas: Se podría haber impreso más información.              *
124:  *————————————————————————————*/
125:
CH05 126: void display_report()
127: {
CH04 128:    grand_total = 0;
CH07 129:    printf("\n\n\n");                /* se salta unas cuantas líneas */
130:    printf("\n        SALARY SUMMARY");
131:    printf("\n        ==============");
132:
CH06 133:    for( x = 0; x <= 12; x++ )  /* para cada mes, incluyendo 0 */
134:     {
135:        month_total = 0;
CH04 136:        for( y = 0; y < ctr; y++ )
CH06 137:        {
138:            if( month[y] == x )
CH04 139:                month_total += income[y];
CH04 140:        }
141:        printf("\nTotal for month %d is %ld", x, month_total);
CH07 142:        grand_total += month_total;
CH04 143:     }
144:     printf("\n\nReport totals:");
CH07 145:     printf("\nTotal Income is %ld", grand_total);
146:     printf("\nAverage Income is %ld", grand_total/ctr );
147:
148:     printf("\n\n* * * End of Report * * *");
149: }
CH02 150:  /*————————————————————————————*/
151:  * Función: continue_function() *
152:  * Objetivo: esta función le pregunta al usuario si quiere continuar. *
153:  * Regresa: YES - si el usuario desea continuar                  *
154:  *          NO  - si el usuario desea terminar                   *
155: *————————————————————————————*/
156:
CH05 157: int continue_function( void )
158: {
```

Listado R1.1. continuación

```
CH07    159:     printf("\n\nDo you wish to continue? (0=NO/1=YES): ");
        160:     scanf( "%d", &x );
        161:
CH06    162:     while( x < 0 || x > 1 )
        163:     {
CH07    164:         printf("\n%d is invalid!", x);
        165:         printf("\nPlease enter 0 to Quit or 1 to Continue: ");
        166:         scanf("%d", &x);
        167:     }
CH04    168:     if(x == 0)
CH05    169:         return(NO);
CH04    170:     else
CH05    171:         return(YES);
        172: }
```

Después de haber completado los cuestionarios y los ejercicios del Día 1, "Comienzo", y del 2, "Los componentes de un programa C", deberá ser capaz de teclear y compilar este programa. Este programa contiene más comentarios que otros listados en este libro. Estos comentarios son típicos de un programa C real. En particular, observe los comentarios que se encuentran al principio del programa y antes de cada función principal. Los comentarios de las líneas 1 a la 5 contienen una descripción del programa completo, incluyendo el nombre del programa. Algunos programadores también incluyen información como el nombre del autor del programa, el compilador usado, su número de versión, las bibliotecas enlazadas en el programa y la fecha en que el programa fue creado. Los comentarios que se encuentran antes de cada función describen el objetivo de la función, los valores de retorno posibles, las convenciones de llamado de la función y cualquier otra cosa que se realciona específicamente con esa función.

Los comentarios de las líneas 1 a 5 especifican que se puede dar información en este programa para un máximo de 100 gentes. Antes de que pueda teclear los datos el programa llama a la función `display_instructions()` (línea 45). Esta función despliega instrucciones sobre el uso del programa y le pregunta si quiere continuar o terminar. En las líneas 66 a 70 se puede ver que esta función usa la función `printf()`, que se vio en el Día 7, "Entrada/salida básica", para desplegar las instrucciones.

La función `continue_function()`, que se encuentra en las líneas 157 a 172, usa algunas de las características tratadas al final de la semana. La función pregunta si se quiere continuar (línea 159). Usando el enunciado de control `while` del Día 6, "Control básico del programa", la función verifica que la respuesta tecleada sea 0 o 1. Mientras

la respuesta no sea alguno de estos dos valores la función se mantiene pidiendo una respuesta. Una vez que el programa recibe la respuesta adecuada, un enunciado `if...else` (Día 4, "Enunciados, expresiones y operadores") regresa una constante de `YES` o `NO`.

La parte medular de este programa se encuentra en dos funciones: `get_data()` y `display_report()`. La función `get_data()` le pide que teclee datos, poniendo la información en los arreglos declarados cerca del principio del programa. Usando un enunciado `for`, en la línea 89, se pide que se tecleen datos hasta que `cont` no sea igual a la constante definida `YES` (regresada de la función `continue_function()`) o el contador, `ctr`, sea mayor o igual al número máximo de elementos del arreglo, `MAX`. Este programa revisa la información tecleada, para asegurarse de que es la adecuada. Por ejemplo, las líneas 94 a 98 le piden que teclee un mes. Los únicos valores que acepta el programa son del 0 al 12. Si se teclea un número mayor de 12 el programa vuelve a pedir el mes. La línea 115 llama a la función `continue_function()` para revisar si se quiere continuar añadiendo datos.

Cuando se responde a la función de continuar con un 0 o cuando el número máximo de juegos de información es tecleado (de acuerdo al ajuste de `MAX`), el programa regresa a la línea 50 de `main()`, donde llama a `display_report()`. La función `display_report()`, que se encuentra en las líneas 119 a 149, imprime un reporte en la pantalla. Este reporte usa un ciclo `for` anidado, para dar el total de ingresos para cada mes y un gran total para todos los meses. Este reporte puede parecer complicado. Si es así, revise el Día 6, "Control básico del programa", sobre la explicación de los enunciados anidados. Muchos de los reportes que se crean cuando uno es programador son más complicados que éste.

Este programa usa lo que se ha aprendido en la primera semana de aprendizaje del C. Ha sido una gran cantidad de material en sólo una semana, ¡pero lo ha logrado! Si usa todo lo que ha aprendido esta semana podrá escribir sus propios programas en C. Sin embargo, todavía hay límites a lo que puede hacer.

SEMANA

2

UN VISTAZO

Ya ha terminado su primera semana de aprendizaje sobre cómo programar en C. Por ahora debe sentirse a gusto tecleando programas y usando el editor y el compilador.

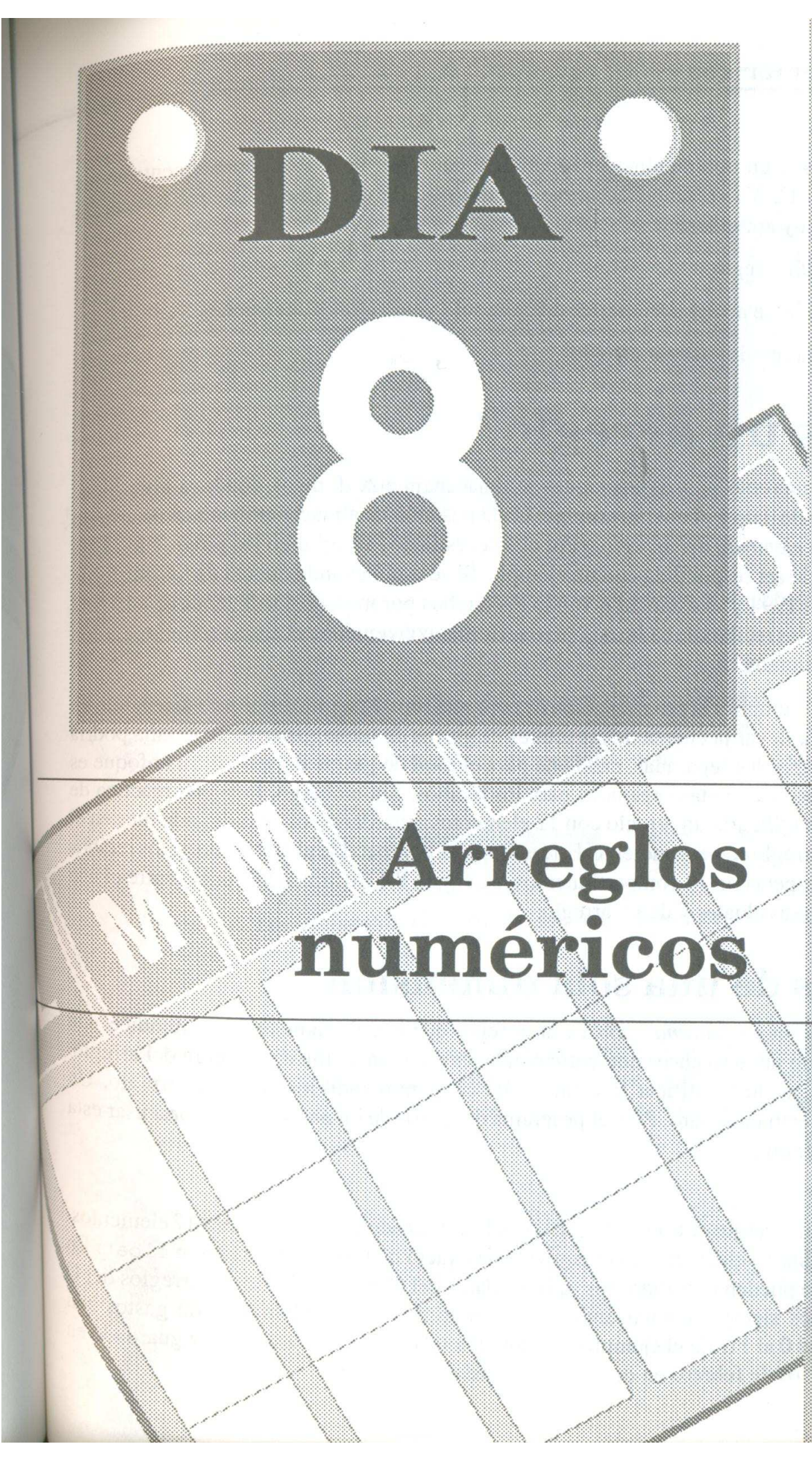
Adónde vamos...

La segunda semana trata una gran cantidad de material. Aprenderá muchas de las características que forman la parte medular del lenguaje C. Aprenderá cómo usar arreglos numéricos y de carácter, expandir variables de tipo carácter en arreglos y cadenas, y agrupar tipos diferentes de variables usando estructuras.

La segunda semana complementa algunos de los temas aprendidos en la primera semana, presenta enunciados adicionales para el control de programa, proporciona explicaciones detalladas de funciones y presenta funciones alternativas.

Los Días 9, "Apuntadores" y 12, "Alcance de las variables", se enfocan en conceptos del C que son extremadamente importantes. Pasará tiempo extra trabajando con apuntadores y sus funciones básicas.

Al final de la primera semana aprendió a escribir varios programas en C simples. Para cuando termine la segunda, deberá ser capaz de escribir programas complejos que puedan acometer casi cualquier tarea.



Arreglos numéricos

Los arreglos son un tipo de almacenamiento de datos que se usan frecuentemente en los programas en C. Ya se tuvo una breve introducción en el Día 6, "Control básico del programa". Hoy aprenderá

- Lo que es un arreglo.
- La definición de arreglos numéricos de una sola y de varias dimensiones.
- Cómo declarar e inicializar arreglos.

¿Qué es un arreglo?

Un *arreglo* es una colección de posiciones de almacenamiento de datos, donde cada una tiene el mismo tipo de dato y el mismo nombre. Cada posición de almacenamiento en un arreglo es llamada un *elemento del arreglo*. ¿Por qué necesitamos arreglos en los programas? Esta pregunta puede ser respondida con un ejemplo. Si se está llevando cuenta de los gastos de un negocio en 1994 y se están archivando los recibos por mes, se puede tener una carpeta separada para los recibos de cada mes, pero sería más conveniente tener una sola carpeta con 12 compartimientos.

Extienda este ejemplo a la programación de computadora. Imagínese que está diseñando un programa para llevar la cuenta de los totales de gastos de un negocio. El programa podría declarar 12 variables separadas, cada una para el total de gastos del mes. Este enfoque es similar a tener 12 carpetas separadas para los recibos. Sin embargo, la buena práctica de programación utilizaría un arreglo con 12 elementos, guardando el total de cada mes en el elemento de arreglo correspondiente. Este enfoque es comparable al archivado de los recibos en una sola carpeta con 12 compartimientos. La figura 8.1 ilustra la diferencia entre el uso de variables individuales y de un arreglo.

Arreglos de una sola dimensión

Un arreglo de una *sola dimensión* es un arreglo que tiene solamente un subíndice. Un *subíndice* es un número encerrado entre corchetes a continuación del nombre del arreglo. Este número puede identificar la cantidad de elementos individuales en el arreglo. Un ejemplo hará esto más claro. Para el programa de gastos de un negocio, se podría usar esta línea del programa

```
float gastos[12];
```

para declarar un arreglo de tipo `float`. El arreglo es llamado gastos y contiene 12 elementos. Cada uno de los 12 elementos es el equivalente exacto de una sola `variable float`. En los arreglos se pueden usar todos los tipos de datos del C. Los elementos de arreglos del C son numerados siempre comenzando en 0, por lo que los 12 elementos de gastos son numerados del 0 al 11. En el ejemplo anterior, el total de gastos de enero sería guardado en `gastos[0]`, los de febrero en `gastos[1]` y así sucesivamente.

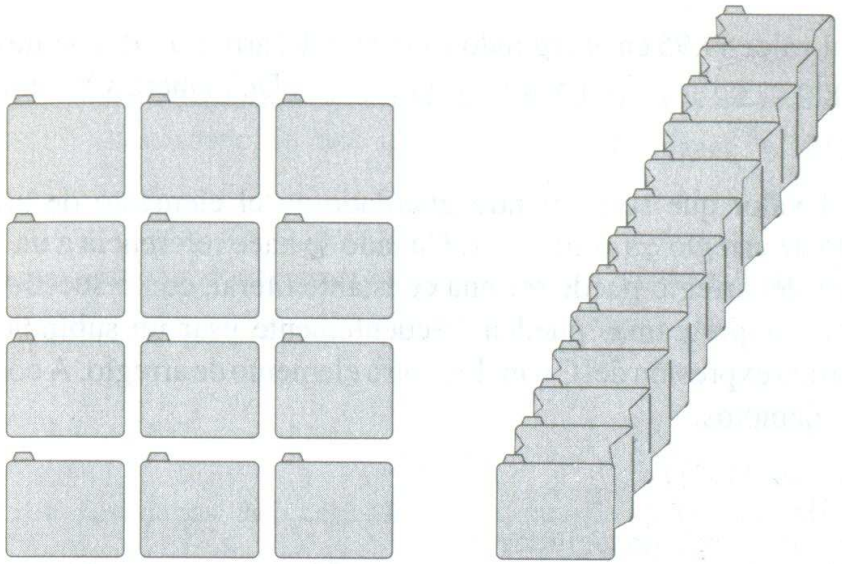


Figura 8.1. *Las variables son como carpetas individuales, y un arreglo es como una sola carpeta con muchos compartimientos.*

Cuando se declara un arreglo, el compilador reserva un bloque de memoria lo suficientemente grande como para guardar el arreglo completo. Los elementos individuales del arreglo son guardados en posiciones consecutivas de memoria, como se ilustra en la figura 8.2.

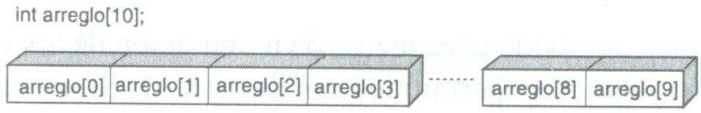


Figura 8.2. *Los elementos del arreglo son guardados en posiciones secuenciales en memoria.*

La ubicación de las declaraciones de arreglos en el código fuente es importante. De manera similar a las variables que no son de arreglo, la posición de la declaración afecta la manera en que el programa puede usar el arreglo. El efecto de la ubicación de la declaración se trata a mayor detalle en el Día 12, "Alcance de las variables". Por ahora ponga las declaraciones de arreglo junto con las otras declaraciones de variable, inmediatamente antes del inicio de `main()`.

Un elemento de arreglo puede ser usado en cualquier parte del programa en donde pueda ser usada una variable del mismo tipo que no sea de arreglo. Los elementos individuales de los arreglos son accesados usando el nombre del arreglo, seguido del subíndice del elemento encerrado entre corchetes. Por ejemplo, el enunciado

```
gastos[1] = 89.95;
```

guarda el valor 89.95 en el segundo elemento del arreglo. (Recuerde que el primer elemento del arreglo es `gastos[0]` y no `gastos[1]`.) De manera similar, el enunciado

```
gastos[10] = gastos[11];
```

asigna el valor que se encuentra guardado en el elemento de arreglo `gastos[11]` al elemento de arreglo `gastos[10]`. Cuando se hace referencia a un elemento de arreglo, el subíndice del arreglo puede ser una constante literal, como sucede en estos ejemplos. Sin embargo, los programas pueden frecuentemente usar un subíndice que es una variable entera, o una expresión del C o incluso otro elemento de arreglo. A continuación se presentan algunos ejemplos:

```
float gastos[100];
int a[10];
/* aquí van otros enunciados */
gastos[i] = 100;      /* i es una variable entera      */
gastos[2+3] = 100;    /* es equivalente a gastos[5]    */
gastos[a[2]] = 100;   /* a[] es un arreglo de enteros */
```

El último ejemplo tal vez necesite una explicación. Digamos, por ejemplo, que se tiene un arreglo de enteros llamado `a[]` y que el valor 8 se encuentra guardado en el elemento `a[2]`. Entonces, al escribir

```
gastos[a[2]]
```

se tiene el mismo efecto que al escribir

```
gastos[8];
```

Cuando use arreglos no olvide el esquema de numeración de los elementos: en un arreglo de *n* elementos los subíndices permitidos van de 0 a *n* - 1. Si se usa un valor de subíndice *n* se pueden tener errores de programa. El compilador de C no se da cuenta si el programa usa un subíndice de arreglo que está fuera de los límites. El programa compila y enlaza, pero, por lo general, los subíndices fuera de rango producen resultados erróneos.

Algunas veces tal vez quiera tratar un arreglo de *n* elementos como si sus elementos estuvieran numerados del 1 al *n*. Por ejemplo, en el ejemplo anterior sería un método más natural guardar el total de gastos de enero en `gastos[1]`, los de febrero en `gastos[2]` y así sucesivamente. La manera más simple de hacer esto es declarar el arreglo con un elemento más de los necesarios e ignorar al elemento `0`. En este caso se declararía al arreglo

```
float gastos[13];
```

También se podría guardar algún dato relacionado en el elemento `0` (tal vez el total de gastos anuales).

El programa EXPENSES.C, que se encuentra en el listado 8.1, muestra el uso de un arreglo. Este es un programa simple que no tiene un uso práctico, sino que es solamente para objetos de demostración.

Captura

Listado 8.1. EXPENSES.C.

```
1:  /* EXPENSES.C - Demostración del uso de un arreglo */
2:
3:  #include <stdio.h>
4:
5:  /* Declara un arreglo donde guarda gastos y una variable de contador */
6:
7:  float expenses[13];
8:  int count;
9:
10: main()
11: {
12:     /* Recibe los datos del teclado y los guarda en el arreglo */
13:
14:     for (count = 1; count < 13; count++)
15:     {
16:         printf("Enter expenses for month %d: ", count);
17:         scanf("%f", &expenses[count]);
18:     }
19:
20:     /* Imprime el contenido del arreglo */
21:
22:     for (count = 1; count < 13; count++)
23:     {
24:         printf("\nMonth %d = $%.2f", count, expenses[count]);
25:     }
26: }
```

Salida

```
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00

Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
```

```
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

Análisis Cuando se ejecuta EXPENSES.C, el programa le pide que teclee los gastos para los meses del 1 al 12. Los valores que se teclean son guardados en un arreglo. Se debe dar algún valor para cada mes. Después de que se da el duodécimo valor se despliega en la pantalla el contenido del arreglo.

El flujo del programa es similar a los listados que se han visto anteriormente. La línea 1 comienza con un comentario que describe lo que el programa va a hacer. Observe que está incluido el nombre del programa, EXPENSES.C. Incluyendo el nombre del programa en un comentario se sabe qué programa se está viendo. Esto es útil cuando se imprimen los listados y luego se quiere hacer algún cambio.

La línea 5 contiene un comentario adicional, con una explicación sobre las variables que se están declarando. En la línea 7 es declarado un arreglo de 13 elementos. En este programa sólo se necesitan 12 elementos, uno para cada mes, pero han sido declarados 13. El ciclo `for` que se encuentra en las líneas 14 a 18 ignora al elemento `0`. Esto permite que el programa use los elementos del `1` al `12`, que están relacionados directamente con los 12 meses. Regresando, en la línea 8 es declarada una variable, `count`, y usada a lo largo del programa como contador e índice de arreglo.

La función `main()` del programa comienza en la línea 10. Como se dijo anteriormente, el programa usa un ciclo `for` para imprimir un mensaje y aceptar un valor para cada uno de los 12 meses. Observe que en la línea 17 la función `scanf()` usa un elemento de arreglo. En la línea 7 el arreglo expenses fue declarado como `float`, por lo que se usa `%f`. También es puesto el operador de *dirección de* (&) antes del elemento del arreglo, como si fuera una variable regular tipo `float` y no un elemento de arreglo.

Las líneas 22 a 25 contienen un segundo ciclo `for`, que imprime los valores que se acaban de teclear. Se ha añadido un comando de formateo adicional a la función `printf()`, para que de esta forma los valores de gastos se impriman en forma mejor ordenada. Por ahora, simplemente sepa que `%.2f` imprime un número de punto flotante con dos decimales. En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se tratan comandos adicionales para el formateo.

DEBE	NO DEBE
NO DEBE Olvidar que los subíndices de arreglo comienzan con el elemento 0.	
DEBE Usar arreglos en vez de crear varias variables que guarden la misma cosa.	

(Por ejemplo, si se quieren guardar las ventas totales para cada mes del año, cree un arreglo con 12 elementos que guarden las ventas, en vez de crear una variable de ventas para cada mes.)

Arreglos multidimensionales

Un arreglo multidimensional tiene más de un subíndice. Un *arreglo bidimensional* tiene dos subíndices, un *arreglo tridimensional* tiene tres subíndices y así sucesivamente. No hay límite a la cantidad de dimensiones que pueda tener un arreglo en C. (Hay un límite sobre el tamaño total del arreglo, que se mencionará posteriormente en este capítulo.)

Por ejemplo, tal vez quiera escribir un programa que juegue damas. El tablero de damas contiene 64 cuadros, acomodados en 8 renglones y 8 columnas. El programa podría representar al tablero como un arreglo bidimensional de la manera siguiente:

```
int cuadro[8][8];
```

El arreglo resultante tiene 64 elementos: `cuadro[0][0]`, `cuadro[0][1]`, `cuadro[0][2]... cuadro[7][6]`, `cuadro[7][7]`. La estructura de este arreglo bidimensional se ilustra en la figura 8.3.

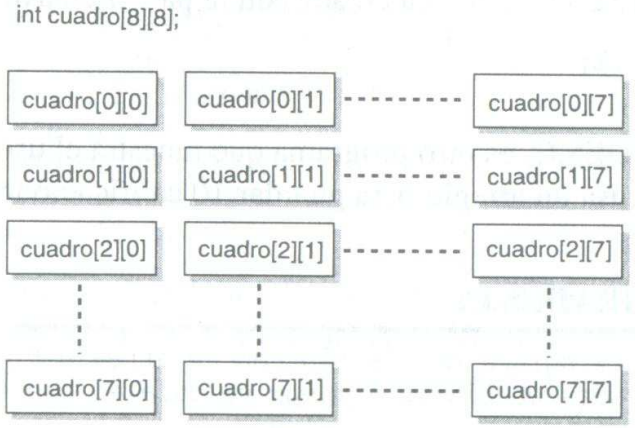


Figura 8.3. *Un arreglo bidimensional tiene una estructura de columnas y renglones.*

De manera similar, se podría pensar en un arreglo tridimensional para un cubo. Dejamos a su imaginación a los arreglos de cuatro dimensiones (y superiores). Todos los arreglos, sin importar qué tantas dimensiones tengan, son guardados secuencialmente en memoria. En el Día 15, "Más sobre apuntadores", se dan más detalles sobre el almacenamiento de arreglos.

Denominación y declaración de arreglos

Las reglas para asignar nombres a los arreglos son las mismas que para los nombres de variables, que fueron tratadas en el Día 3, "Variables y constantes numéricas". Un nombre de arreglo debe ser único. No puede ser usado para otro arreglo o para cualquier otro identificador (variable, constante, etc.). Como probablemente se ha dado cuenta, las declaraciones de arreglo siguen la misma forma que las variables que no son de arreglo, a excepción de que la cantidad de elementos del arreglo debe ser encerrada entre corchetes y puesta inmediatamente después del nombre del arreglo.

Cuando se declara un arreglo se puede especificar la cantidad de elementos con una constante literal (como se ha hecho en los ejemplos anteriores) o con una constante simbólica creada con la directiva `#define`. Por lo tanto,

```
#define MESES 12
int arreglo[MESES];
```

es equivalente a

```
int arreglo[12];
```

Sin embargo, en la mayoría de los compiladores no se pueden declarar los elementos del arreglo con una constante simbólica creada con la palabra clave `const`.

```
const int MESES = 12;
int arreglo[MESES];              /* ¡Erróneo! */
```

El listado 8.2, GRADES.C, es otro programa que muestra el uso de un arreglo unidimensional. GRADES.C usa un arreglo para guardar 10 calificaciones escolares.

Captura

Listado 8.2. GRADES.C.

```
1:   /* GRADES.C - Programa de ejemplo con un arreglo */
2:   /* Pide 10 grados escolares y luego calcula el promedio */
3:
4:   #include <stdio.h>
5:
6:   #define MAX_GRADE 100
7:   #define STUDENTS  10
8:
9:   int grades[STUDENTS];
10:
11:  int idx;
12:  int total = 0;           /* usado para promedio */
13:
14:  main()
15:  {
16:      for( idx=0;idx<1 STUDENTS;idx++)
```

```
17:     {
18:         printf( "Enter Person %d's grade: ", idx +1);
19:         scanf( "%d", &grades[idx] );
20:
21:         while ( grades[idx] > MAX_GRADE )
22:         {
23:             printf( "\nThe highest grade possible is %d",
    MAX_GRADE );
24:             printf( "\nEnter correct grade: " );
25:             scanf( "%d", &grades[idx] );
26:         }
27:
28:         total += grades[idx];
29:     }
30:
31:     printf( "\n\nThe average score is %d", ( total / STUDENTS) );
32:
33:     return (0);
34: }
```

Salida

```
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

Análisis

De manera similar a EXPENSES.C, este listado le pide datos al usuario. Le pide las calificaciones grados de 10 gentes. En vez de imprimir cada calificación, imprime el promedio.

Como ha visto anteriormente, los arreglos son nombrados de manera similar a las variables regulares. En la línea 9, el arreglo para este programa es llamado `grades`. Es correcto suponer que este arreglo guarda calificaciones escolares. En las líneas 6 y 7, se definen dos constantes, `MAX_GRADE` y `STUDENTS` (grado máximo y estudiantes). Estas constantes pueden ser cambiadas fácilmente. Sabiendo que `STUDENTS` está definido como 10, se sabe que el arreglo `grades` tiene 10 elementos. En el listado están declaradas otras dos variables, `idx` y `total`. Se usa `idx`, una abreviatura para índice, como contador y subíndice del arreglo. Un gran total de todos los grados se guarda en `total`.

La parte medular de este programa es el ciclo `for`, que se encuentra en las líneas 16 a 29. El enunciado `for` inicializa `idx` a `0`, el primer subíndice del arreglo. Luego se mantiene

haciendo ciclos mientras `idx` sea menor que la cantidad de estudiantes. Cada vez que inicia el ciclo, incrementa a `idx` en 1. En cada ciclo el programa pide la calificación de la persona (líneas 18 y 19). Observe que en la línea 18 se añade 1 a `idx`, para contar a la gente de 1 a 10 en vez de 0 a 9. Debido a que cualquier arreglo comienza con el subíndice 0, la primera calificación es puesta en `grade[0]`. En vez de confundir al usuario pidiéndole la calificación de la persona 0, se le pide la calificación de la persona 1.

Las líneas 21 a 26 contienen un ciclo `while` anidado dentro del ciclo `for`. Esta es una revisión de edición, que asegura que el grado no sea mayor que el grado máximo, `MAX_GRADE`. Se le pide al usuario teclear una calificación correcta, en caso de que haya dado una calificación que sea demasiado alta. Se deben revisar los datos del programa cada vez que sea posible.

La línea 28 suma el grado tecleado a un contador total. En la línea 31 es usado este total para imprimir el grado promedio (`total/STUDENTS`).

DEBE	NO DEBE

DEBE Usar enunciados `#define` para crear constantes que puedan usarse cuando se declaran arreglos. De esta manera podrá cambiar fácilmente la cantidad de elementos del arreglo. En GRADES.C se podría cambiar la cantidad de estudiantes en `#define` y no se tendría que hacer ningún otro cambio en el programa.

DEBE Evitar los arreglos multidimensionales con más de tres dimensiones. Recuerde que los arreglos multidimensionales pueden hacerse muy grandes rápidamente.

Inicialización de arreglos

Se puede inicializar a todo o parte del arreglo cuando se le declara. Ponga a continuación de la declaración de arreglo un signo de igual y una lista de valores, encerrados entre llaves y separados por comas. Los valores listados son asignados en orden a los elementos del arreglo, comenzando con el número 0. Por ejemplo,

```
int arreglo[4] = { 100, 200, 300, 400 };
```

asigna el valor `100` a `arreglo[0]`, `200` a `arreglo[1]`, `300` a `arreglo[2]` y `400` a `arreglo[3]`. Si se omite el tamaño del arreglo, el compilador crea un arreglo lo suficientemente grande como para guardar los valores de inicialización. Por lo tanto, el enunciado

```
int arreglo[] = { 100, 200, 300, 400 };
```

tendría exactamente el mismo efecto que el enunciado de declaración de arreglo anterior. Sin embargo, se pueden incluir menos valores de inicialización, como se ve en este ejemplo:

```
int arreglo[10] = {1, 2, 3 };
```

Si no se inicializa explícitamente a los elementos del arreglo, no se puede estar seguro del valor que contienen cuando ejecuta el programa. Si se incluyen demasiados valores inicializadores (más inicializadores que elementos de arreglo), el compilador detecta un error.

Los arreglos multidimensionales también pueden ser inicializados. La lista de valores de inicialización es asignada en orden a los elementos del arreglo, cambiando primero el último subíndice del arreglo. Por ejemplo,

```
int arreglo[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

da como resultado las siguientes asignaciones:

```
arreglo[0][0] es igual a 1
arreglo[0][1] es igual a 2
arreglo[0][2] es igual a 3
arreglo[1][0] es igual a 4
arreglo[1][1] es igual a 5
arreglo[1][2] es igual a 6
...
arreglo[3][1] es igual a 11
arreglo[3][2] es igual a 12
```

Cuando se inicializan arreglos multidimensionales, se puede hacer más claro el código fuente usando llaves adicionales para agrupar los valores de inicialización, y también distribuyéndolos en varias líneas. La siguiente inicialización es equivalente a la anterior:

```
int arreglo[4][3] = {{ 1, 2, 3 } , { 4, 5, 6 } ,
                     { 7, 8, 9 } , { 10, 11, 12 } };
```

Recuerde, los valores de inicialización deben estar separados por comas, incluso aunque haya llaves entre ellos. También asegúrese de usar las llaves en pares, una llave derecha para cada llave izquierda, ya que si no lo hace el compilador se confunde.

Ahora veamos un ejemplo que muestra las ventajas de los arreglos. El programa del listado 8.3, RANDOM.C, crea un arreglo tridimensional de 1000 elementos y lo llena con números al azar. Luego el programa despliega en la pantalla los elementos del arreglo. Imagine qué tantas líneas de código fuente necesitaría para ejecutar la misma tarea con variables que no sean de arreglo.

En este programa verá una nueva función de biblioteca, `getch()`. La función `getch()` lee un solo carácter del teclado. En el listado 8.3 `getch()` hace pausa en el programa hasta que el usuario oprima una tecla. `getch()` se trata a detalle en el Día 14, "Trabajando con la pantalla, la impresora y el teclado".

Captura

Listado 8.3. RANDOM.C.

```
1:  /* RANDOM.C - Demostración del uso de un arreglo multidimensional */
2:
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:  /* Declara un arreglo tridimensional con 1000 elementos */
6:
7:  int random[10][10][10];
8:  int a, b, c;
9:
10: main()
11: {
12:     /* Llena el arreglo con números al azar. La función de  */
13:     /* biblioteca del C, rand(), regresa un número al azar. */
14:     /* Usa un ciclo for para cada subíndice del arreglo.    */
15:
16:     for (a = 0; a < 10; a++)
17:     {
18:         for (b = 0; b < 10; b++)
19:         {
20:             for (c = 0; c < 10; c++)
21:             {
22:                 random[a][b][c] = rand();
23:             }
24:         }
25:     }
26:
27:     /* Ahora despliega los elementos del arreglo de 10 en 10 */
28:
29:     for (a = 0; a < 10; a++)
30:     {
31:         for (b = 0; b < 10; b++)
32:         {
33:             for (c = 0; c < 10; c++)
34:             {
35:                 printf("\nrandom[%d][%d][%d] = ", a, b, c);
36:                 printf("%d", random[a][b][c]);
37:             }
38:             printf("\nPress a key to continue, CTRL-C to \
                        quit.");
39:             getch();
40:         }
41:     }
42: }    /* fin de main() */
```

Salida

```
random[0][0][0] = 346
random[0][0][1] = 130
random[0][0][2] = 10982
```

```
random[0][0][3] = 1090
random[0][0][4] = 11656
random[0][0][5] = 7117
random[0][0][6] = 17595
random[0][0][7] = 6415
random[0][0][8] = 22948
random[0][0][9] = 31126
Press a key to continue, CTRL-C to quit.
random[0][1][0] = 9004
random[0][1][1] = 14558
random[0][1][2] = 3571
random[0][1][3] = 22879
random[0][1][4] = 18492
random[0][1][5] = 1360
random[0][1][6] = 5412
random[0][1][7] = 26721
random[0][1][8] = 22463
random[0][1][9] = 25047
Press a key to continue, CTRL-C to quit
      ...          ...
random[9][8][0] = 6287
random[9][8][1] = 26957
random[9][8][2] = 1530
random[9][8][3] = 14171
random[9][8][4] = 6951
random[9][8][5] = 213
random[9][8][6] = 14003
random[9][8][7] = 29736
random[9][8][8] = 15028
random[9][8][9] = 18968
Press a key to continue, CTRL-C to quit.
random[9][9][0] = 28559
random[9][9][1] = 5268
random[9][9][2] = 20182
random[9][9][3] = 3633
random[9][9][4] = 24779
random[9][9][5] = 3024
random[9][9][6] = 10853
random[9][9][7] = 28205
random[9][9][8] = 8930
random[9][9][9] = 2873
Press a key to continue, CTRL-C to quit.
```

Análisis En el Día 6, "Control básico del programa", se vio un programa que usaba un enunciado `for` anidado. Este programa tiene dos ciclos `for` anidados. Antes de que vea a detalle los enunciados `for`, observe que las líneas 7 y 8 declaran cuatro variables. La primera es un arreglo llamado `random`, usado para guardar los números al azar. `random` es un arreglo tipo `int` de tres dimensiones de 10 por 10 por 10, dando un total de 1,000 elementos tipo `int` (10 x 10 x 10). Imagínese tener que tratar con 1,000 nombres de variable únicos si no fuera posible usar arreglos. Luego la línea 8 declara tres variables, `a`, `b` y `c`, usadas para control de los ciclos `for`.

Este programa también incluye en la línea 4 un nuevo archivo de encabezado, STDLIB.H (que significa standard library: biblioteca estándar). Este es incluido para proporcionar el prototipo para la función `rand()`, que es usada en la línea 22.

El grueso del programa está contenido en dos enunciados `for` anidados. El primero se encuentra en las líneas 16 a 25 y el segundo en las líneas 29 a 41. Ambos `for` anidados tienen la misma estructura. Trabajan de manera similar a los ciclos del listado 6.2 pero van a un nivel más profundo. En el primer juego de enunciados `for`, la línea 22 se ejecuta repetidamente. La línea 22 asigna el valor de retorno de una función, `rand()`, a un elemento del arreglo `random`. `rand()` es una función de biblioteca que regresa un número al azar.

Regresando en el listado se puede ver que la línea 20 cambia a la variable `c` de `0` a `9`. Esto hace ciclo por el subíndice de la extrema derecha del arreglo `random`. La línea 18 hace ciclos sobre `b`, el subíndice medio del arreglo random. Cada vez que cambia a `b`, hace ciclo por todos los elementos `c`. La línea 16 incrementa la variable `a`, que hace ciclo por el subíndice de la extrema izquierda. Cada vez que cambia este subíndice hace ciclo por todos los 10 valores del subíndice `b`, que a su vez hace ciclo por todos los 10 valores de `c`. Este ciclo inicializa a cada uno de los valores del arreglo `random` con un número al azar.

Las líneas 29 a 41 contienen el segundo anidamiento de enunciados `for`. Funcionan de manera similar a los enunciados `for` anteriores, pero este ciclo imprime cada uno de los valores asignados anteriormente. Después de que son desplegados 10, la línea 38 imprime un mensaje y espera a que se oprima una tecla. La línea 39 se ocupa de la opresión de tecla. `getch()` regresa el valor de la tecla que ha sido oprimida. Si no se oprime una tecla, `getch()` espera hasta que suceda. Ejecute este programa y observe los valores desplegados.

Tamaño máximo del arreglo

Debido a la manera en que funcionan los modelos de memoria, se debe tratar, por ahora, de no crear más de 64 K de datos en variables. La explicación de este límite se encuentra más allá del alcance de este libro, pero no hay por qué preocuparse, ya que ninguno de los programas de este libro excede esta limitación. Para aprender más, o para resolver esta limitación, consulte los manuales del compilador. Por lo general 64 K es suficiente espacio de datos para los programas, en particular para los programas relativamente simples que se escribirán mientras estudie con este libro. Un solo arreglo puede ocupar los 64 K del almacenamiento de datos, si el programa no usa otras variables. De otra forma se necesita repartir el espacio de datos disponible en la manera adecuada.

El tamaño de un arreglo en bytes depende de la cantidad de elementos que tiene y del tamaño de cada elemento. El tamaño del elemento depende del tipo de dato del arreglo. Los tamaños para cada tipo de dato numérico, dados en la tabla 3.2, son repetidos aquí en la tabla 8.1 para su comodidad.

Tabla 8.1. Requerimientos de espacio de almacenamiento para los tipos de datos numéricos de la mayoría de las PC.

Elemento	
Tipo de dato	**Tamaño (Bytes)**
`int`	2
`short`	2
`long`	4
`float`	4
`double`	8

Para calcular el espacio de almacenamiento requerido para un arreglo, se multiplica la cantidad de elementos del arreglo por el tamaño del elemento. Por ejemplo, un arreglo de 500 elementos de tipo `float` requiere `(500) * (4) = 2000` bytes de espacio de almacenamiento.

El espacio de almacenamiento puede ser determinado dentro de un programa usando el operador `sizeof()` del C. `sizeof()` es un operador unario y no una función. Toma como argumento el nombre de una variable o el nombre de un tipo de dato y regresa el tamaño en bytes de su argumento. El uso de `sizeof()` es ilustrado en el listado 8.4.



Listado 8.4. Uso del operador `sizeof()` para determinar los requerimientos de espacio de almacenamiento para un arreglo.

```
1:  /* Demostración del operador sizeof() */
2:
3:  #include <stdio.h>
4:
5:  /* Declara varios arreglos de 100 elementos */
6:
7:  int intarray[100];
8:  float floatarray[100];
9:  double doublearray[100];
10:
11: main()
12: {
13:     /* Despliega el tamaño de los tipos de datos numéricos */
14:
15:     printf("\n\nSize of int = %d bytes", sizeof(int));
16:     printf("\nSize of short = %d bytes", sizeof(short));
17:     printf("\nSize of long = %d bytes", sizeof(long));
```

Listado 8.4. continuación

```
18:      printf("\nSize of float = %d bytes", sizeof(float));
19:      printf("\nSize of double = %d bytes", sizeof(double));
20:
21:      /* Despliega el tamaño de los tres arreglos */
22:
23:      printf("\nSize of intarray = %d bytes", sizeof(intarray));
24:      printf("\nSize of floatarray = %d bytes",
               sizeof(floatarray));
25:      printf("\nSize of doublearray = %d bytes",
               sizeof(doublearray));
26:
27:  }
```

Salida

```
Size of int = 2 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 200 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

Análisis

Teclee y compile el programa de este listado usando los procedimientos que aprendió en el Día 1, "Comienzo". Cuando el programa ejecuta, despliega el tamaño en bytes de los tres arreglos y de los cinco tipos de datos numéricos.

En el Día 3, "Variables y constantes numéricas", se ejecutó un programa similar. Sin embargo, este listado usa `sizeof()` para determinar el tamaño de almacenamiento de arreglos. Las líneas 7, 8 y 9 declaran tres arreglos, cada uno de diferente tipo. Las líneas 23, 24 y 25 imprimen el tamaño de cada arreglo. El tamaño debe ser igual al tamaño del tipo de variable del arreglo multiplicado por la cantidad de elementos. Por ejemplo, si un `int` es de dos bytes, `intarray` debe ser 2 por 100, o sea 200 bytes. Ejecute el programa y revise los valores.

Resumen

Este capítulo presenta los arreglos numéricos, un método poderoso de almacenamiento de datos que le permite agrupar una cantidad de conceptos de datos del mismo tipo bajo el mismo nombre de grupo. Los conceptos individuales, o elementos del arreglo, son identificados usando un subíndice después del nombre del arreglo. Las tareas de programación de computadora que involucran el procesamiento repetitivo de datos lo llevan por sí mismas al almacenamiento en arreglos.

De manera similar a las variables que no son de arreglos, los arreglos deben ser declarados antes de que puedan usarse. En forma opcional, los elementos del arreglo pueden ser inicializados cuando es declarado el arreglo.

Preguntas y respuestas

1. ¿Qué pasa si uso en un arreglo un subíndice que es mayor a la cantidad de elementos del arreglo?

   Si se usa un subíndice que está fuera de los límites de la declaración del arreglo, es probable que el programa compile y hasta corra. Sin embargo, los resultados de este error son impredecibles. Este puede ser un error difícil de encontrar cuando empieza a causar problemas, por lo que debe asegurarse de ser cuidadoso cuando inicialice y accese los elementos del arreglo.

2. ¿Qué pasa si uso un arreglo sin inicializarlo?

   Este error no produce errores de compilación. Si no se inicializa un arreglo habrá cualquier valor en los elementos del arreglo. Se pueden obtener resultados impredecibles. Se debe siempre inicializar las variables y los arreglos, para que de esta forma se sepa exactamente lo que hay en ellos. En el Día 12, "Alcance de las variables", se presenta una excepción a la necesidad de inicialización. Por el momento asegúrese de hacerlo.

3. ¿Qué tantas dimensiones puede tener un arreglo?

   Como se dijo en el capítulo, se pueden tener tantas dimensiones como quiera. Conforme añade más dimensiones se usa más espacio de almacenamiento de datos. Se debe declarar al arreglo tan grande como se necesite para evitar el desperdicio de espacio de almacenamiento.

4. ¿Hay alguna manera fácil de inicializar todo un arreglo de un jalón?

   Cada elemento de un arreglo debe ser inicializado. La manera más segura para que un programador de C novato inicialice un arreglo es con una declaración, como se mostró en este capítulo, o con un enunciado `for`. Hay otras formas de inicializar un arreglo, pero están más allá del alcance de este capítulo y del libro en este momento.

5. ¿Puedo sumar dos arreglos (o multiplicarlos, dividirlos o restarlos)?

   Si se declaran dos arreglos no se les puede sumar. Cada elemento debe ser sumado individualmente. El ejercicio 10 ilustra este punto.

6. ¿Por qué es mejor usar un arreglo en vez de variables individuales?

   Con los arreglos se pueden agrupar valores con un solo nombre. En el listado 8.3 fueron guardados 1,000 valores. La creación de 1,000 nombres de variable, y la inicialización de cada una de ellas a un número al azar, habría necesitado una tremenda cantidad de tecleo. Usando un arreglo se facilita la tarea.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado y ejercicios para darle experiencia en el uso de lo que ha aprendido.

Cuestionario

1. ¿Cuáles de los tipos de datos del C pueden usarse en un arreglo?
2. Si se declara un arreglo con 10 elementos, ¿cuál es el subíndice del primer elemento?
3. En un arreglo de una sola dimensión declarado con n elementos, ¿cuál es el subíndice del último elemento?
4. ¿Qué pasa si el programa trata de accesar un elemento con un subíndice fuera de rango?
5. ¿Cómo se declara un arreglo multidimensional?
6. Un arreglo es declarado con el enunciado

   ```
   int arreglo[2][3][5][8];
   ```

   ¿Qué tantos elementos tiene el arreglo?

Ejercicios

1. Escriba una línea de programa en C que declare tres arreglos enteros de una dimensión, llamados uno, dos y tres, con 1,000 elementos cada uno.
2. Escriba los enunciados necesarios para declarar un arreglo entero de 10 elementos e inicializar todos sus elementos a 1.
3. Dado el arreglo

   ```
   int ochentayocho[88];
   ```

   escriba el código para inicializar a todos los elementos del arreglo a 88.
4. Dado el arreglo

   ```
   int cosa[12][10];
   ```

   escriba el código para inicializar todos los elementos del arreglo a 0.

5. BUSQUEDA DE ERRORES: ¿Qué hay de erróneo en el siguiente fragmento de código?

```
int x, y;
int array[10][3];
main()
{
    for ( x = 0; x < 3; x++ )
        for ( y = 0; y < 10; y++ )
            array[x][y] = 0;
}
```

6. BUSQUEDA DE ERRORES: ¿Qué error tiene lo siguiente?

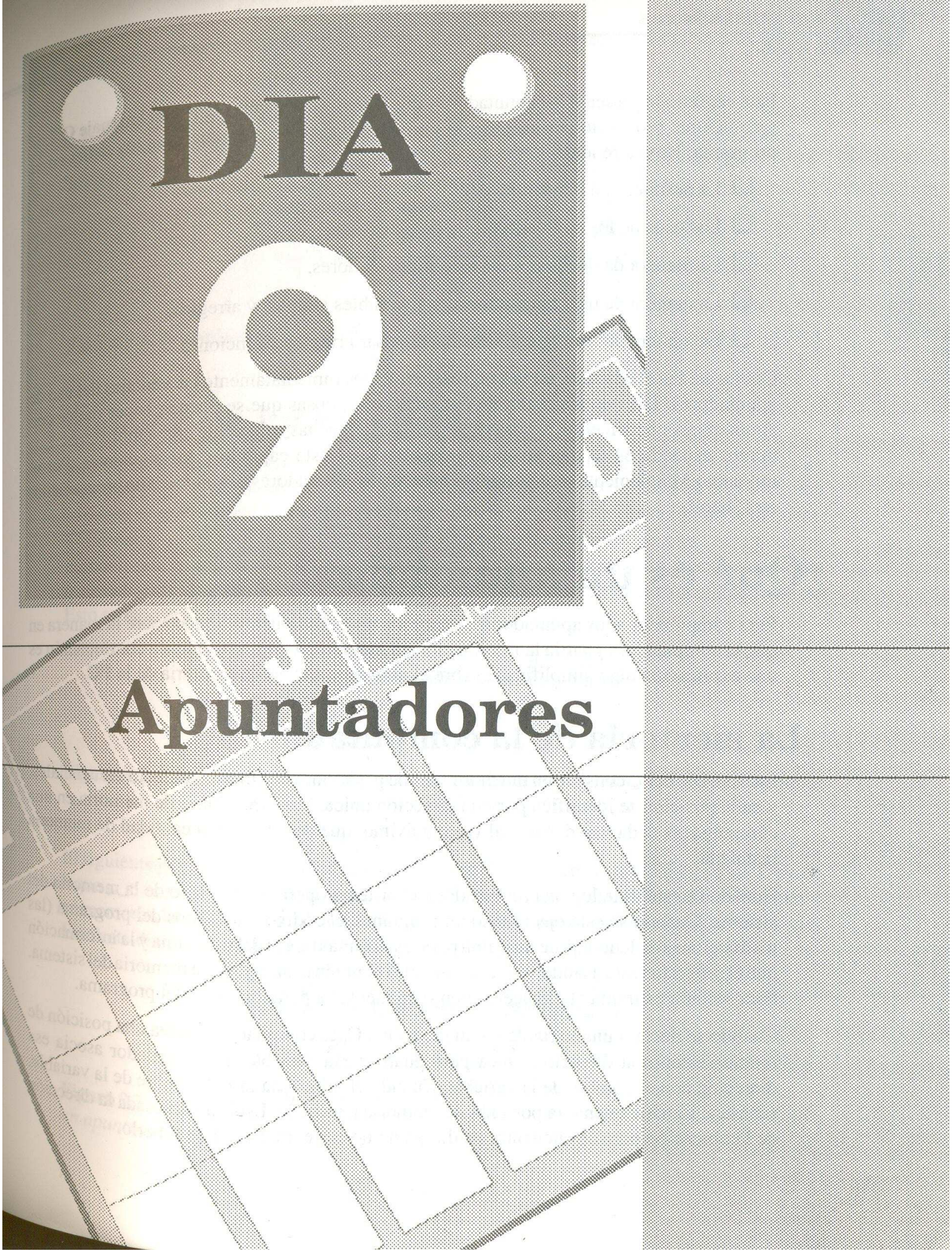
```
int array[10];
int x = 1;

main()
{
    for ( x = 1; x <= 10; x++ )
        array[x] = 99;
}
```

7. Escriba un programa que ponga números al azar en un arreglo bidimensional de 5 por 4. Imprima en la pantalla los valores en columnas. (Consejo: Use la función `rand()` del listado 8.3.)

8. Vuelva a escribir el listado 8.3 para usar un arreglo de una sola dimensión. Imprima el promedio de las 1,000 variables antes de imprimir los valores individuales. Nota: No olvide hacer una pausa después de imprimir cada 10 valores.

9. Escriba un programa que inicialice un arreglo de 10 elementos. El valor de cada elemento debe ser igual a su subíndice. Luego, el programa debe imprimir cada uno de los 10 elementos.

10. Modifique el programa del ejercicio 9. Después de imprimir los valores inicializados, el programa debe copiar los valores a un nuevo arreglo y sumar 10 a cada valor. Luego se deben imprimir los valores del nuevo arreglo.



Apuntadores

Este capítulo le presenta los apuntadores, que son una parte importante del lenguaje C. Los apuntadores proporcionan un método poderoso y flexible para manejar los datos en el programa. Hoy aprenderá

- La definición de un apuntador.
- Los usos de los apuntadores.
- La manera de declarar e inicializar apuntadores.
- La manera de usar apuntadores con variables simples y arreglos.
- La manera de usar apuntadores para pasar arreglos a funciones.

Conforme lea el capítulo, tal vez no sean evidentes inmediatamente las ventajas del uso de apuntadores. Las ventajas caen en dos categorías: cosas que se pueden hacer mejor con apuntadores que sin ellos, y cosas que pueden ser hechas solamente con apuntadores. Los puntos específicos se irán aclarando conforme lea este capítulo y los siguientes. Por el momento, simplemente sepa que debe entender los apuntadores, si quiere ser un programador en C hábil.

¿Qué es un apuntador?

Para comprender a los apuntadores se necesita un conocimiento básico sobre la manera en que la computadora guarda la información en memoria. Lo que se indica a continuación es una explicación algo simplificada sobre el almacenamiento de memoria de la PC.

La memoria de la computadora

La RAM de la PC consiste en muchos miles de posiciones de almacenamiento secuenciales, y cada posición se identifica por una dirección única. Las direcciones de memoria en una computadora dada van desde 0 al valor máximo, que depende de la cantidad de memoria instalada.

Cuando se está usando una computadora, el sistema operativo usa algo de la memoria del sistema. Cuando se está ejecutando un programa, el código y los datos del programa (las instrucciones de lenguaje de máquina para las diversas tareas del programa y la información que el programa está usando, respectivamente) también usan algo de la memoria del sistema. Esta sección examina el almacenamiento en memoria para los datos del programa.

Cuando se declara una variable en un programa C, el compilador reserva una posición de memoria con una dirección única para guardar esa variable. El compilador asocia esa dirección con el nombre de la variable. Cuando el programa usa el nombre de la variable, accesa automáticamente la posición de memoria adecuada. Está siendo usada la dirección de la ubicación pero se encuentra oculta y uno no tiene necesidad de saberlo.

La figura 9.1 muestra esto esquemáticamente. Una variable, llamada `tasa`, ha sido declarada e inicializada a `100`. El compilador ha reservado espacio de almacenamiento en la dirección `1004` para la variable, y ha asociado el nombre `tasa` con la dirección `1004`.

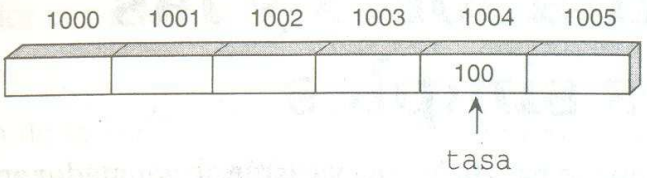


Figura 9.1. *Una variable de programa es guardada en una dirección de memoria específica.*

Creación de un apuntador

Se debe observar que la dirección de la variable `tasa` (o de cualquier otra variable) es un número y puede ser tratado como cualquier otro número en C. Si se sabe la dirección de una variable, se puede crear una segunda variable donde se guarde la dirección de la primera. El primer paso es declarar una variable para guardar la dirección de `tasa`. Démosle el nombre, por ejemplo, `p_tasa`. Al principio `p_tasa` se encuentra sin inicializar. Se ha reservado espacio de almacenamiento para `p_tasa`, pero su valor se encuentra indeterminado. Esto se muestra en la figura 9.2.

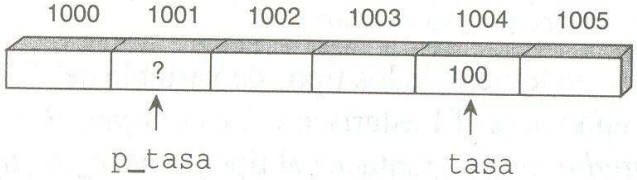


Figura 9.2. *El espacio de almacenamiento de memoria ha sido ubicado para la variable* `p_tasa`.

El siguiente paso es guardar la dirección de la variable `tasa` en la variable `p_tasa`. Debido a que `p_tasa` ahora contiene la dirección de `tasa`, indica su posición de almacenamiento en memoria. En la manera de hablar del C, `p_tasa` apunta a `tasa`, o es un apuntador a `tasa`. Esto está diagramado en la figura 9.3.

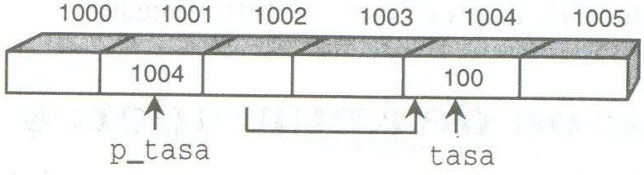


Figura 9.3. *La variable* `p_tasa` *contiene la dirección de la variable* `tasa`, *y es, por lo tanto, un apuntador a* `tasa`.

Resumiendo, un apuntador es una variable que contiene la dirección de otra variable. Ahora podemos ir a los detalles sobre el uso de apuntadores en los programas en C.

Los apuntadores y las variables simples

En los ejemplos que se han dado, una variable de apuntador apuntaba a una variable simple (esto es, que no es un arreglo). Esta sección le muestra cómo crear y usar apuntadores a variables simples.

Declaración de apuntadores

Un apuntador es una variable numérica y, de manera similar a todas las variables, debe ser inicializada antes de que pueda ser usada. Los `nombres` de variables de apuntador siguen las mismas reglas que otras variables y deben ser únicos. Este capítulo usa la convención de que un apuntador a la variable `nombre` es llamado *p_nombre*. Esto no es necesario, ya que se puede nombrar a los apuntadores en cualquier forma que se desee (siguiendo las reglas del C).

Una declaración de apuntador toma la siguiente forma:

```
nombre_de_tipo *nombre_de_apuntador;
```

nombre_de_tipo es cualquiera de los tipos de variable del C, e indica el tipo de variable a la cual apunta el apuntador. El asterisco (*) es el *operador de indirección*, e indica que *nombre_de_apuntador* es un apuntador al tipo *nombre_de_tipo* y no una variable de tipo *nombre_de_tipo*. Los apuntadores pueden ser declarados junto con variables que no son apuntadores. A continuación se presentan algunos ejemplos:

```
char *ch1, *ch2;          /* ch1 y ch2 son apuntadores a tipo char */
float *valor, porcentaje; /* valor es un apuntador a tipo float */
                          /*y porcentaje es una variable float ordinaria */
```

El símbolo * es usado tanto como operador de indirección como operador de multiplicación. No se preocupe pensando que el compilador puede confundirse. El contexto donde se usa el * siempre proporciona suficiente información, de tal forma que el compilador puede imaginarse si se trata de indirección o de multiplicación.

Inicialización de apuntadores

Ahora que ya se ha declarado un apuntador, ¿qué se puede hacer con él? No puede hacer nada con él mientras no haga que apunte a algo. De manera similar a las variables regulares, los apuntadores sin inicializar pueden ser usados, pero los resultados son impredecibles y

potencialmente desastrosos. Mientras un apuntador no guarde la dirección de una variable, es inútil. La dirección no se almacena mágicamente en el apuntador, sino que el programa debe ponerla ahí usando el operador de *dirección de*, (&). Cuando es puesto antes del nombre de una variable, el operador de *dirección de* regresa la dirección de la variable. Por lo tanto, se inicializa un apuntador con un enunciado de la forma

```
apuntador = &variable;
```

Regresemos al ejemplo de la figura 9.3. El enunciado de programa para inicializar a la variable `p_tasa` para que apunte a la variable `tasa`, debe ser

```
p_tasa = &tasa;        /* asigna la dirección de tasa a p_tasa */
```

Antes de la inicialización `p_tasa` no apuntaba a nada en particular. Después de la inicialización `p_tasa` es un apuntador a `tasa`.

Uso de apuntadores

Ahora que ya sabemos cómo declarar e inicializar los apuntadores, probablemente se pregunte cómo se les usa. Aquí entra nuevamente en juego el operador (*) de indirección. Cuando el * precede al nombre de un apuntador, hace referencia a la variable a la que apunta.

Continuemos con el ejemplo anterior, donde el apuntador `p_tasa` ha sido inicializado para que apunte a la variable `tasa`. Si se escribe `*p_tasa` se hace referencia a la variable `tasa`. Si se quiere imprimir el valor de `tasa` (que es de `100` en este ejemplo) se podría escribir

```
printf(%d,  tasa);
```

o se podría escribir

```
printf(%d,  *p_tasa);
```

En C los dos enunciados son equivalentes. Accesar el contenido de una variable usando el nombre de la variable es llamado *acceso directo*. Accesar el contenido de una variable usando un apuntador a la variable es llamado *acceso indirecto* o *indirección*. La figura 9.4 ilustra que un nombre de apuntador precedido por el operador de indirección hace referencia al valor de la variable apuntada.

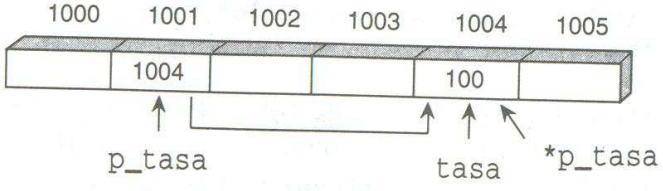


Figura 9.4. *Uso del operador de indirección con apuntadores.*

Haga una pausa y piense acerca de esto. Los apuntadores son una parte integral del lenguaje C y es esencial que los comprenda. Los apuntadores han confundido a mucha gente, por lo que no se preocupe si a usted también le pasa. Si necesita revisarlo, es correcto. Tal vez el siguiente resumen le ayude. Si tiene un apuntador llamado `ptr`, que ha sido inicializado para que apunte a la variable `var`, entonces

> `*ptr` y `var` hacen referencia al contenido de `var` (esto es, a cualquier valor que el programa haya almacenado ahí).
> `ptr` y `&var` hacen referencia a la dirección de `var`.

Como puede ver, un nombre de apuntador sin el operador de indirección accesa el propio valor del apuntador, el cual es, por supuesto, la dirección de la variable a la que apunta.

El programa que se encuentra en el listado 9.1 muestra el uso básico de los apuntadores. Usted debe teclear, compilar y ejecutar este programa.

Captura

Listado 9.1. Ilustración del uso básico de apuntadores.

```
1: /* Demuestra el uso básico de apuntadores */
2:
3: #include <stdio.h>
4:
5: /* Declara e inicializa una variable int */
6:
7: int var = 1;
8:
9: /* Declara un apuntador a int */
10:
11: int *ptr;
12:
13: main()
14: {
15:     /* Inicializa ptr para que apunte a var */
16:
17:     ptr = &var;
18:
19:     /* Accesa var directa e indirectamente */
20:
21:     printf("\nDirect access, var = %d", var);
22:     printf("\nIndirect access, var = %d", *ptr);
23:
24:     /* Despliega la dirección de var de dos maneras */
25:
26:     printf("\n\nThe address of var = %d", &var);
27:     printf("\nThe address of var = %d", ptr);
28: }
```

Salida

A continuación se muestra la salida de este programa. Las direcciones reportadas para `var` tal vez no sean 96 en su sistema.

```
Direct access, var = 1
Indirect access, var = 1

The address of var = 96
The address of var = 96
```

Análisis En este listado se declaran dos variables. En la línea 7 es declarada `var` como `int` e inicializada a 1. En la línea 11 es declarado un apuntador a una variable de tipo `int` y denominado `ptr`. En la línea 17, al apuntador `ptr` le es asignada la dirección de `var` usando el operador de *dirección de* (&). El resto del programa imprime en la pantalla los valores de estas dos variables. La línea 21 imprime el valor de `var`, y la línea 22 imprime el valor guardado en la posición apuntada por `ptr`. En este programa el valor es 1. La línea 26 imprime la dirección de `var` usando al operador de *dirección de*. Este es el mismo valor que es impreso por la línea 27 usando la variable de apuntador, `ptr`.

Es conveniente estudiar este listado. En él se muestra la relación entre una variable, su dirección, un apuntador y la referencia a un apuntador.

DEBE	NO DEBE
DEBE Entender lo que son los apuntadores y la manera en que funcionan. El dominio de C requiere el dominio de los apuntadores.	**NO DEBE** Usar apuntadores sin inicializar. Si lo hace, los resultados pueden ser desastrosos.

Los apuntadores y los tipos de variables

La discusión anterior ignora el hecho de que tipos diferentes de variables ocupan cantidades diferentes de memoria. En la mayoría de las PC un `int` ocupa dos bytes, un `float` ocupa cuatro bytes y así sucesivamente. Cada byte individual de memoria tiene su propia dirección, por lo que una variable que ocupa varios bytes, de hecho ocupa varias direcciones.

Entonces, ¿cómo manejan los apuntadores las direcciones de las variables de varios bytes? Esta es la manera en que funcionan: la dirección de una variable es, de hecho, la dirección del byte más bajo que ocupa. Esto puede ilustrarse con un ejemplo que declara e inicializa tres variables.

```
int vint = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```

Estas variables son guardadas en memoria como lo muestra la figura 9.5. La variable `int` ocupa dos bytes, la variable `char` ocupa un byte y la variable `float` ocupa cuatro bytes.

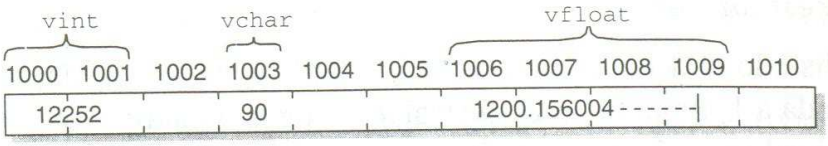


Figura 9.5. *Los diferentes tipos de variables numéricas ocupan diferente cantidad de espacio de almacenamiento en memoria.*

Ahora declare e inicialice apuntadores a estas tres variables.

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
/* aquí va código adicional */
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Cada apuntador es igual a la dirección del primer byte de la variable a la que apunta. Por lo tanto, `p_vint` es igual a `1000`, `p_vchar` es igual a `1003`, y `p_vfloat` es igual a `1006`. Sin embargo, recuerde que cada apuntador fue declarado para que apuntara a determinado tipo de variable. El compilador "sabe" que un apuntador a tipo `int` apunta al primero de los dos bytes, un apuntador de tipo `float` apunta al primero de los cuatro bytes, y así sucesivamente. Esto es diagramado en la figura 9.6.

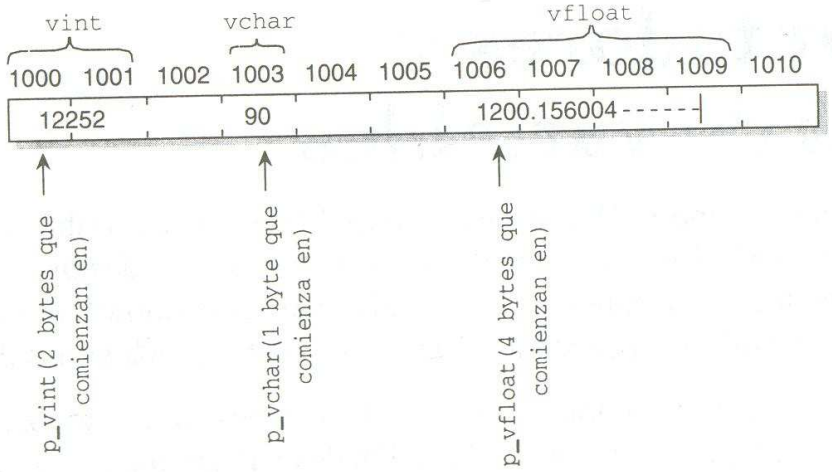


Figura 9.6. *El compilador "sabe" el tamaño de la variable a la que apunta el apuntador.*

Las figuras 9.5 y 9.6 muestran algunas posiciones de almacenamiento de memoria vacías entre las tres variables. Esto es para dar claridad visual. En la práctica real, el compilador de C guarda las tres variables en posiciones de memoria adyacentes sin ningún byte sin usar entre ellas.

Los apuntadores y los arreglos

Los apuntadores pueden ser útiles cuando se está trabajando con variables simples, pero son más útiles con los arreglos. Hay una relación especial en C entre los apuntadores y los arreglos. De hecho, cuando se usa la notación de subíndices de arreglos, que se aprendió en el Día 8, "Arreglos numéricos", en realidad se están usando apuntadores sin saberlo. Los siguientes párrafos explican cómo funciona esto.

El nombre del arreglo como un apuntador

Un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Por lo tanto, si se ha declarado un arreglo `datos[]`, `datos` es la dirección del primer elemento del arreglo.

Tal vez esté pensando "espere un momento, ¿qué no se necesita el operador de *dirección de* para obtener una dirección?". Sí, también se puede usar la expresión `&datos[0]` para obtener la dirección del primer elemento del arreglo. En el C, la relación `(datos == &datos[0])` es cierta.

Ya ha visto que el nombre de un arreglo es un apuntador al arreglo. Recuerde que esto es una *constante de apuntador*; no puede ser cambiada y permanece fija por toda la duración de la ejecución del programa. Esto tiene sentido, si se cambiara su valor apuntaría a cualquier otro lado y no al arreglo (que permanece en una posición fija en memoria).

Sin embargo, se puede declarar una variable de apuntador e inicializarla para que apunte hacia el arreglo. Por ejemplo, el código

```
int arreglo[100], *p_arreglo;
/* aquí va código adicional */
p_arreglo = arreglo;
```

inicializa la variable de apuntador `p_arreglo` con la dirección del primer elemento de `arreglo[]`. Debido a que `p_arreglo` es una variable de apuntador, puede ser modificada para que apunte a cualquier otro lugar. A diferencia de `arreglo`, `p_arreglo` no está atada para que apunte al primer elemento de `arreglo[]`. Podría, por ejemplo, ser apuntada a otro elemento de `arreglo[]`. ¿Cómo se haría eso? Primero, necesitamos ver la manera en que son guardados los elementos del arreglo en la memoria.

Almacenamiento de elementos de arreglo

Como debe recordar de lo que se dijo el Día 8, "Arreglos numéricos", los elementos de un arreglo son guardados en posiciones secuenciales de memoria, estando el primer elemento en la dirección más baja. Los siguientes elementos del arreglo (aquellos que tienen un índice mayor que 0) son guardados en direcciones más altas. Qué tan altas, depende del tipo de datos del arreglo (`char`, `int`, `float`, etc.).

Tomemos un arreglo de tipo `int`. Como se aprendió en el Día 3, "Variables y constantes numéricas", una simple variable `int` puede ocupar dos bytes de memoria. Por lo tanto, cada elemento del arreglo está ubicado dos bytes por encima del elemento anterior, y la dirección de cada elemento del arreglo es mayor en dos que la dirección del elemento anterior. Por otro lado, un tipo `float` puede ocupar cuatro bytes. En un arreglo de tipo `float` cada elemento del arreglo está ubicado cuatro bytes por encima del elemento anterior, y la dirección de cada elemento es mayor en cuatro que la dirección del elemento anterior.

La figura 9.7 ilustra la relación entre el almacenamiento del arreglo y las direcciones para un arreglo `int` de seis elementos y un arreglo `float` de tres elementos.

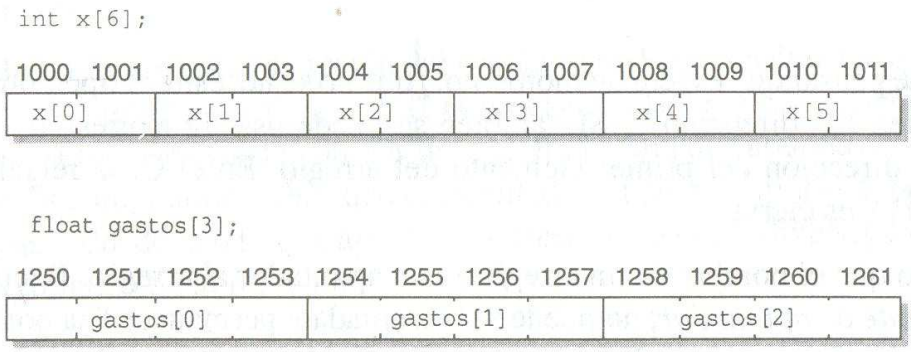


Figura 9.7. *Almacenamiento de arreglos para diferentes tipos de arreglos.*

Viendo la figura 9.7 se debe ser capaz de ver el porqué las siguientes relaciones son ciertas:

```
1: x == 1000
2: &x[0] == 1000
3: &x[1] == 1002
4: gastos == 1250
5: &gastos[0] == 1250
6: &gastos[1] == 1254
```

x sin los corchetes de arreglo es la dirección del primer elemento. En la figura 9.7 se puede ver que `x[0]` tiene la dirección 1000. La línea 2 también muestra esto. Puede leerse como que la dirección del primer elemento del arreglo `x` es igual a 1000. La línea 3 muestra que la dirección del segundo elemento (con subíndice 1 del arreglo) es 1002. Nuevamente la figura puede confirmar esto. Las líneas 4, 5 y 6 son virtualmente idénticas a las 1, 2 y 3,

respectivamente. Varían en la diferencia que hay entre las direcciones de los dos elementos de arreglo. En el arreglo `x`, tipo `int`, la diferencia es de dos bytes, y en el arreglo `gastos`, tipo `float`, la diferencia es de cuatro bytes.

¿Cómo se accesan estos elementos de arreglos sucesivos usando un apuntador? Puede ver en estos ejemplos que un apuntador debe ser aumentado en 2 para accesar los elementos sucesivos de un arreglo tipo `int`, y en 4 para accesar los elementos sucesivos de un arreglo tipo `float`. Se puede generalizar y decir que para accesar elementos sucesivos de un arreglo de un tipo de datos particular, el apuntador debe ser incrementado en `sizeof(tipo_de_dato)`. Recuerde del Día 3, "Variables y constantes numéricas", que el operador `sizeof()` regresa el tamaño en bytes de un tipo de datos del C.

El programa del listado 9.2 ilustra la relación entre las direcciones y los elementos de arreglos de diferente tipo, declarando arreglos de tipo `int`, `float` y `double`, y desplegando las direcciones de sus elementos sucesivos.

Captura

Listado 9.2. Desplegado de las direcciones de elementos sucesivos de arreglo.

```
1: /* Demuestra la relación entre direcciones y */
2: /* elementos de arreglos de diferentes tipos de datos */
3:
4: #include <stdio.h>
5:
6: /* Declara tres arreglos y una variable de contador */
7:
8: int i[10], x;
9: float f[10];
10: double d[10];
11:
12: main()
13: {
14:     /* Imprime el encabezado de la tabla */
15:
16:     printf("\t\tInteger\t\tFloat\t\tDouble");
17:
18:     printf("\n================================");
19:     printf("======================");
20:
21:     /* Imprime las direcciones de cada elemento de arreglo */
22:
23:     for (x = 0; x < 10; x++)
24:         printf("\nElement %d:\t%d\t\t%d\t\t%d", x, &i[x],
25:             &f[x], &d[x]);
26:
27:     printf("\n================================");
28:     printf("======================");
29: }
```

Salida

Aquí se muestra la salida del programa. Las direcciones exactas que despliegue su sistema pueden ser diferentes, pero las relaciones son las mismas: 2 bytes entre elementos `int`, 4 bytes entre elementos `float` y 8 bytes entre elementos `double`. (Nota: Algunas máquinas usan diferentes tamaños para los tipos de variables. Si la máquina es diferente, la siguiente salida puede tener diferentes tamaños de intervalo, sin embargo, los intervalos serán consistentes.)

```
                Integer     Float     Double
=============================================
Element 0:      1392        1414      1454
Element 1:      1394        1418      1462
Element 2:      1396        1422      1470
Element 3:      1398        1426      1478
Element 4:      1400        1430      1486
Element 5:      1402        1434      1494
Element 6:      1404        1438      1502
Element 7:      1406        1442      1510
Element 8:      1408        1446      1518
Element 9:      1410        1450      1526
=============================================
```

Análisis

Este listado aprovecha los caracteres de escape que se aprendieron en el Día 7, "Entrada/salida básica". Las llamadas a `printf()` en las líneas 16 y 24 usan el carácter de escape tab (`\t`), para ayudarse en el formateo de la tabla alineando las columnas.

Viendo más de cerca el listado, se puede observar que son creados tres arreglos en las líneas 8, 9 y 10. La línea 8 declara al arreglo `i` de tipo `int`, la línea 9 declara al arreglo `f` de tipo `float` y la línea 10 declara al arreglo `d` de tipo `double`. La línea 16 imprime los encabezados de columna para la tabla que será desplegada. Las líneas 18 y 19, junto con las líneas 27 y 28, imprimen líneas de guiones en la parte superior e inferior de la tabla de datos. Este es un toque agradable para un informe. Las líneas 23, 24 y 25, son un ciclo `for` que imprime cada uno de los renglones de la tabla. El número del elemento, `x`, es escrito primero. Esto es seguido por la dirección del elemento en cada uno de los tres arreglos.

Aritmética de apuntadores

Se tiene un apuntador al primer elemento del arreglo. El apuntador debe ser incrementado por una cantidad igual al tamaño del tipo de dato guardado en el arreglo. ¿Cómo se accesan los elementos del arreglo usando notación de apuntadores? Con *aritmética de apuntadores*.

Tal vez piense "precisamente lo que no necesito, ¡otro tipo de aritmética que tengo que aprender!" No se preocupe. La aritmética de apuntadores es simple, y facilita el uso de apuntadores en los programas. Sólo se tiene que tratar con dos operaciones de apuntadores: incremento y decremento.

Incremento de apuntadores

Cuando se incrementa un apuntador se está incrementando su valor. Por ejemplo, cuando se incrementa un apuntador en 1, la aritmética de apuntadores incrementa automáticamente el valor del apuntador para que apunte al siguiente elemento del arreglo. Dicho de otra forma, el C sabe el tipo de dato al que apunta el apuntador (a partir de la declaración del apuntador) e incrementa la dirección guardada en el apuntador en el tamaño del tipo de dato.

Supongamos que `p_a_int` es una variable de apuntador a algún elemento de un arreglo `int`. Si se ejecuta el enunciado

```
p_a_int++;
```

el valor de `p_a_int` es aumentado en el tamaño de tipo `int` (que es por lo general, de 2 bytes) y `p_a_int` apunta ahora al siguiente elemento del arreglo. De manera similar, si `p_a_float` apunta a un elemento de un arreglo de tipo `float`, luego

```
p_a_float++;
```

incrementa el valor de `p_a_float` en el tamaño del tipo `float` (que por lo general es de 4 bytes).

Lo mismo se mantiene para incrementos mayores de 1. Si se suma el valor n a un apuntador, el C incrementa el apuntador en n elementos de arreglo del tipo de datos asociado. Por lo tanto,

```
p_a_int += 4;
```

incrementa el valor guardado en `p_a_int` en ocho, para que de esta forma apunte a cuatro elementos de arreglo adelante. De manera similar

```
p_a_float += 10;
```

incrementa el valor guardado en `p_a_float` en 40, para que apunte a 10 elementos de arreglo adelante.

Decremento de apuntadores

Se aplica el mismo concepto para el decremento de un apuntador como para el incremento. El decremento de un apuntador es, de hecho, un caso de incremento sumando un valor *negativo*. Si se decrementa un apuntador con los operadores `--` o `-=`, la aritmética de apuntadores se ajusta automáticamente al tamaño de los elementos del arreglo.

El listado 9.3 presenta un ejemplo sobre la manera en que puede ser usada la aritmética de apuntadores para accesar elementos de un arreglo. Incrementando apuntadores, el programa puede ir paso a paso por todos los elementos del arreglo en forma eficiente.

Captura

Listado 9.3. Uso de la aritmética de apuntadores y de la notación de apuntadores para accesar elementos de arreglo.

```
1: /* Demuestra el uso de aritmética de apuntadores para accesar */
2: /* elementos de arreglo con notación de apuntador */
3:
4: #include <stdio.h>
5: #define MAX 10
6:
7: /* Declara e inicializa un arreglo de enteros */
8:
9: int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9 };
10:
11: /* Declara un apuntador a int y una variable int */
12:
13: int *i_ptr, count;
14:
15: /* Declara e inicializa un arreglo de flotantes */
16:
17: float f_array[MAX] = { .0, .1, .2, .3, .4, .5, .6, .7, .8, .9 };
18:
19: /* Declara un apuntador a float */
20:
21: float *f_ptr;
22:
23: main()
24: {
25:     /* Inicializa los apuntadores */
26:
27:     i_ptr = i_array;
28:     f_ptr = f_array;
29:
30:     /* Imprime los elementos del arreglo */
31:
32:     for (count = 0; count < MAX; count++)
33:         printf("\n%d\t%f", *i_ptr++, *f_ptr++);
34: }
```

Salida

```
0       0.000000
1       0.100000
2       0.200000
3       0.300000
4       0.400000
5       0.500000
6       0.600000
7       0.700000
8       0.800000
9       0.900000
```

Análisis

En este programa, una constante definida llamada `MAX` es puesta a 10 en la línea 5 y es usada por todo el listado. En la línea 9, `MAX` es usada para poner la cantidad de elementos en un arreglo de `int`s llamado `i_array`. Los elementos en este arreglo son inicializados al mismo tiempo que es declarado el arreglo. La línea 13 declara dos variables `int` adicionales. La primera es un apuntador llamado `i_ptr`. Se sabe que es un apuntador, debido a que se usa un operador de indirección (*). La otra variable es una variable simple de tipo `int` llamada `count`. En la línea 17 es definido e inicializado un segundo arreglo. Este arreglo es de tipo `float`, contiene `MAX` valores y es inicializado con valores `float`. La línea 21 declara un apuntador a un `float` llamado `f_ptr`.

La función `main()` se encuentra en las líneas 23 a 34. El programa asigna la dirección inicial de los dos arreglos a los apuntadores de sus respectivos tipos en las líneas 27 y 28. Recuerde que un nombre de arreglo sin el subíndice es lo mismo que la dirección del comienzo del arreglo. Un enunciado `for`, en las líneas 32 y 33, usa la variable `int count` para contar de 0 al valor de `MAX`. Para cada cuenta, la línea 33 hace referencia a los dos apuntadores e imprime sus valores con una llamada a la función `printf()`. El operador de incremento luego incrementa cada uno de los apuntadores para que apunte al siguiente elemento del arreglo, antes de continuar con la siguiente iteración del ciclo `for`.

Tal vez piense que el programa del listado 9.3 podría también haber usado notación de subíndices para los arreglos y evitarnos el manejo de apuntadores. Esto es cierto, y en tareas simples de programación como ésta, el uso de la notación de apuntadores no ofrece ninguna ventaja. Sin embargo, cuando comience a escribir programas más complejos encontrará útil el uso de apuntadores.

Por favor, recuerde que no puede ejecutar operaciones de incremento y decremento sobre constantes de apuntador. (Un nombre de arreglo sin corchetes es una constante de apuntador.) También recuerde que cuando se están manejando apuntadores a elementos de arreglo, el compilador C no lleva cuenta del comienzo y fin del arreglo. Si no se tiene cuidado, se puede incrementar o decrementar el apuntador de tal forma que apunte a cualquier lugar en memoria antes o después del arreglo. Hay algo guardado ahí, pero no es un elemento de arreglo. Se debe llevar cuenta de los apuntadores y hacia dónde están apuntando.

Otras manipulaciones de apuntadores

La única otra operación de aritmética de apuntadores es llamada *diferencia*, que se refiere a la resta de dos apuntadores. Si se tienen dos apuntadores hacia diferentes elementos del mismo arreglo, se les puede restar y encontrar qué tan distantes se encuentran. Nuevamente, la aritmética de apuntadores escala automáticamente la respuesta para que haga referencia a elementos de arreglo. Por lo tanto, si `ptr1` y `ptr2` apuntan a elementos de un arreglo (de cualquier tipo), la expresión

```
ptr1 - ptr2
```

le dice qué tan distantes se encuentran los elementos.

Las comparaciones de apuntadores son válidas solamente entre apuntadores que apunten al mismo arreglo. Bajo estas circunstancias, los operadores relacionales `==`, `!=`, `>`, `<`, `>=` y `<=` funcionan adecuadamente. Los elementos inferiores del arreglo (esto es, los que tienen un subíndice menor) siempre tienen direcciones menores que los elementos superiores del arreglo. Por lo tanto, si `ptr1` y `ptr2` apuntan a elementos del mismo arreglo, la comparación

```
ptr1 < ptr2
```

es cierta si `ptr1` apunta a un miembro anterior del arreglo que al que apunta `ptr2`.

Con esto se tratan todas las operaciones permitidas de apuntadores. Muchas operaciones aritméticas que pueden ser ejecutadas con variables regulares, como la multiplicación o la división, no tienen sentido con los apuntadores. El compilador C no las permite. Por ejemplo, si `ptr` es un apuntador, el enunciado

```
ptr *= 2;
```

genera un mensaje de error. Se pueden hacer un total de seis operaciones con un apuntador, y todas han sido tratadas en este capítulo:

Asignación. Se puede asignar un valor a un apuntador. El valor debe ser una dirección, obtenida con el operador dirección de (&) o a partir de una constante de apuntador (un nombre de arreglo).

Indirección. El operador de indirección (*) le da el valor guardado en la posición apuntada.

dirección de. Se puede usar el operador de dirección de para encontrar la dirección de un apuntador, por lo que se pueden tener apuntadores hacia apuntadores. Esto es un tema avanzado y se trata en el Día 15, "Más sobre apuntadores".

Incremento.

Diferencia.

Comparación. Sólo son válidas con dos apuntadores que apunten al mismo arreglo.

Precauciones con los apuntadores

Cuando se está escribiendo un programa que usa apuntadores se debe evitar un error serio: el uso de un apuntador sin inicializar al lado izquierdo de un enunciado de asignación. Por ejemplo, el enunciado

```
int *ptr;
```

declara a un apuntador de tipo `int`. Este apuntador todavía no está inicializado, por lo que no apunta a ningún lado. Para ser más exactos, no apunta a ningún lado *conocido*. Un

apuntador sin inicializar tiene algún valor, pero simplemente uno no sabe cuál es. En muchos casos es cero. Por lo tanto, si se usa un apuntador sin inicializar en un enunciado de asignación, esto es lo que pasa:

```
*ptr = 12;
```

El valor `12` es asignado a donde esté apuntando `ptr`. Esta dirección puede ser cualquiera en memoria, donde está guardado el sistema operativo o en algún lugar del código de programa. El `12` que es guardado ahí puede sobreescribir alguna información importante, y el resultado puede ir desde un error extraño del programa hasta una caída completa del sistema.

El lado izquierdo de un enunciado de asignación es el lugar más peligroso para usar un apuntador no inicializado. Otros errores, aunque menos serios, pueden también ser resultado del uso de un apuntador sin inicializar en cualquier lado del programa, por lo que asegúrese de que los apuntadores de su programa estén correctamente inicializados antes de usarlos. Se debe hacer esto por uno mismo. ¡El compilador no puede verlo por uno!

DEBE	NO DEBE

NO DEBE Tratar de hacer operaciones matemáticas, como la división, multiplicación y módulo, sobre apuntadores. La suma (incremento) y la resta (diferencia) de apuntadores es aceptable.

NO DEBE Olvidar que restar o sumar a un apuntador incrementa el apuntador basándose en el tamaño del tipo de dato al que apunta, y no por 1 o el número que es sumado (a menos que sea un apuntador a un carácter de un byte).

DEBE Comprender el tipo de variables que tiene la PC. Como podrá ver, se necesita saber el tamaño de las variables cuando se trabaja con apuntadores y memoria.

NO DEBE Tratar de incrementar o decrementar una variable de arreglo. Asigne un apuntador a la dirección inicial del arreglo e increméntela (véase el listado 9.3).

Notación de subíndices de arreglo y apuntadores

Un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Por lo tanto, se puede accesar el primer elemento del arreglo usando el operador de indirección. Si `arreglo[]` es un arreglo declarado, la expresión `*arreglo` es el primer elemento del

arreglo, `*(arreglo + 1)` es el segundo elemento del arreglo, y así sucesivamente. Si se generaliza para el arreglo completo, las siguientes relaciones son ciertas:

```
*(arreglo) == arreglo[0]
*(arreglo + 1) == arreglo[1]
*(arreglo + 2) == arreglo[2]
...
*(arreglo + n) == arreglo[n]
```

Esto ilustra la equivalencia de la notación de subíndices de arreglo y de apuntadores de arreglo. Se puede usar cualquiera de ellas en el programa, ya que el compilador las ve como dos diferentes maneras de accesar los datos del arreglo usando apuntadores.

Paso de arreglos a funciones

Este capítulo ya ha tratado la relación especial que existe en C entre apuntadores y arreglos. Esta relación viene al caso cuando se necesita pasar un arreglo como argumento a una función. La única manera en que se puede pasar un arreglo a una función es por medio de un apuntador.

Como aprendió en el Día 5, "Funciones: lo básico", un argumento es un valor que el programa llamador pasa a una función. Puede ser un `int`, un `float` o cualquier otro tipo de dato simple, pero tiene que ser un solo valor numérico. Puede ser un solo elemento de arreglo, pero no puede ser un arreglo completo.

¿Qué pasa si necesita pasar un arreglo completo a una función? Bueno, se puede tener un apuntador hacia el arreglo, y ese apuntador es un solo valor número (la dirección del primer elemento del arreglo). Si se pasa ese valor a la función, la función "sabe" la dirección del arreglo, y puede accesar los elementos del arreglo usando notación de apuntador.

Sin embargo, considere otro problema. Si se escribe una función que toma a un arreglo como argumento, se quiere que la función sea capaz de manejar arreglos de diferentes tamaños. Por ejemplo, se podría escribir una función que encuentre al elemento más grande en un arreglo de enteros. La función no sería de mucha utilidad si estuviera limitada a tratar con arreglos de tamaño fijo (cantidad de elementos).

¿Cómo hace la función para saber el tamaño del arreglo, cuya dirección se pasa? Recuerde, el valor pasado a una función es un apuntador al primer elemento del arreglo. Puede ser el primero de 10 elementos o el primero de 10,000. Hay dos métodos para permitir que una función "sepa" el tamaño del arreglo.

Se puede indentificar al último elemento del arreglo, guardando en él algún valor especial. Conforme la función procesa al arreglo revisa cada elemento para ver si contiene el valor. Si lo contiene, se ha llegado al final del arreglo. La desventaja de este método es que lo fuerza

a reservar algún valor como el indicador de fin de arreglo, reduciendo la flexibilidad que se tiene para guardar datos reales en el arreglo.

El otro método es más flexible y directo: pase a la función el tamaño del arreglo como argumento. Este puede ser un simple argumento de tipo `int`. Por lo tanto, a la función se le pasan dos argumentos: un apuntador al primer elemento del arreglo, y un entero especificando la cantidad de elementos del arreglo. Este segundo método es usado en este libro.

El listado 9.4 acepta una lista de valores del usuario y los guarda en un arreglo. Luego llama a una función, `largest()`, pasándole el arreglo (tanto el apuntador como el tamaño). La función encuentra el valor más grande en el arreglo y lo regresa al programa llamador.

Captura

Listado 9.4. Demostración del paso de un arreglo a una función.

```
1: /* Paso de un arreglo a una función */
2:
3: #include <stdio.h>
4:
5: #define MAX 10
6:
7: int array[MAX], count;
8:
9: int largest(int x[], int y);
10:
11: main()
12: {
13:     /* Recibe del teclado MAX valores */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:     }
20:
21:     /* Llama a la función y despliega el valor de retorno */
22:
23:     printf("\n\nLargest value = %d", largest(array, MAX));
24: }
25:
26: /* La función largest() regresa el valor más grande */
27: /* de un arreglo entero */
28:
29: int largest(int x[], int y)
30: {
31:     int count, biggest = -12000;
32:
33:     for ( count = 0; count < y; count++)
```

Listado 9.4. continuación

```
34:     {
35:         if (x[count] > biggest)
36:             biggest = x[count];
37:     }
38:
39:     return biggest;
40: }
```

Análisis Hay un prototipo de función en la línea 9 y un encabezado de función en la línea 29 que son idénticos. Ellos dicen

```
int largest(int x[], int y)
```

La mayor parte de esta línea debe serle familiar: `largest()` es una función que regresa un `int` al programa llamador, y su segundo argumento es un `int`, representado por el parámetro `y`. La única cosa nueva es el primer parámetro, `int x[]`, que indica que el primer argumento es un apuntador a tipo `int`, representado por el parámetro `x`. También se podría escribir la declaración de función y el encabezado de la manera siguiente:

```
int largest(int *x, int y);
```

Esto es equivalente a la primera forma, ya que tanto `int x[]` como `int *x` significan "apuntador a `int`". La primera forma puede ser preferible, ya que le recuerda que el parámetro representa un apuntador a un arreglo. Por supuesto que el apuntador no sabe si apunta a un arreglo, pero la función lo usa de esa manera.

Ahora veamos la función `largest()`. Cuando es llamada por primera vez, el parámetro `x` guarda el valor del primer argumento, y es, por lo tanto, un apuntador al primer elemento del arreglo. Se puede usar a `x` en cualquier lugar en que pueda ser usado un apuntador de arreglo. En `largest()` los elementos de arreglo son accesados usando notación de subíndices en las líneas 35 y 36. También se podría haber usado notación de apuntadores, reescribiendo el ciclo `if` para que dijera

```
for ( count = 0; count < y; count++)
{
    if (*(x+count) > biggest)
        biggest = *(x+count);
}
```

El listado 9.5 muestra la otra manera de pasar arreglos a funciones.

Listado 9.5. Una forma alterna para pasar un arreglo a una función.

```
1: /* Paso de un arreglo a una función. Método alterno. */
2:
3: #include <stdio.h>
4:
5: #define MAX 10
6:
7: int array[MAX+1], count;
8:
9: int largest(int x[]);
10:
11: main()
12: {
13:     /* Recibe del teclado MAX valores. */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:
20:         if ( array[count] == 0 )
21:             count = MAX;                /* Terminará el ciclo for */
22:     }
23:     array[MAX] = 0;
24:
25:     /* Llama a la función y despliega el valor de retorno. */
26:
27:     printf("\n\nLargest value = %d", largest(array));
28: }
29:
30: /* La función largest() regresa el valor más grande */
31: /* de un arreglo entero */
32:
33: int largest(int x[])
34: {
35:     int count, biggest = -12000;
36:
37:     for ( count = 0; x[count] != 0; count++)
38:     {
39:         if (x[count] > biggest)
40:             biggest = x[count];
41:     }
42:
43:     return biggest;
44: }
```

Análisis Este programa usa una función `largest()` que tiene la misma funcionalidad que el listado anterior. La diferencia es que solamente es necesario el nombre del arreglo. El ciclo `for` de la línea 37 continúa buscando el valor más grande hasta que encuentra un 0, ya que en ese momento sabe que ha terminado.

Viendo las primeras partes del listado se pueden ver las diferencias entre el listado 9.4 y el 9.5. En primer lugar, en la línea 7 se necesita añadir un elemento adicional al arreglo, para guardar el valor que indica el final. En las líneas 20 y 21 es añadido un enunciado `if`, para ver si el usuario ha tecleado un cero, señalando con esto que ya ha terminado de dar valores. Si se teclea un cero, count es puesto a su valor máximo para que pueda salir del ciclo `for` limpiamente. La línea 23 asegura que el último elemento es un cero, en caso de que el usuario haya dado la máxima cantidad de valores (`MAX`).

Añadiendo los comandos adicionales, cuando se teclean los datos se puede hacer que la función `largest()` funcione con cualquier tamaño de arreglo, mas, sin embargo, hay un pero. ¿Qué pasa si se olvida de poner un cero al final del arreglo? Entonces `largest()` continúa más allá del final del arreglo, comparando valores en memoria hasta que encuentra un cero.

Como puede ver, el pasar un arreglo a una función no tiene gran dificultad. Simplemente se pasa un apuntador al primer elemento del arreglo. En la mayoría de las situaciones, también es necesario pasar la cantidad de elementos del arreglo. En la función el valor del apuntador puede ser usado para accesar los elementos del arreglo, ya sea con notación de subíndice o de apuntador.

Recuerde del Día 5, "Funciones: lo básico", que cuando es pasada una variable simple a una función sólo se pasa una copia del valor de la variable. La función puede usar el valor pero no puede cambiar a la variable original, debido a que no tiene acceso a la variable misma. Cuando se pasa un arreglo a una función las cosas son diferentes. A la función se le pasa la dirección del arreglo, y no simplemente una copia de los valores del arreglo. El código de la función está trabajando con los elementos actuales del arreglo y puede modificar los valores guardados en el arreglo.

Resumen

Este capítulo le presentó a los apuntadores, que son una parte central de la programación en C. Un apuntador es una variable que guarda la dirección de otra variable. Se dice que un apuntador "apunta" a la variable cuya dirección guarda. Los dos operadores necesarios con los apuntadores son el operador de *dirección de* (&) y el operador de indirección (*). Cuando es puesto antes de un nombre de variable, el operador de *dirección de* regresa la dirección de la variable. Cuando es puesto antes de un nombre de apuntador, el operador de indirección regresa la dirección de la variable apuntada.

Los apuntadores y los arreglos tienen una relación especial. Un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Las características especiales de

la artitmética de apuntadores facilitan el acceso a los elementos del arreglo usando apuntadores. La notación de subíndices de arreglo es, de hecho, una forma especial de notación de apuntadores.

También aprendió la manera de pasar arreglos como argumentos a funciones, pasando un apuntador hacia el arreglo. Una vez que la función "sabe" la dirección del arreglo y su longitud, puede accesar los elementos del arreglo usando notación de apuntadores o notación de subíndices.

Preguntas y respuestas

1. ¿Por qué son tan importantes los apuntadores en C?

   Los apuntadores le dan mayor control sobre la computadora y los datos. Cuando se usan con funciones, los apuntadores le permiten cambiar el valor de variables que fueron pasadas sin tomar en cuenta dónde han sido originadas. En el Día 15, "Más sobre apuntadores", aprenderá usos adicionales de los apuntadores.

2. ¿Cómo sabe el compilador la diferencia entre * para multiplicación, * para referencia y * para la declaración de un apuntador?

   El compilador interpreta los usos diferentes del asterisco, basándose en el contexto donde es usado. Si el enunciado que está evaluando comienza con un tipo de variable, puede suponerse que el asterisco es para declarar un apuntador. Si el asterisco es usado con una variable que ha sido declarada como apuntador, pero no en una declaración de variable, el asterisco es interpretado como referencia. Si es usado en una expresión matemática pero no con una variable de apuntador, se puede suponer que el asterisco corresponde al operador de multiplicación.

3. ¿Qué pasa si uso el operador de *dirección de* sobre un apuntador?

   Se obtiene la dirección de la variable de apuntador. Recuerde que un apuntador es simplemente otra variable, que guarda la dirección de la variable a la que apunta.

4. ¿Son guardadas las variables siempre en la misma dirección?

   No. Cada vez que se ejecuta un programa sus variables pueden ser guardadas en diferentes direcciones. Nunca se debe asignar un valor constante de dirección a un apuntador.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Qué operador se usa para determinar la dirección de una variable?
2. ¿Qué operador se usa para determinar el valor de la dirección apuntada por un apuntador?
3. ¿Qué es un apuntador?
4. ¿Qué es indirección?
5. ¿Cómo son guardados en memoria los elementos de un arreglo?
6. Muestre dos maneras para obtener la dirección del primer elemento del arreglo `datos[]`.
7. Si se le pasa un arreglo a una función, ¿cuáles son las dos maneras de saber dónde se encuentra el fin del arreglo?
8. ¿Cuáles son las seis operacionse tratadas en este capítulo que pueden realizarse con un apuntador?
9. Supongamos que tiene dos apuntadores. Si el primero apunta al tercer elemento de un arreglo de `int` y el segundo apunta al cuarto elemento, ¿qué valor se obtiene si se resta el primer apuntador del segundo?
10. Supongamos que el arreglo de la pregunta anterior es de valores `float`. ¿Qué valor se obtiene si se restan los dos apuntadores?

Ejercicios

1. Haga una declaración de un apuntador a una variable tipo `char`. Llame al apuntador `char_ptr`.
2. Si se tiene una variable tipo `int` llamada `costo`, ¿cómo declararía e inicializaría un apuntador llamado `p_costo` que apuntara a esa variable?
3. Continuando con el ejercicio 2, ¿cómo asignaría el valor 100 a la variable `costo` usando acceso directo y acceso indirecto?
4. Continuando con el ejercicio 3, ¿cómo imprimiría el valor del apuntador y el valor al que está apuntando?
5. Muestre la manera de asignar la dirección de un valor `float` llamado radio a un apuntador.
6. Muestre dos maneras de asignar el valor de 100 al tercer elemento de `datos[]`.

7. Escriba una función, llamada `totarreglo()`, que acepte dos arreglos como argumentos, obtenga el total de todos los valores de ambos arreglos y regrese el total al programa llamador.

8. Use la función creada en el ejercicio siete en un programa simple.

9. Escriba una función, llamada `suma_arreglo()`, que acepte dos arreglos que sean del mismo tamaño. La función debe sumar los elementos correspondientes de los arreglos y dejar los valores en un tercer arreglo.

10. Modifique la función del ejercicio 9 para que regrese un apuntador al arreglo que contiene los totales. Ponga esta función en un programa que también imprima los valores de los tres arreglos.

DIA 10

Caracteres y cadenas

Un *carácter* es una simple letra, número, signo de puntuación o cualquier otro símbolo. Una *cadena* es cualquier secuencia de caracteres. Las cadenas se usan para guardar datos de texto, que constan de letras, números, signos de puntuación y otros símbolos. No hay duda de que los caracteres y las cadenas son muy útiles en muchas aplicaciones de programación. Hoy aprenderá

- Cómo usar el tipo de dato `char` del C para guardar caracteres solos.
- Cómo crear arreglos de tipo `char` para guardar cadenas de varios caracteres.
- Cómo inicializar caracteres y cadenas.
- Cómo usar apuntadores con cadenas.
- Cómo imprimir y capturar caracteres y cadenas.

El tipo de dato *char*

El C usa el tipo de dato `char` para guardar caracteres. Ya vio en el Día 3, "Variables y constantes numéricas", que `char` es uno de los tipos de datos numéricos enteros del C. Si `char` es un tipo numérico, ¿cómo puede ser usado para guardar caracteres?

La respuesta se encuentra en la manera en que el C guarda los caracteres. La memoria de la computadora guarda todos los datos en forma numérica. No hay una manera directa de guardar caracteres. Sin embargo, existe un código numérico para cada carácter. Este se llama el *código ASCII* o el juego de caracteres ASCII. (ASCII es la abreviatura de American Standard Code for Information Interchange.) El código asigna valores, que van de 0 a 255, para las letras mayúsculas y minúsculas, los dígitos numéricos, los signos de puntuación y otros símbolos. El juego de caracteres ASCII se lista en el apéndice A, "Tabla de caracteres ASCII".

Por ejemplo, `97` es el código ASCII para la letra `a`. Cuando se guarda el carácter `a` en una variable tipo `char`, en realidad se está guardando el valor `97`. Como el rango numérico permitido para el tipo `char` corresponde al juego de caracteres ASCII estándar, `char` se presta adecuadamente para guardar caracteres.

Tal vez en este momento se sienta un poco confundido. Si el C guarda caracteres como números, ¿cómo sabe el programa cuando una variable tipo `char` es un carácter o un número? Como aprenderá más adelante, declarar una variable como tipo `char` no es suficiente, sino que se debe hacer algo más con la variable.

- Si se usa una variable `char` en alguna parte de un programa en C donde se espera un carácter, es interpretada como carácter.
- Si se usa una variable `char` en alguna parte de un programa en C donde se espera un número, es interpretada como número.

Esto le da una idea de la manera en que el C usa el tipo de datos numéricos para guardar datos de carácter. Ahora podemos pasar a los detalles.

Uso de variables de carácter

De manera similar a las demás variables, se debe declarar a las de tipo `char` antes de usarlas, y se les puede inicializar al momento de la declaración. A continuación se presentan algunos ejemplos:

```
char a, b, c;          /* Declara tres variables char sin inicializar */
char codigo = 'x';     /* Declara la variable char llamada código     */
...                    /* y guarda ahí el carácter x                  */
codigo = '!';          /* Guarda ! en la variable llamada código      */
```

Para crear constantes literales de carácter encierre un solo carácter entre comillas simples. El compilador traduce automáticamente las constantes literales de carácter a sus códigos ASCII correspondientes, y el valor numérico del código es asignado a la variable.

Se pueden crear constantes de carácter simbólicas usando la directiva `#define` o la palabra clave `const`:

```
#define EX 'x'
char codigo = EX;       /* Hace que código sea igual a 'x' */
const char A = 'Z';
```

Ahora que ya sabe cómo declarar e inicializar variables de carácter, es tiempo de una demostración. El programa del listado 10.1 ilustra la naturaleza numérica del almacenamiento de caracteres, usando la función `printf()` que se aprendió en el Día 7, "Entrada/salida básica". La función `printf()` puede ser usada para imprimir caracteres y números. El formato `%c` le da instrucciones a `printf()` para que imprima un carácter, y el formato `%d` le da instrucciones de que imprima un entero decimal. El listado 10.1 inicializa dos variables tipo `char` e imprime cada una, primero como carácter y luego como número.

Captura

Listado 10.1. Demostración de la naturaleza numérica de las variables tipo `char`.

```
1:  /* Demuestra la naturaleza numérica de las variables char*/
2:
3:  #include <stdio.h>
4:
5:  /* Declara e inicializa dos variables char */
6:
7:  char c1 = "a";
8:  char c2 = 90;
9:
10: main()
11: {
```

Listado 10.1. continuación

```
12:     /* Imprime la variable c1 como un carácter y luego como un número */
13:
14:     printf("\nAs a character, variable c1 is %c", c1);
15:     printf("\nAs a number, variable c1 is %d", c1);
16:
17:     /* Hace lo mismo para la variable c2 */
18:
19:     printf("\nAs a character, variable c2 is %c", c2);
20:     printf("\nAs a number, variable c2 is %d", c2);
21: }
```

Salida

```
As a character, variable c1 is a
As a number, variable c1 is 97
As a character, variable c2 is Z
As a number, variable c2 is 90
```

Análisis

Se aprendió en el Día 3, "Variables y constantes numéricas", que el rango permitido para las variables de tipo `char` llega solamente hasta 127, aunque el código ASCII va hasta 255. Los códigos ASCII están, de hecho, divididos en dos partes. Los códigos ASCII estándares van hasta 127; este rango incluye todas las letras, números, signos de puntuación y otros símbolos del teclado. Los códigos del 128 al 255 son los códigos ASCII extendidos, y representan caracteres especiales, como letras extranjeras y símbolos gráficos (véase el apéndice A, "Tabla de caracteres ASCII", para una lista completa). Por lo tanto, para los datos de texto estándar se pueden usar variables tipo `char`. Si se quieren imprimir los caracteres ASCII extendidos, se debe usar `unsigned char`.

El programa del listado 10.2 muestra la impresión de algunos de los caracteres ASCII extendidos.

Captura

Listado 10.2. Impresión de caracteres ASCII extendidos.

```
1:  /* Demuestra la impresión de caracteres ASCII extendidos */
2:
3:  #include <stdio.h>
4:
5:  unsigned char x;    /* Debe ser unsigned para el ASCII extendido */
6:
7:  main()
8:  {
9:      /* Imprime caracteres ASCII extendidos del 180 al 203 */
10:
11:     for (x = 180; x < 204; x++)
12:     {
13:         printf("\nASCII code %d is character %c", x, x);
14:     }
15: }
```

Análisis Con este programa se ve que la línea 5 declara una variable de carácter sin signo, `unsigned x`. Esto nos da un rango de 0 a 255. De manera similar a otros tipos de datos numéricos, no se debe inicializar una variable `char` a un valor que se encuentra fuera del rango permitido, ya que de hacerlo así se obtendrán resultados impredecibles. En la línea 11, `x` no se inicializa fuera de rango, sino que, en vez de ello, se inicializa a 180. En el enunciado `for`, `x` se incrementa en 1 hasta que llega a 204. Cada vez que se incrementa `x`, la línea 13 imprime el valor de `x` y del valor del carácter de `x`. Recuerde que `%c` imprime el valor del carácter, o ASCII, de `x`.

DEBE	NO DEBE
DEBE Usar %c para imprimir el valor de carácter de un número.	
	NO DEBE Usar comillas dobles cuando inicialice una variable de carácter.

DEBE	NO DEBE
DEBE Usar comillas simples cuando inicialice una variable.	
	NO DEBE Tratar de poner valores de caracteres ASCII extendidos en una variable de tipo `char` con signo.
DEBE Estudiar la tabla ASCII del apéndice A para ver los caracteres interesantes que pueden ser impresos.	

Uso de cadenas

Las variables de tipo `char` pueden guardar un solo carácter únicamente, por lo que tienen una utilidad limitada. También se necesita una manera de guardar *cadenas*, que son una secuencia de caracteres. El nombre de una persona o una dirección son ejemplos de cadenas. Aunque no hay un tipo de datos especial para las cadenas, el C maneja este tipo de información con arreglos de caracteres.

Arreglos de caracteres

Por ejemplo, para guardar una cadena de seis caracteres se necesita declarar un arreglo de tipo `char` con siete elementos. Los arreglos de tipo `char` son declarados de manera similar a los arreglos de otros tipos de datos. Por ejemplo, el enunciado

```
char cadena[10];
```

declara un arreglo de 10 elementos de tipo `char`. Este arreglo pudiera ser usado para guardar una cadena de nueve o menos caracteres.

Tal vez esté pensando, "espere un momento, si es un arreglo de 10 elementos, ¿por qué puede guardar solamente nueve caracteres?". En C, una cadena está definida como una secuencia de caracteres que termina con el carácter nulo, un carácter especial representado por `\0`. Aunque es representado por dos caracteres (diagonal inversa-cero) el carácter nulo es interpretado como un solo carácter, y tiene el valor ASCII de `0`. Es una de las secuencias de escape del C que han sido tratadas el Día 7, "Entrada/salida básica".

Cuando un programa en C guarda, por ejemplo, la palabra `Veracruz`, de hecho, guarda los ocho caracteres `V`, `e`, `r`, `a`, `c`, `r`, `u` y `z` seguidos del carácter nulo, `\0`, haciendo un total de nueve caracteres. Por lo tanto, un arreglo de caracteres puede guardar una cadena de caracteres que sea menor en uno que el número total de elementos del arreglo.

Una variable de tipo `char` es de un byte de tamaño, por lo que la cantidad de bytes en las variables de arreglo tipo `char` es la misma que la cantidad de elementos del arreglo.

Inicialización de arreglos de caracteres

De manera similar a otros tipos de datos del C, los arreglos de caracteres pueden ser inicializados cuando son declarados. Se puede asignar valor, elemento por elemento, a los arreglos de caracteres, como se muestra aquí:

```
char cadena[10] = { 'V', 'e', 'r', 'a', 'c', 'r', 'u', 'z', '\0' };
```

Sin embargo, es más conveniente usar una *cadena literal,* que es una secuencia de caracteres encerrada entre comillas dobles:

```
char cadena[10] = "Veracruz";
```

Cuando se usa una cadena literal en el programa, el compilador añade automáticamente el carácter nulo de terminación al final de la cadena. Si no se especifica el número del subíndice cuando se declara al arreglo, el compilador calcula el tamaño del arreglo. Por lo tanto,

```
char cadena[] = "Veracruz";
```

crea e inicializa un arreglo de nueve elementos.

Recuerde que las cadenas requieren el carácter nulo de terminación. Las funciones del C que manejan las cadenas (tratadas en el Día 17, "Manipulación de cadenas") determinan la longitud de la cadena buscando el carácter nulo. Las funciones no tienen otra manera de reconocer el final de la cadena. Si falta el carácter nulo, el programa piensa que la cadena se extiende hasta donde aparezca en memoria el siguiente carácter nulo. A causa de este error pueden resultar molestos errores de programa.

Cadenas y apuntadores

Ya ha visto que las cadenas son guardadas en arreglos tipo `char`, con el final de la cadena marcado por el carácter nulo (ya que probablemente la cadena no ocupa el arreglo completo). Debido a que el final de la cadena se encuentra marcado, todo lo que se necesita para definir una cadena dada es algo que apunte a su inicio. (¿Es *apuntar* la palabra correcta? ¡Claro que sí!)

Con esta pista tal vez se esté adelantando. Por lo que se dijo en el Día 9, "Apuntadores", ya sabe que el nombre de un arreglo es un apuntador al primer elemento del arreglo. Por lo tanto, para una cadena que se encuentra guardada en un arreglo sólo necesita el nombre del arreglo para poder accesarlo. De hecho, el uso del nombre del arreglo es el método estándar del C para accesar cadenas.

Para ser más precisos, el uso del nombre del arreglo para accesar cadenas es el método esperado por las funciones de biblioteca del C. La biblioteca estándar del C incluye varias funciones que manejan cadenas. (Estas funciones se tratan en el Día 17, "Manipulación de cadenas".) Para pasar una cadena a alguna de estas funciones se pasa el nombre del arreglo. Lo mismo se aplica para las funciones de desplegado de cadenas `printf()` y `puts()`, tratadas posteriormente, en este capítulo.

Tal vez se haya dado cuenta de que se usa la frase "las cadenas son guardadas en arreglos". ¿Implica esto que algunas cadenas no son guardadas en arreglos? Así es, y la siguiente sección explica cómo.

Cadenas sin arreglos

En la sección anterior se dijo que una cadena está definida por el nombre del arreglo de carácter, que es un apuntador tipo `char` que apunta al inicio de la cadena, y por un carácter nulo, que indica su fin, y que el espacio que ocupa la cadena en el arreglo es banal. De hecho, para lo único que sirve el arreglo es para proporcionar espacio a la cadena.

¿Qué tal si se encontrara algún otro espacio de almacenamiento de memoria sin asignar un arreglo? Se podría entonces guardar ahí una cadena con su carácter nulo de terminación. Un apuntador al primer carácter podría servir para especificar el comienzo de la cadena, de manera similar a que si la cadena estuviera en un arreglo asignado. ¿Cómo se haría para encontrar espacio de almacenamiento de memoria? Hay dos métodos: uno asigna espacio para un texto literal cuando el programa es compilado, y el otro usa la función `malloc()` para asignar espacio mientras el programa está ejecutando (un proceso llamado asignación dinámica).

Asignación de espacio para la cadena en la compilación

El comienzo de una cadena, como se dijo anteriormente, está indicado por un apuntador a una variable de tipo `char`. Sería bueno recordar la manera en que se declara este tipo de apuntador:

```
char *mensaje;
```

Este enunciado declara un apuntador a una variable de tipo `char` llamado `mensaje`. En este momento no apunta a nada, pero qué tal si cambiamos la declaración de apuntador para que diga:

```
char *mensaje = "¡Qué buen día!";
```

Cuando este enunciado ejecuta, la cadena `¡Qué buen día!` (con su carácter nulo de terminación) es guardada en algún lugar de memoria, y el apuntador `mensaje` es inicializado para que apunte al primer carácter de la cadena. No hay que preocuparse por el lugar de memoria donde se encuentra guardada la cadena, ya que esto es manejado automáticamente por el compilador. Una vez que ha sido definido, `mensaje` es un apuntador a la cadena y puede ser utilizado como tal.

La declaración/inicialización anterior es equivalente a:

```
char mensaje[] = "¡Qué buen día!";
```

y las dos notaciones `*mensaje` y `mensaje[]` son equivalentes. Ambas significan "un apuntador a".

Este método de asignación de espacio para el almacenamiento de cadenas es conveniente cuando se sabe lo que se necesita mientras se está escribiendo el programa. ¿Qué pasaría si el programa tuviera necesidades variables de almacenamiento de cadenas, dependiendo de los datos del usuario o de otros factores que son desconocidos al momento de escribir el programa? Se usa la función `malloc()`, que le permite asignar espacio de almacenamiento "al vuelo".

La función *malloc()*

`malloc()` es una de las funciones de asignación de memoria del C (acrónimo de *m*emory *alloc*ation). Cuando se usa `malloc()` se le pasa la cantidad de bytes de memoria que se necesita. `malloc()` encuentra y reserva un bloque de memoria del tamaño pedido y regresa la dirección del primer byte del bloque. No hay por qué preocuparse sobre qué parte de memoria se usa, ya que esto es manejado automáticamente.

La función `malloc()` regresa una dirección, y su tipo de retorno es un apuntador a tipo `void`. ¿Por qué `void`? Un apuntador a tipo `void` es compatible con todos los tipos de datos. Como la memoria asignada por `malloc()` puede ser usada para guardar cualquiera de los tipos de datos del C, es adecuado el tipo de retorno `void`.

Sintaxis

La función *malloc()*

```
#include <stdlib.h>
void *malloc(tipo_del_tamaño tamaño);
```

`malloc()` asigna un bloque de memoria de la cantidad de bytes indicada en tamaño. Usando `malloc()` para asignar memoria en el momento en que se necesita, en vez de asignarla toda cuando el programa se inicia, se puede usar más eficientemente la memoria de la computadora. Cuando se usa `malloc()` se necesita incluir el archivo de encabezado STDLIB.H. Algunos compiladores tienen otros archivos de encabezado que pueden ser incluidos, mas, sin embargo, por compatibilidad, es mejor incluir el STDLIB.H.

`malloc()` regresa un apuntador al bloque de memoria asignado. Si `malloc()` no fue capaz de asignar la cantidad de memoria solicitada, regresa nulo. Cada vez que trate de asignar memoria debe revisar el valor de retorno, aunque la cantidad de memoria por asignar sea pequeña.

Ejemplo 1

```
#include <stdlib.h>
main()
{
   /* asigna memoria para una cadena de 100 caracteres */
   char *string;
   if (( str = (char *) malloc(100)) == NULL)
   {
  printf( "No hay suficiente memoria para asignar buffer\n");
      exit(1);
   }
   printf( "¡La cadena fue asignada!" );
   return 0;
}
```

Ejemplo 2

```
/* asigna memoria para un arreglo de 50 enteros */
int *numbers;
numbers = (int *) malloc(50 * sizeof(int));
```

Ejemplo 3

```
/* asigna memoria para un arreglo de 10 valores float */
float *numbers;
numbers = (float *) malloc(10 * sizeof(float));
```

Se puede usar `malloc()` para asignar memoria que almacene un solo espacio tipo `char`. Primero declare un apuntador a tipo `char`:

```
char *ptr;
```

Luego, llame a `malloc()` y pase el tamaño del bloque de memoria deseado. Como el tipo `char` ocupa un solo byte, se necesita un bloque de un byte. El valor regresado por `malloc()` es asignado al apuntador.

```
ptr = malloc(1);
```

Este enunciado asigna un bloque de memoria de un byte, y asigna su dirección a `ptr`. A diferencia de las variables que son declaradas en el programa, este byte de memoria no tiene nombre. Solamente el apuntador puede hacer referencia a la variable. Por ejemplo, para guardar ahí el carácter `'x'` se podría escribir

```
*ptr = 'x';
```

La asignación de espacio de almacenamiento con `malloc()` para una cadena es casi idéntica al uso de `malloc()` para asignar espacio para una sola variable de tipo `char`. La principal diferencia es que se necesita saber la cantidad de espacio por asignar, es decir, la cantidad máxima de caracteres de la cadena. Este máximo depende de las necesidades del programa. Para este ejemplo, digamos que se quiere asignar espacio para una cadena de 99 caracteres, más 1 para el carácter nulo terminal, dando un total de 100. Primero se declara un apuntador a tipo `char` y luego se llama a `malloc()`.

```
char *ptr;
ptr = malloc(100);
```

Ahora `ptr` apunta al bloque reservado de 100 bytes, que puede ser usado para el almacenamiento y manejo de la cadena. Se puede usar `ptr` de manera similar a si el programa hubiera asignado explícitamente ese espacio con la siguiente declaración de arreglo.

```
char ptr[100];
```

El uso de `malloc()` le permite al programa asignar el espacio de almacenamiento conforme se necesita, en respuesta a los requerimientos. Por supuesto que el espacio disponible no es ilimitado, ya que depende de la cantidad de memoria instalada en la computadora y de los requerimientos de almacenamiento de otros programas. Si no se tiene suficiente memoria disponible, `malloc()` regresa `0 (nulo)`. El programa debe revisar el valor de retorno de `malloc()`, para estar seguro de que la memoria solicitada fue asignada satisfactoriamente. Siempre se debe comparar el valor de retorno de `malloc()` contra la constante simbólica `NULL`, definida en STDLIB.H. El listado 10.3 ilustra el uso de `malloc()`. Cualquier programa que use a `malloc()` debe incluir con `#include` el archivo de encabezado STDLIB.H.



Listado 10.3. Uso de la función `malloc()` para asignar espacio de almacenamiento para datos de cadena.

```
1: /* Demuestra el uso de malloc() para asignar espacio de */
2: /* almacenamiento para datos de cadena */
3:
4: #include <stdio.h>
```

```
5: #include <stdlib.h>
6:
7: char count, *ptr, *p;
8:
9: main()
10: {
11:     /* Asigna un bloque de 35 bytes. Revisa si se efectuó. */
12:     /* La función de biblioteca exit() termina al programa. */
13:
14:     ptr = malloc(35 * sizeof(char));
15:
16:     if ( ptr == NULL )
17:     {
18:         puts("Memory allocation error.");
19:         exit(1);
20:     }
21:
22:     /* Llena la cadena con valores del 65 al 90, */
23:     /* que son los códigos ASCII de la A a la Z. */
24:
25:     /* p es un apuntador usado para desplazarse por la cadena. */
26:     /* Se desea que ptr siga apuntando al */
27:     /* comienzo de la cadena */
28:
29:     p = ptr;
30:
31:     for (count = 65; count < 91 ; count++)
32:         *p++ = count;
33:
34:     /* Añade el carácter nulo de terminación */
35:
36:     *p = "\0";
37:
38:     /* Despliega la cadena en la pantalla. */
39:
40:     puts(ptr);
41: }
```

Análisis Este programa usa a `malloc()` de manera sencilla. Aunque este programa se ve largo, está lleno de comentarios. Las líneas 1, 2, 11, 12, 22 a 27, 34 y 38, son comentarios que detallan todo lo que hace el programa. La línea 5 incluye el archivo de encabezado STDLIB.H, necesario para `malloc()`, y la línea 4 incluye el archivo de encabezado STDIO.H para la función `puts()`. La línea 7 declara dos apuntadores y una variable de carácter, usada posteriormente en el listado. Ninguna de estas variables se encuentra inicializada, por lo que no deben usarse (¡todavía!).

La función `malloc()` es llamada en la línea 14 con un parámetro de 35 multiplicado por *el tamaño de* un `char`, obtenido mediante el operador `sizeof()`. ¿No se podría haber puesto simplemente 35? Sí, pero se estaría haciendo la suposición de que cualquiera que ejecute este

programa estaría usando una computadora que guarda las variables tipo `char` en un byte. Recuerde del Día 3, "Variables y constantes numéricas", que diferentes compiladores pueden usar variables de diferente tamaño. Usar el operador `sizeof()` es una manera fácil de crear código portable.

Nunca suponga que `malloc()` obtiene la memoria que se le dice que obtenga. De hecho, no se le está diciendo que obtenga memoria, sino que se le está pidiendo. La línea 16 muestra la manera fácil de revisar para ver si `malloc()` proporcionó la memoria. Si la memoria fue asignada, `ptr` apunta a ella y, en caso contrario, `ptr` es nulo. Si el programa falla al tratar de conseguir memoria, las líneas 18 y 19 despliegan un mensaje de error y terminan el programa.

La línea 29 inicializa el otro apuntador declarado en la línea 7, `p`. A él se le asigna el mismo valor de dirección que `ptr`. Un ciclo `for` usa este nuevo apuntador para poner valores en la memoria asignada. En la línea 31 se ve que `count` es inicializado a `65` e incrementado en `1` hasta que llega a `91`. Para cada ciclo del enunciado `for`, el valor de `count` es asignado a la dirección apuntada por `p`. Observe que cada vez que `count` se incrementa también se incrementa la dirección a la que apunta `p`. Esto significa que, en memoria, cada valor es puesto uno tras otro.

Ya debe haberse dado cuenta de que están siendo asignados números a `count`, que es una variable tipo `char`. No olvide la discusión acerca de los caracteres ASCII y sus equivalentes numéricos. El número 65 es equivalente a `A`, 66 = `B`, 67 = `C`, etc. El ciclo `for` termina después de que el alfabeto es asignado a las posiciones de memoria apuntadas. La línea 36 termina los valores de carácter apuntados, poniendo un nulo en la dirección final a la que apunta `p`. Añadiendo el nulo ahora se pueden usar estos valores como una cadena. Recuerde que `ptr` todavía apunta al primer valor, A, por lo que si lo usa como una cadena imprime cada uno de los caracteres hasta que llega al nulo. La línea 40 usa a `puts()` para probar este punto y para mostrar el resultado de lo que se ha hecho.

DEBE NO DEBE

NO DEBE Asignar más memoria de la que necesita. No todos tienen gran cantidad de memoria, por lo que debe tratar de usarla con mesura.

NO DEBE Tratar de asignar una cadena nueva a un arreglo de caracteres que fue asignado anteriormente sólo con la memoria suficiente para contener una cadena más pequeña. Por ejemplo,

```
char una_cadena[] = "MAL";
```

En esta declaración una_cadena apunta a "MAL". Si se trata de asignar "BIEN" a este arreglo, se pueden tener serios problemas. El arreglo inicialmente puede contener solamente cuatro caracteres, 'M', 'A', 'L', y un nulo. "BIEN" es de cinco caracteres, 'B', 'I', 'E', 'N', y un nulo. No se puede saber lo que el quinto carácter, el nulo, sobreescribe.

Desplegado de cadenas y caracteres

Si el programa usa datos de cadena, probablemente necesita desplegar los datos en la pantalla en algún momento. El desplegado de cadena se hace, por lo general, con las funciones `puts()` o `printf()`.

La función *puts()*

Ya ha visto la función de biblioteca `puts()` en algunos de los programas dados en este libro. La función `puts()` pone una cadena en la pantalla, y a esto se debe su nombre. El único argumento que toma `puts()` es un apuntador a la cadena que ha de ser desplegada. Debido a que una cadena literal evalúa como un apuntador a una cadena, `puts()` puede ser usado para desplegar cadenas literales y variables de cadena. La función `puts()` inserta automáticamente un carácter de nueva línea al final de cada cadena que despliega, por lo que cada cadena subsecuente desplegada con `puts()` se encuentra en su propia línea.

El programa del listado 10.4 ilustra el uso de `puts()`.

Captura

Listado 10.4. Uso de la función puts() para desplegar texto en la pantalla.

```
1: /* Demuestra el desplegado de cadenas con puts(). */
2:
3: #include <stdio.h>
4:
5: char *message1 = "C";
6: char *message2 = "is the";
7: char *message3 = "best";
8: char *message4 = "programming";
9: char *message5 = "language!!";
10:
11: main()
12: {
13:     puts(message1);
14:     puts(message2);
15:     puts(message3);
16:     puts(message4);
17:     puts(message5);
18: }
```

Salida

```
C
is the
best
programming
language!!
```

Análisis

Este es un listado bastante simple de seguir. Debido a que `puts()` es una función estándar de entrada/salida, "se necesita" incluir el archivo de encabezado STDIO.H, como se hace en la línea 3. Las líneas 5 a 9 declaran e inicializan cinco variables de mensaje diferentes. Cada una de estas variables es un apuntador a carácter o una variable de cadena. Las líneas 13 a 17 usan la función `puts()` para imprimir cada cadena.

La función *printf()*

También puede desplegar cadenas con la función de biblioteca `printf()`. Recuerde del Día 7, "Entrada/salida básica", que `printf()` usa un formato y especificadores de conversión para darle forma a su salida. Para desplegar una cadena use el especificador de conversión `%s`.

Cuando `printf()` encuentra un %s en su formato, la función aparea al `%s` con el argumento correspondiente de su lista de argumentos. Cuando se trata de una cadena, este argumento debe ser un apuntador a la cadena que se quiere desplegar. La función `printf()` despliega la cadena en la pantalla, deteniéndose cuando llega al carácter nulo terminal de la cadena. Por ejemplo,

```
char *cad = "Un mensaje por desplegar";
printf("%s", cad);
```

También se pueden desplegar varias cadenas y mezclarlas con texto literal y/o variables numéricas.

```
char *banco = "Banco S.A.";
char *nombre = "Juan Pérez";
int saldo = 1000;
printf("El saldo en %s de %s es %d. ", banco, nombre, saldo);
```

La salida resultante es

```
El saldo en Banco S.A. de Juan Pérez es 1000.
```

Por ahora esta información debe serle suficiente para que sea capaz de desplegar datos de cadena en los programas. En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se dan detalles completos sobre el uso de `printf()`.

Lectura de cadenas desde el teclado

Además de desplegar cadenas, los programas necesitan frecuentemente aceptar la entrada de datos de cadenas del usuario por medio del teclado. La biblioteca del C tiene dos funciones que pueden usarse para este objetivo, `gets()` y `scanf()`. Sin embargo, antes de que pueda leer una cadena del teclado debe tener algún lugar donde ponerla. El espacio para guardar la cadena puede ser creado con cualquiera de los métodos tratados anteriormente, el día de hoy, una declaración de arreglo o la función `malloc()`.

Entrada de cadenas con la función *gets()*

La función `gets()` obtiene una cadena del teclado. Cuando se llama a `gets()`, ésta lee todos los caracteres tecleados hasta que encuentra el primer carácter de nueva línea (que se genera cuando se oprime Enter). La función desecha a la nueva línea, añade un carácter nulo y le da la cadena al programa llamador. La cadena es guardada en la posición indicada por un apuntador de tipo `char` que ha sido pasado a `gets()`. Un programa que usa `gets()` debe incluir con `#include` el archivo STDIO.H. El listado 10.5 presenta un ejemplo.

Captura

Listado 10.5. Uso de `gets()` para aceptar datos de cadena del teclado.

```
1: /* Demuestra el uso de la función de biblioteca gets() */
2:
3: #include <stdio.h>
4:
5: /* Asigna un arreglo de caracteres para guardar la entrada */
6:
7: char input[81];
8:
9: main()
10: {
11:     puts("Enter some text, then press Enter");
12:     gets(input);
13:     printf("You entered %s", input);
14: }
```

Análisis

En este ejemplo, el argumento para `gets()` es la expresión `input`, que es el nombre de un arreglo tipo `char` y, por lo tanto, un apuntador al primer elemento del arreglo. El arreglo fue declarado de 81 elementos en la línea 7. Debido a que la longitud de línea máxima posible en la mayoría de las pantallas de computadora es de 80 caracteres, este tamaño de arreglo proporciona espacio para la línea de entrada más larga posible (más el carácter nulo que `gets()` añade al final).

La función `gets()` tiene un valor de retorno que fue ignorado en el ejemplo anterior. `gets()` regresa un apuntador a tipo `char` con la dirección donde fue guardada la cadena de entrada. Sí, es el mismo valor que le fue pasado a `gets()`, pero el tener de esta manera el valor regresado le permite al programa revisar si se trata de una línea en blanco. El listado 10.6 muestra la manera de hacer esto.

Captura

Listado 10.6. Uso del valor de retorno de `gets()` para probar la entrada de una línea en blanco.

```
1: /* Demuestra el uso del valor de retorno de gets(). */
2:
3: #include <stdio.h>
4:
5: /* Declara un arreglo de carácter para la entrada, y un apuntador. */
6:
7: char input[81], *ptr;
8:
9: main()
10: {
11:     /* Despliega instrucciones. */
12:
13:     puts("Enter text a line at a time, then press Enter.");
14:     puts("Enter a blank line when done.");
15:
16:     /* Hace ciclo mientras no se dé una línea en blanco. */
17:
18:     while ( *(ptr = gets(input)) != NULL)
19:         printf("You entered %s\n", input);
20:
21:     puts("Thank you and good-bye");
22: }
```

Análisis

Ahora puede ver la manera en que funciona el programa. Si se da una línea en blanco (esto es, si simplemente se oprime Enter) en respuesta a la línea 18, la cadena (que contiene 0 caracteres) todavía es guardada con un carácter nulo en la primera posición. Esta es la posición a la que apunta el valor de retorno de `gets()`, por lo que si se revisa esa posición y se encuentra un carácter nulo se sabe que fue dada una línea en blanco.

El listado 10.6 ejecuta esta prueba en el enunciado `while` que se encuentra en la línea 18.

Este enunciado es un poco complicado, por lo que observe cuidadosamente sus detalles en orden.

La figura 10.1 etiqueta los componentes del enunciado para una referencia más fácil.

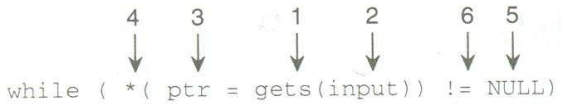


Figura 10.1. *Los componentes de un enunciado* `while` *que prueban la entrada de una línea en blanco.*

1. La función `gets()` acepta entrada del teclado hasta que llega un carácter de nueva línea.
2. La cadena de entrada, menos la nueva línea y con el carácter nulo al final, es guardada en la posición de memoria apuntada por `input`.
3. La dirección de la cadena (el mismo valor que `input`) es regresada al apuntador `ptr`.
4. Un enunciado de asignación es una expresión que evalúa al valor de su variable al lado izquierdo del operador de asignación. Por lo tanto la expresión completa, `ptr = gets(input)`, evalúa al valor de `ptr`. Encerrando esta expresión en paréntesis, y precediéndola con el operador de indirección (*), es obtenido el valor guardado en la dirección apuntada. Esto es, por supuesto, el primer carácter de la cadena de entrada.
5. `NULL` es una constante simbólica definida en el archivo de encabezado STDIO.H. Tiene el valor del carácter nulo (cero).
6. Si el primer carácter de la cadena de entrada no es el carácter nulo (si no se ha tecleado una línea en blanco), el operador de comparación regresa cierto y ejecuta el ciclo `while`. Si el primer carácter es el carácter nulo (si ha tecleado una línea en blanco), el operador de comparación regresa falso y termina el ciclo `while`.

Cuando use a `gets()`, o a cualquier otra función que guarde datos usando un apuntador, asegúrese de que el apuntador apunta a algún espacio asignado. Es fácil cometer un error como el siguiente:

```
char *ptr;
gets(ptr);
```

El apuntador `ptr` ha sido declarado pero no inicializado. Apunta a algún lado pero no sabemos dónde. La función `gets()` no puede saber esto, por lo que simplemente continúa y guarda la cadena de entrada en la dirección contenida en `ptr`. La cadena puede sobreescribir algo importante, como el código de programa o el sistema operativo. El compilador no puede detectar este tipo de errores, por lo que usted, el programador, debe estar vigilando.

Sintaxis

La función *gets()*

```
#include <stdio.h>
char *gets(char *cad);
```

La función `gets()` obtiene una cadena, `cad`, a partir del dispositivo estándar de entrada, que por lo general es el teclado. La cadena consiste en cualesquier caracteres tecleados hasta que se lee un carácter de nueva línea. En este momento se añade un nulo al final de la cadena.

Luego la función `gets()` también regresa un apuntador a la cadena acabada de leer. Si hay algún problema en la obtención de la cadena, `gets()` regresa nulo.

Ejemplo

```
d/* ejemplo de gets() */
#include <stdio.h>

char linea[256];

void main()
{
   printf( "Teclee una cadena:\n");
   gets( linea );
   printf( "\nSe tecleó la siguiente cadena:\n");
   printf( "%s", linea );
}
```

Entrada de cadenas con la función *scanf()*

Se vio en el Día 7, "Entrada/salida básica", que la función de biblioteca `scanf()` acepta entrada de datos numéricos del teclado. Esta función también puede aceptar cadenas. Recuerde que `scanf()` usa un *formato* que le indica la manera de leer la entrada. Para leer una cadena incluya el especificador `%s` en el formato de `scanf()`. De manera similar a `gets()`, a `scanf()` se le pasa un apuntador a la posición de almacenamiento de la cadena.

¿Cómo decide `scanf()` cuándo comienza y termina la cadena? El comienzo es el primer carácter diferente de espacio en blanco que encuentra. El final puede ser especificado de alguna de dos maneras. Si se usa `%s` en el formato, la cadena llega hasta (pero no incluye) el siguiente carácter de espacio en blanco (espacio, tabulador o nueva línea). Si se usa `%ns` (donde `n` es una constante entera que especifica el ancho de campo), `scanf()` da entrada a los siguientes `n` caracteres o hasta que encuentre un carácter de espacio en blanco, lo que suceda primero.

Se pueden leer varias cadenas con `scanf()` incluyendo más de un `%s` en el formato. Para cada `%s` del formato, `scanf()` usa las reglas anteriores para encontrar la cantidad de cadenas pedidas a la entrada. Por ejemplo,

```
scanf("%s%s%s", s1, s2, s3);
```

Si en respuesta a este enunciado se teclea `enero febrero marzo`, `enero` es asignado a la cadena `s1`, `febrero` es asignado a `s2` y `Marzo` es asignado a `s3`.

¿Qué pasa si se usa el especificador de ancho de campo? Si se ejecuta el enunciado

```
scanf("%3s%3s%3s", s1, s2, s3);
```

y en respuesta se teclea

```
noviembre
```

`nov` es asignado a `s1`, `iem` es asignado a `s2` y `bre` es asignado a `s3`.

¿Qué pasa si se teclean menos o más cadenas que las que espera la función `scanf()`? Si se teclean menos cadenas, `scanf()` continúa "esperando" las cadenas faltantes y el programa no continúa sino hasta que sean tecleadas. Por ejemplo, si en respuesta al enunciado

```
scanf("%s%s%s", s1, s2, s3);
```

se teclea

```
enero febrero
```

el programa se sienta a esperar la tercera cadena especificada en el formato de `scanf()`. Si se teclean más cadenas que las pedidas, las cadenas sobrantes permanecen pendientes (esperando en el buffer de teclado) y son leídas por cualquier `scanf()` subsecuente o cualquier otro enunciado de entrada. Por ejemplo,

```
scanf("%s%s", s1, s2);
scanf("%s", s3);
```

Si se teclea

```
enero febrero marzo
```

el resultado es que `enero` es asignado a la cadena `s1`, `febrero` es asignado a `s2` y `marzo` a `s3`.

La función `scanf()` tiene un valor de retorno, un valor entero que es igual a la cantidad de conceptos recibidos satisfactoriamente. Frecuentemente es ignorado el valor de retorno. Cuando se está leyendo solamente texto, es preferible, por lo general, usar la función `gets()` en vez de `scanf()`. La función `scanf()` es mejor usada cuando se está leyendo una combinación de texto y datos numéricos. Esto es ilustrado por el programa del listado 10.7. Recuerde del Día 7, "Entrada/salida básica", que se debe usar el operador de *dirección de* (&) cuando se reciban variables numéricas con `scanf()`.

Captura **Listado 10.7. Entrada de datos numéricos y texto con `scanf()`.**

```
1: /* Muestra el uso de scanf() para entrada de datos numéricos y texto */
2:
3: #include <stdio.h>
4:
5: char lname[81], fname[81];
6: int count, id_num;
7:
8: main()
9: {
10:     /* Da indicaciones al usuario. */
11:
12:     puts("Enter last name, first name, ID number separated");
13:     puts("by spaces, then press Enter.");
14:
15:     /* Recibe los tres conceptos de datos. */
16:
17:     count = scanf("%s%s%d", lname, fname, &id_num);
18:
19:     /* Despliega los datos. */
20:
21:     printf("%d items entered: %s %s %d", count, fname, lname,
            id_num);
22: }
```

Análisis Recuerde que `scanf()` requiere las direcciones de las variables como parámetros. En el listado 10.7 `lname` y `fname` son apuntadores (esto es, direcciones), por lo que no necesitan el operador de *dirección de* (&). Por el contrario, `id_num` es un nombre de variable regular, por lo que requiere el `&` cuando es pasado a `scanf()` en la línea 17.

Algunos programadores sienten que la entrada de datos con `scanf()` propicia los errores. Ellos prefieren recibir todos los datos, numéricos y de cadena, usando `gets()`, y luego hacer que el programa separe los números y los convierta a variables numéricas. Estas técnicas están más allá del alcance de este libro, pero harían un buen ejercicio de programación. Para esta tarea se necesitan las funciones para manipulación de cadenas, tratadas en el Día 17, "Manipulación de cadenas".

Resumen

Este capítulo trató el tipo de dato `char` del C. Un uso para las variables de tipo `char` es el almacenamiento de caracteres individuales. Se vio que, de hecho, los caracteres son guardados como números: el código ASCII tiene asignado un código numérico para cada carácter. Por lo tanto se puede usar al tipo `char` para guardar también valores enteros pequeños. Se encuentran disponibles los tipos `char` con `signo` y `sin signo`.

Una cadena es una secuencia de caracteres terminada por el carácter nulo. Las cadenas pueden usarse para datos de texto. El C guarda las cadenas en arreglos de tipo `char`. Para guardar una cadena de longitud `n`, se necesita un arreglo de tipo `char` con `n+1` elementos.

Se pueden usar funciones de asignación de memoria, como `malloc()`, para hacer más dinámico el programa. Usando `malloc()` se puede asignar la cantidad correcta de memoria para el programa. Sin estas funciones se tendría que adivinar la cantidad de espacio de memoria que necesita el programa. Los estimados son, por lo general, altos, por lo que se asigna más memoria que la necesaria.

Preguntas y respuestas

1. ¿Cuál es la diferencia entre una cadena y un arreglo de caracteres?

   Una cadena está definida como una secuencia de caracteres que termina en el carácter nulo. Un arreglo es una secuencia de caracteres. Por lo tanto, una cadena es un arreglo de caracteres terminado en nulo.

   Si se define un arreglo de tipo `char`, el espacio actual de almacenamiento asignado para el arreglo es el tamaño especificado, y no el tamaño -1. Se está limitado a ese tamaño. No se puede guardar una cadena más grande. A continuación se presenta un ejemplo:

```
char estado[10]="Aguascalientes"; /* ¡Error! La cadena es más larga  */
                                  /* que el arreglo  */
char estado[10]="Ags";            /* Correcto, pero desperdicia espacio porque */
                                  /* la cadena es más corta que el arreglo  */
```

capaz de asignar memoria solamente cuando se necesita. Cuando se termina de usar la memoria se la puede regresar, *liberándola*. Cuando se libera memoria, alguna otra variable o arreglo que se encuentra en otra parte del programa puede usar la memoria. (El Día 7, "Entrada/salida básica", trata la liberación de memoria asignada.)

3. ¿Todas las computadoras aceptan el juego de caracteres ASCII extendido?

   No. La mayoría de las PC aceptan el juego ASCII extendido. Algunas PC antiguas no, pero la cantidad de PC antiguas a las que les falta este soporte está disminuyendo. La mayoría de los programadores usan los caracteres de línea y bloque del juego extendido.

4. ¿Qué pasa si pongo en un arreglo de caracteres una cadena que es más grande que el arreglo?

   Esto causa un error difícil de encontrar. Se puede hacer esto en C, pero es sobreescrita cualquier cosa que se encuentra guardada en la memoria inmediatamente después del arreglo de caracteres. Esta puede ser un área de memoria que no está siendo usada, algunos otros datos o alguna información vital del sistema. Los resultados dependen de lo que se sobreescriba. Muchas veces no sucede nada... por un tiempo. Usted no querrá que suceda eso.

Taller

Cuestionario

1. ¿Qué es el rango de valores numéricos en el juego de caracteres ASCII?
2. Cuando el compilador C encuentra un solo carácter encerrado en comillas simples, ¿cómo es interpretado?
3. ¿Cuál es la definición del C para una cadena?
4. ¿Qué es una cadena literal?
5. Para guardar una cadena de `n` caracteres se necesita un arreglo de caracteres de `n` + 1 elementos. ¿Para qué se necesita el elemento extra?
6. Cuando el compilador C encuentra una cadena literal, ¿cómo es interpretada?
7. Usando la tabla ASCII del apéndice A, "Tabla de caracteres ASCII", encuentre los valores numéricos guardados para cada uno de los siguientes.

   a. a

   b. A

c. 9

d. un espacio

e. ┘└
   ┐┌

f. ♠

8. Usando la Tabla ASCII del apéndice A, "Tabla de caracteres ASCII", traduzca los siguientes valores numéricos a sus caracteres equivalentes.

   a. 73

   b. 32

   c. 99

   d. 97

   e. 110

   f. 0

   g. 2

9. ¿Qué tantos bytes de almacenamiento se encuentran asignados para cada una de las siguientes variables?

   a. `char *cad1 = { "Cadena 1"};`

   b. `char cad2[] = { "Cadena 2"};`

   c. `char cadena3;`

   d. `char cad4[20] = { "Esta es la cadena 4" };`

   e. `char cad5[20];`

10. Usando la siguiente declaración:

```
char *cadena = "¡Una cadena!";
```

¿Cuáles son los valores de lo siguiente?

   a. `cadena[0];`

   b. `*cadena`

   c. `cadena[9]`

   d. `cadena[33]`

   e. `*cadena+8`

   f. `cadena`

Ejercicios

1. Escriba una línea de código que declare una variable tipo `char` llamada `letra` e inicialícela al carácter $.
2. Escriba una línea de código que declare un arreglo de tipo `char`, e inicialícelo a la cadena `"Los apuntadores son divertidos"`. Haga que el arreglo sea sólo lo bastante grande como para que guarde la cadena.
3. Escriba una línea de código que asigne almacenamiento para la cadena `"Los apuntadores son divertidos"`, como en el ejercicio 2, pero sin usar un arreglo.
4. Escriba código que asigne espacio para una cadena de 80 caracteres, y luego reciba la cadena del teclado y guárdela en el espacio asignado.
5. Escriba una función que copie un arreglo de caracteres a otro. (Consejo: hágalo de manera parecida a los programas que hizo en el Día 9, "Apuntadores".)
6. Escriba una función que acepte dos cadenas. Cuente la cantidad de caracteres en cada una y regrese un apuntador a la cadena más larga.
7. Escriba una función que acepte dos cadenas. Use la función `malloc()` para asignar suficiente memoria para guardar las dos cadenas después de que hayan sido concatenadas (puestas juntas). Regrese un apuntador a esta nueva cadena.

   Por ejemplo, si paso "`Hola`" y "`Amigos`", la función regresa un apuntador a "`Hola Amigos`". Haciendo que el valor concatenado sea la tercera cadena, es más fácil. (Tal vez pueda usar las respuestas de los ejercicios 5 y 6.)
8. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
char a_string[10] = "This is a string";
```

9. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
char *quote[100] = { "Smile, Friday is almost here!" };
```

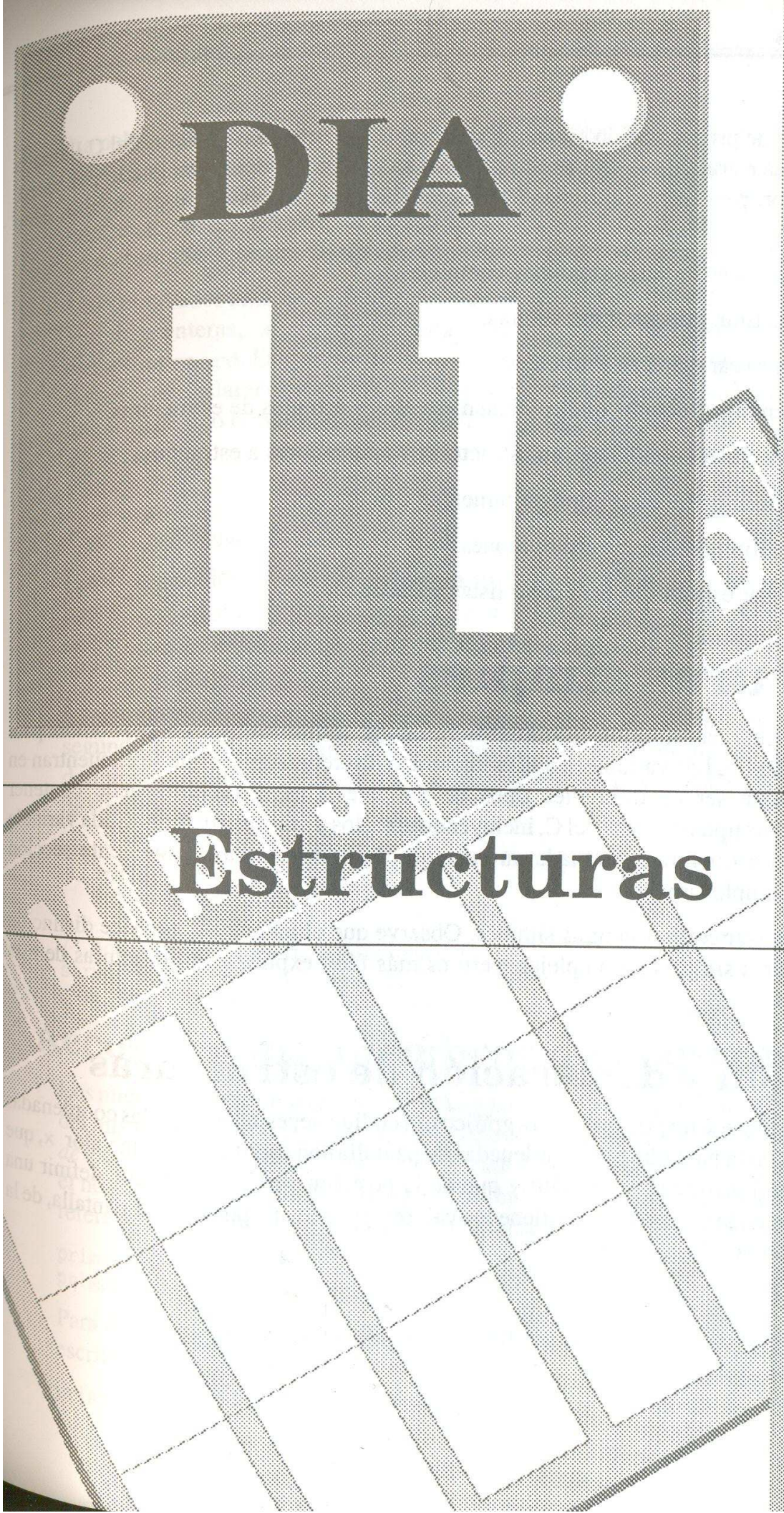
10. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
char *string1;
char *string2 = "Second";

string1 = string2;
```

11. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
char string1[];
char string2[] = "Second";
string1 = string2;
```

12. Usando la tabla de caracteres ASCII escriba un programa que trace un cuadro en la pantalla usando los caracteres de doble línea.



Estructuras

Muchas tareas de programación se simplifican con la construcción de datos del C llamada *estructura*. Una estructura es un método de almacenamiento de datos designado por usted, el programador, para que se ajuste exactamente a las necesidades de programación. Hoy aprenderá

- Qué son las estructuras simples y complejas.
- Cómo definir y declarar estructuras.
- Cómo accesar datos en estructuras.
- Cómo crear estructuras que contienen arreglos y arreglos de estructuras.
- Cómo declarar apuntadores en estructuras y apuntadores a estructuras.
- Cómo pasar estructuras como argumentos a funciones.
- Cómo definir, declarar y usar uniones.
- Cómo usar estructuras para crear listas encadenadas.

Estructuras simples

Una estructura es una colección de una o más variables, agrupadas bajo un solo nombre para facilitar su manejo. Las variables en la estructura, a diferencia de las que se encuentran en arreglos, pueden ser de diferentes tipos de variable. Una estructura puede contener cualquiera de los tipos de datos del C, incluyendo arreglos y otras estructuras. Cada variable dentro de una estructura es llamada un *miembro* de la estructura. La siguiente sección muestra un ejemplo simple.

Se debe comenzar con estructuras simples. Observe que el lenguaje C no hace distinción entre estructuras simples y complejas, pero es más fácil explicar las estructuras de esta manera.

Definición y declaración de estructuras

Si se está escribiendo un programa para gráficos, el código necesita manejar las coordenadas de los puntos en la pantalla. Las coordenadas de pantalla son escritas como un valor `x`, que da la posición horizontal, y un valor y que da la posición vertical. Se puede definir una estructura, llamada `coord`, que contiene los valores `x` y `y` de una posición de pantalla, de la manera siguiente:

```
struct coord{
   int x;
   int y;
};
```

La palabra clave `struct`, que identifica el comienzo de una definición de estructura, debe ser seguida inmediatamente por el nombre de la estructura o *etiqueta* (que sigue las mismas reglas que los otros nombres de variables del C). Dentro de las llaves que se encuentran a continuación del nombre de la estructura va una lista de las variables miembro de la estructura. Se debe dar un tipo de variable y un nombre para cada miembro.

Los enunciados anteriores definen un tipo de estructura, llamada `coord`, que contiene dos variables enteras, `x` y `y`. Sin embargo, ellos no crean, de hecho, ninguna instancia de la estructura `coord`. Esto es, no declaran (reservan espacio para) ninguna estructura. Hay dos maneras de declarar estructuras. Una es poner a continuación de la definición de la estructura una lista de uno o más nombres de variables, como se hace aquí:

```
struct coord{
   int x;
   int y;
} primera, segunda;
```

Estos enunciados definen la estructura tipo `coord` y declaran dos estructuras, llamadas `primera` y `segunda`, de tipo `coord`. `primera` y `segunda` son cada una *instancias* de tipo `coord`; primera contiene dos miembros enteros llamados `x` y `y`, y lo mismo sucede con `segunda`.

El método anterior para declarar estructuras combina la definición con la declaración. El segundo método es declarar las variables de estructura en una posición diferente de la definición en el código fuente. Los siguientes enunciados también declaran dos instancias de tipo `coord`:

```
struct coord{
   int x;
   int y;
};
/* Aquí puede ir código adicional */
struct coord primera, segunda;
```

Acceso de los miembros de la estructura

Los miembros individuales de la estructura pueden usarse de manera similar a otras variables del mismo tipo. Los miembros de estructura son accesados usando el *operador de miembro de estructura* (.), también llamado el *operador de punto,* entre el nombre de la estructura y el nombre del miembro. Por lo tanto, para hacer que la estructura llamada `primera` haga referencia a una posición de pantalla con coordenadas `x = 50, y = 100`, se podría escribir

```
primera.x = 50;
primera.y = 100;
```

Para desplegar las posiciones de pantalla guardadas en la estructura `segunda`, se podría escribir

```
printf( "%d,%d", segunda.x, segunda.y);
```

En este momento tal vez se pregunte cuál es la ventaja de usar estructuras en vez de usar variables individuales. Una gran ventaja es la habilidad de copiar información entre estructuras del mismo tipo con un simple enunciado de asignación. Continuando con el ejemplo anterior, el enunciado

```
primera = segunda;
```

es equivalente a

```
primera.x = segunda.x;
primera.y = segunda.y;
```

Cuando el programa usa estructuras complejas con muchos miembros esta notación puede ahorrar mucho tiempo. Otras ventajas de las estructuras serán evidentes conforme aprenda más capacidades avanzadas. En general, encontrará que las estructuras son útiles en cualquier momento en que información de diferentes tipos de variables necesita ser tratada como un grupo. Por ejemplo, en una base de datos de listas de direcciones cada registro puede ser una estructura, y cada pieza de información (nombre, dirección, ciudad, etc.), un miembro de la estructura.

Sintaxis

La palabra clave *struct*

```
struct etiqueta {
miembros_de_estructura
/* aquí pueden ir enunciados adicionales */
} instancia;
```

La palabra clave `struct` es usada para la declaración de estructuras. Una estructura es una colección de una o más variables (*miembros_de_estructura)* que han sido agrupadas bajo un solo nombre para facilitar su manejo. Las variables no tienen que ser del mismo tipo ni deben ser variables simples. Las estructuras también pueden contener arreglos, apuntadores y otras estructuras.

La palabra clave `struct` identifica el inicio de una definición de estructura. Es seguida por una etiqueta, que es el nombre dado a la estructura. A continuación de la etiqueta se encuentran los miembros de la estructura encerrados entre llaves. Una *instancia,* la declaración actual de una estructura, también puede ser definida. Si se define la estructura sin la instancia es simplemente una plantilla, que puede ser usada posteriormente en un programa para declarar estructuras. A continuación se presenta el formato de la plantilla.

```
struct etiqueta {
miembros_de_estructura
/* aquí pueden ir enunciados adicionales */
};
```

Para usar la plantilla se podría usar el siguiente formato:

```
struct etiqueta instancia;
```

Para usar este formato se debe haber declarado previamente una estructura con la etiqueta dada.

Ejemplo 1

```
/* Declara una plantilla de estructura llamada RFC */

struct RFC {
   char tres_primeros;
   char guion1;
   char dos_segundos;
   char guion2;
   char cuatro_últimos;
}
/* Usa la plantilla de la estructura */
struct RFC rfc_cliente;
```

Ejemplo 2

```
/* Declara una estructura y una instancia */
struct fecha {
   char mes[2];
   char día[2];
   char año[4];
} fecha_actual;
```

Ejemplo 3

```
/* Declara e inicializa una estructura */
struct tiempo {
   int horas;
   int minutos;
   int segundos;
} hora_de_nacimiento = { 8, 45, 0 };
```


Estructuras más complejas

Ahora que ya ha visto las estructuras simples, puede usted pasar a los tipos de estructuras más interesantes y complejos. Estas son estructuras que contienen como miembros a otras estructuras, y estructuras que contienen arreglos como miembros.

Estructuras que contienen estructuras

Como se dijo anteriormente, una estructura de C puede contener cualquiera de los tipos de datos del C. Por ejemplo, una estructura puede contener otras estructuras. Los ejemplos anteriores pueden ser extendidos para ilustrar esto.

Suponga que el programa de gráficos ahora necesita manejar rectángulos. Un *rectángulo* puede ser definido por las coordenadas de sus dos esquinas diagonalmente opuestas. Ya ha visto cómo definir una estructura que pueda guardar las dos coordenadas que se requieren para un solo punto. Necesitará dos de esta estructuras para definir un rectángulo. Se puede

definir una estructura de la manera siguiente (suponiendo, por supuesto, que ya haya definido la estructura tipo coord):

```
struct rectangulo {
   struct coord arribaizquierda;
   struct coord abajoderecha;
};
```

Este enunciado define una estructura de tipo rectángulo que contiene dos estructuras de tipo coord. Estas dos estructuras de tipo coord son llamadas arribaizquierda y abajoderecha.

El enunciado anterior define solamente la estructura tipo rectángulo. Para declarar una estructura se debe incluir luego un enunciado como

```
struct rectángulo micuadro;
```

Se podría haber combinado la definición y la declaración, como se hizo anteriormente para el tipo coord.

```
struct rectángulo {
   struct coord arribaizquierda;
   struct coord abajoderecha;
} micuadro;
```

Para accesar las posiciones actuales de datos (los miembros tipo int) se debe aplicar dos veces al operador de miembro (.). Por lo tanto, la expresión

```
micuadro.arribaizquierda.x
```

hace referencia al miembro x del miembro arribaizquierda de la estructura tipo rectángulo llamada micuadro. Para definir un rectángulo con las coordenadas (0, 10), (100, 200), se podría escribir

```
micuadro.arribaizquierda.x = 0;
micuadro.arribaizquierda.y = 10;
micuadro.abajoderecha.x = 100;
micuadro.abajoderecha.y = 200;
```

Esto puede ser algo confuso. Tal vez lo comprenda mejor si ve la figura 11.1, que muestra la relación entre la estructura tipo rectángulo, las dos estructuras tipo coord que contienen y las dos variables tipo int que contiene cada estructura tipo coord. Las estructuras son nombradas igual que en el ejemplo anterior.

Es tiempo de ver un ejemplo sobre el uso de estructuras que contienen otras estructuras. El programa en el listado 11.1 recibe datos del usuario para las coordenadas de un rectángulo, y luego calcula y despliega el área del rectángulo. Observe las hipótesis del programa, que se dan en comentarios al principio del código (líneas 3 a 8).

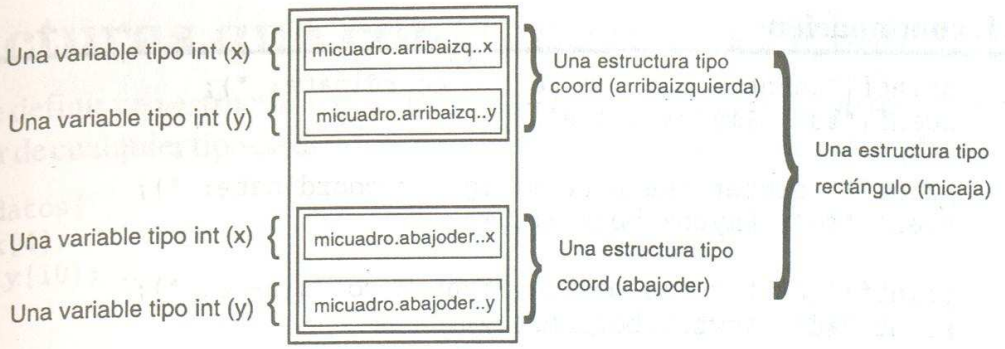


Figura 11.1. *Un diagrama de la relación entre una estructura, estructuras dentro de una estructura y los miembros de estructura.*

Captura

Listado 11.1. Una demostración de estructuras que contienen otras estructuras.

```
1: /* Demuestra estructuras que contienen otras estructuras. */
2:
3: /* Recibe entrada de coordenadas de esquina de un rectángulo y
4:    calcula el área. Supone que la coordenada y de la esquina superior
5:    izquierda es mayor que la coordenada y de la esquina inferior
6:    derecha, y que la coordenada x de la esquina inferior derecha es
7:    mayor que la coordenada x de la esquina superior izquierda, y que
8:    todas las coordenadas son positivas. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16:     int x;
17:     int y;
18: };
19:
20: struct rectangle{
21:     struct coord topleft;
22:     struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27:     /* Recibe las coordenadas */
28:
29:     printf("\nEnter the top left x coordinate: ");
30:     scanf("%d", &mybox.topleft.x);
31:
```

Listado 11.1. continuación

```
32:     printf("\nEnter the top left y coordinate: ");
33:     scanf("%d", &mybox.topleft.y);
34:
35:     printf("\nEnter the bottom right x coordinate: ");
36:     scanf("%d", &mybox.bottomrt.x);
37:
38:     printf("\nEnter the bottom right y coordinate: ");
39:     scanf("%d", &mybox.bottomrt.y);
40:
41:     /* Calcula la longitud y el ancho */
42:
43:     width = mybox.bottomrt.x - mybox.topleft.x;
44:     length = mybox.bottomrt.y - mybox.topleft.y;
45:
46:     /* Calcula y despliega el área */
47:
48:     area = width * length;
49:     printf("The area is %ld units.", area);
50: }
```

Salida

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

Análisis

La estructura `coord` se encuentra definida en las líneas 15 a 18 con sus dos miembros, `x` y `y`. Las líneas 20 a 23 declaran una instancia, llamada `mybox`, de la estructura `rectangle`. Los dos miembros de la estructura rectangle son `topleft` y `bottomrt`, siendo ambos estructuras de tipo `coord`.

Las líneas 29 a 39 rellenan los valores para la estructura `mybox`. Al principio puede parecer que hay solamente dos valores que llenar, debido a que `mybox` tiene solamente dos miembros. Sin embargo, cada uno de los miembros de `mybox` tiene sus propios miembros. `topleft` y `bottomrt` tienen dos miembros cada uno, `x` y `y`, de la estructura `coord`. Esto nos da un total de cuatro miembros que deben ser llenados. Después de que los miembros son llenados con valores, el área es calculada usando los nombres de estructura y de miembro. Cuando se usan los valores `x` y `y` se debe incluir el nombre de instancia de estructura. Debido a que `x` y `y` están en una estructura dentro de otra estructura, se deben usar los nombres de instancia de ambas estructuras, `mybox.bottomrt.x`, `mybox.bottomrt.y`, `mybox.topleft.x` y `mybox.topleft.y`, en los cálculos.

El C no pone límites sobre el anidado de estructuras. Mientras lo permita la memoria, se pueden definir estructuras que contengan estructuras que contengan estructuras que contengan estructuras ...Bien, ¡ya tiene usted la idea! Por supuesto que hay un límite que, al rebasarlo, el anidado llega a ser improductivo. Es muy raro que se usen más de tres niveles de anidado en cualquier programa del C.

Estructuras que contienen arreglos

Se puede definir una estructura que contiene uno o más arreglos como miembros. El arreglo puede ser de cualquier tipo de datos del C (entero, carácter, etc.). Por ejemplo, los enunciados

```
struct datos{
   int x[4];
   char y[10];
};
```

definen una estructura de tipo `datos` que contiene un miembro de arreglo entero de 4 elementos, llamado `x`, y un miembro de arreglo de carácter de 10 elementos, llamado `y`. Luego se puede declarar una estructura, llamada `registro`, de tipo `datos`, de la manera siguiente:

```
struct datos registro;
```

La organización de esta estructura se muestra en la figura 11.2. Observe que en esta figura los elementos del arreglo `x` ocupan el doble de espacio que los elementos del arreglo `y`. Esto se debe (como lo aprendió en el Día 3, "Variables y constantes numéricas") a que un tipo `int` por lo general requiere dos bytes de almacenamiento y un tipo char por lo general requiere solamente de un byte.

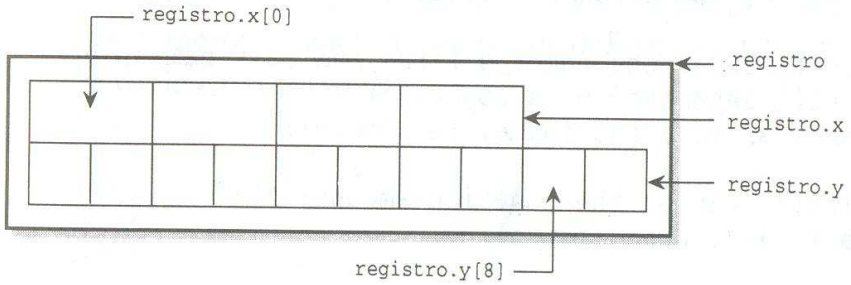


Figura 11.2. *La organización de una estructura que contiene arreglos como miembros.*

El acceso a los miembros individuales de los arreglos que son miembros de estructuras se hace con una combinación del operador de miembro y de los subíndices de arreglo:

```
registro.x[2] = 100;
registro.y[1] = 'x';
```

Probablemente recuerde que los arreglos de caracteres se usan, por lo general, para guardar cadenas. También debe recordar (del Día 9, "Apuntadores") que el nombre de un arreglo sin corchetes es un apuntador al arreglo. Debido a que esto es cierto para los arreglos que son miembros de estructuras, la expresión

```
registro.y
```

es un apuntador al primer elemento del arreglo `y[]` en la estructura `registro`. Por lo tanto, se podría imprimir el contenido de `y[]` en la pantalla con el enunciado

```
puts(registro.y);
```

Ahora veamos otro ejemplo. El programa del listado 11.2 usa una estructura que contiene una variable tipo `float` y dos arreglos tipo `char`.

Captura

Listado 11.2. Una demostración de una estructura que contiene miembros de arreglo.

```
1: /* Demuestra una estructura que tiene miembros de arreglo. */
2:
3: #include <stdio.h>
4:
5: /* Define y declara una estructura para guardar los datos. */
6: /* Contiene una variable float y dos arreglos char. */
7:
8: struct data{
9:     float amount;
10:     char fname[30];
11:     char lname[30];
12: } rec;
13:
14: main()
15: {
16:     /* Recibe los datos del teclado. */
17:
18:     printf("Enter the donor_s first and last names,\n");
19:     printf("separated by a space: ");
20:     scanf("%s %s", rec.fname, rec.lname);
21:
22:     printf("\nEnter the donation amount: ");
23:     scanf("%f", &rec.amount);
24:
25:     /* Despliega la información. */
26:     /* Nota: %.2f especifica un valor de punto flotante */
27:     /* a ser desplegado con dos dígitos a la derecha */
28:     /* del punto decimal. */
29:
30:     /* Depliega los datos en la pantalla. */
31:
32:     printf("\nDonor %s %s gave $%.2f.", rec.fname, rec.lname,
33:             rec.amount);
34: }
```

Salida

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

Análisis

Este programa incluye una estructura que contiene miembros de arreglo llamados `fname[30]` y `lname[30]`. Ambos son arreglos de caracteres que guardan el nombre y el apellido de una persona, respectivamente. La estructura declarada en las líneas 8 a 12 es llamada `data`. Contiene los arreglos de carácter `fname` y `lname` con una variable tipo `float` llamada `amount`. Esta estructura es ideal para guardar el nombre de una persona (en dos partes, nombre y apellido) y un valor, como, por ejemplo, la cantidad que una persona ha donado a una organización de caridad.

Una instancia del arreglo, llamada `rec`, también ha sido declarada en la línea 12. El resto del programa usa `rec` para obtener valores del usuario (líneas 18 a 23) y luego imprimirlos (líneas 32 a 33).

Arreglos de estructuras

Si se tienen estructuras que contienen arreglos, ¿se podrá también tener arreglos de estructuras? ¡Claro que se puede! De hecho, los arreglos de estructuras son una herramienta de programación muy poderosa. Aquí está cómo se hace.

Ya ha visto la manera en que una definición de estructura puede ser acondicionada para que le quepan los datos con los cuales necesita trabajar el programa. Por lo general, un programa necesita trabajar con más de una instancia de los datos. Por ejemplo, en un programa que va a manejar una lista de números de teléfonos se puede definir una estructura para que guarde el nombre y el número de cada persona.

```
struct entrada{
   char nombre[10];
   char apellido[12];
   char telefono[8];
};
```

Sin embargo, una lista de teléfonos debe guardar muchas entradas, por lo que una sola instancia de la estructura no sirve de mucho. Lo que necesita es un arreglo de estructuras de tipo `entrada`. Después de que la estructura ha sido definida, se puede declarar un arreglo de la manera siguiente:

```
struct entrada lista[1000];
```

Este enunciado declara un arreglo llamado `lista` que contiene 1,000 elementos. Cada elemento es una estructura de tipo `entrada`, y es identificada con subíndices de manera similar a cualesquier otros tipos de elementos de arreglo. Cada una de estas estructuras tiene tres elementos, siendo cada uno de ellos un arreglo de tipo `char`. Esta creación compleja se diagrama por completo en la figura 11.3.

Cuando se ha declarado el arreglo de estructuras se pueden manejar los datos de muchas maneras. Por ejemplo, para asignar los datos de un elemento de arreglo a otro elemento de arreglo, se escribe

```
lista[1] = lista[5];
```

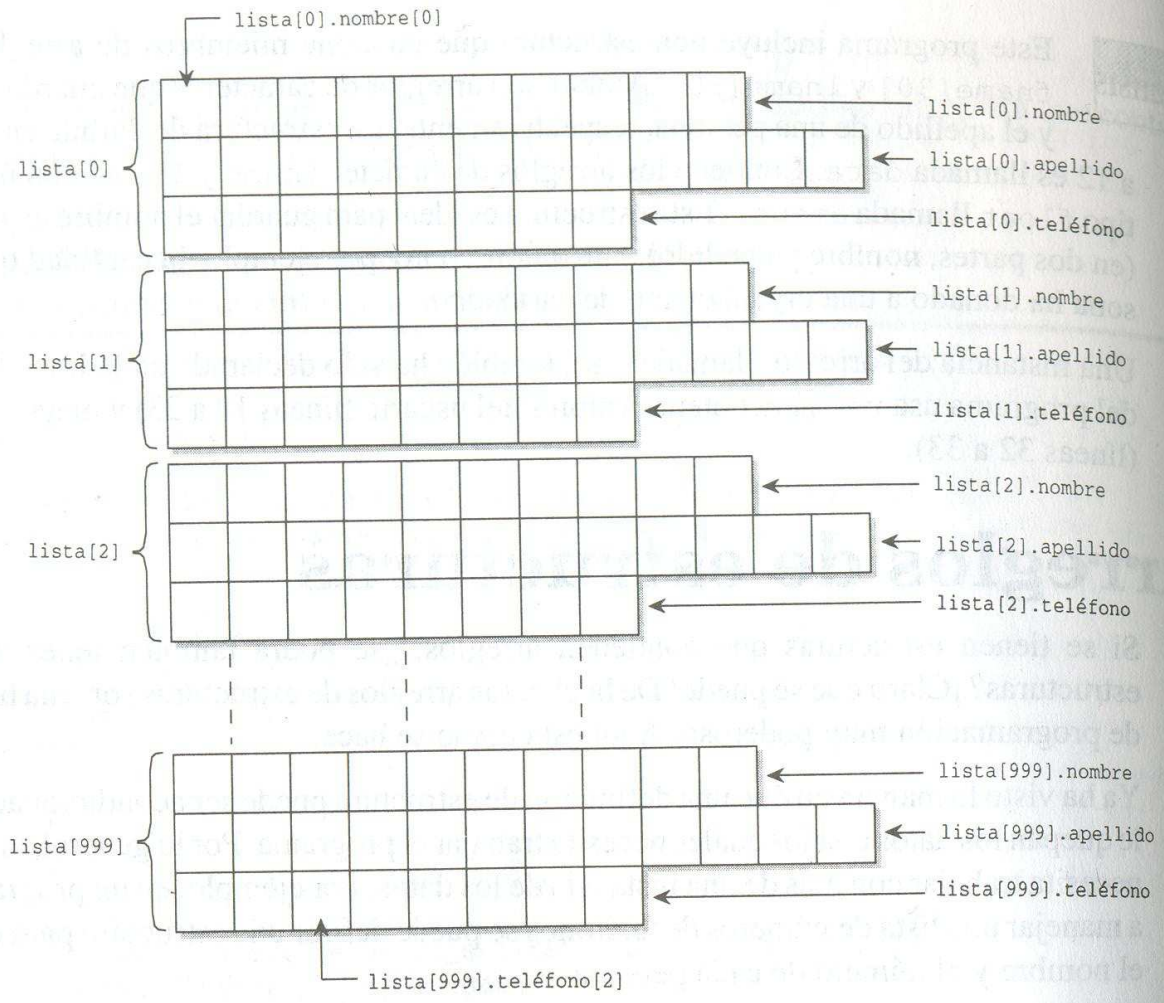


Figura 11.3. *Un diagrama de la relación entre una estructura, estructuras dentro de una estructura y los miembros de estructura.*

Este enunciado asigna a cada miembro de la estructura `lista[1]` los valores contenidos en los miembros correspondientes de `lista[5]`. También se pueden mover datos entre miembros individuales de la estructura. El enunciado

```
strcpy(lista[1].teléfono,      lista[5].teléfono);
```

copia la cadena que se encuentra en `lista[5].teléfono` a `lista[1].teléfono`. (La función de biblioteca `strcpy()` copia una cadena a otra cadena. Aprenderá los detalles de esto en el Día 17, "Manipulación de cadenas".) También puede, si lo desea, mover datos entre elementos individuales de los arreglos miembros de la estructura:

```
lista[5].teléfono[1] = lista[2].teléfono[3];
```

Este enunciado mueve el segundo carácter del número de teléfono de `lista[5]` a la cuarta posición del número de teléfono de `lista[2]`. (No olvide que los subíndices se inician en 0.)

El programa del listado 11.3 demuestra el uso de arreglos de estructuras. Es más, demuestra arreglos de estructuras que contienen arreglos como miembros.

Captura

Listado 11.3. Demostración de arreglos de estructuras.

```
1: /* Demuestra el uso de arreglos de estructuras. */
2:
3: #include <stdio.h>
4:
5: /* Define una estructura para guardar las entradas. */
6:
7: struct entry {
8:     char fname[20];
9:     char lname[20];
10:    char phone[10];
11: };
12:
13: /* Declara un arreglo de estructuras. */
14:
15: struct entry list[4];
16:
17: int i;
18:
19: main()
20: {
21:
22:     /* Hace ciclo para recibir datos de cuatro personas. */
23:
24:     for (i = 0; i < 4; i++)
25:     {
26:         printf("\nEnter first name: ");
27:         scanf("%s", list[i].fname);
28:         printf("Enter last name: ");
29:         scanf("%s", list[i].lname);
30:         printf("Enter phone in 123-4567 format: ");
31:         scanf("%s", list[i].phone);
32:     }
33:
34:     /* Imprime dos líneas en blanco. */
35:
36:     printf("\n\n");
37:
38:     /* Hace ciclo para desplegar los datos. */
39:
40:     for (i = 0; i < 4; i++)
41:     {
42:         printf("Name: %s %s", list[i].fname, list[i].lname);
43:         printf("\t\tPhone: %s\n", list[i].phone);
44:     }
45: }
```

Salida

```
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212

Enter first name: Peter
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434

Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212

Enter first name: John
Enter last name: Smith
Enter phone in 123-4567 format: 555-1234


Name: Bradley Jones              Phone: 555-1212
Name: Peter Aitken               Phone: 555-3434
Name: Melissa Jones              Phone: 555-1212
Name: John Smith                 Phone: 555-1234
```

Análisis

Este listado sigue el mismo formato general que la mayoría de los otros listados. Comienza con los comentarios en la línea 1, y para las funciones de entrada/salida el `#include` para el archivo STDIO.H en la línea 3. Las líneas 7 a 11 definen una plantilla de estructura, llamada `entry`, que contiene tres arreglos de carácter, `fname`, `lname` y `phone`. La línea 15 usa la plantilla para definir un arreglo de cuatro variables de la estructura entry llamadas `list`. La línea 17 define una variable de tipo `int` para ser usada como contador en el programa. `main()` se inicia en la línea 19. La primera función de `main()` es ejecutar un ciclo cuatro veces con un enunciado `for`. Este ciclo es usado para obtener información para el arreglo de estructuras. Esto puede verse en las líneas 24 a 32. Observe que list es usado con subíndice, de manera similar a como fueron usadas con subíndice las variables de arreglo que se vieron en el Día 8, "Arreglos numéricos".

La línea 36 proporciona un corte para la entrada, antes de comenzar la salida. Ella imprime dos líneas en blanco, en una forma que no debe serle extraña. Las líneas 40 a 44 despliegan los datos que el usuario tecleó en el paso anterior. Los valores en el arreglo de estructuras son impresos con el nombre de arreglo con subíndices, seguido por el operador de miembro (.) y el nombre del miembro de la estructura.

Debe familiarizarse con las técnicas usadas en el listado 11.3. Muchas tareas de programación reales se logran mejor usando arreglos de estructuras que contiene arreglos como miembros.

DEBE

NO DEBE

NO DEBE Olvidar el nombre de instancia de la estructura y al operador de miembro (.) cuando use miembros de estructuras.

NO DEBE ¡Confundir la etiqueta de la estructura con sus instancias! La etiqueta es para declarar la plantilla o formato de la estructura. La instancia es una variable declarada usando la etiqueta.

NO DEBE Olvidar la palabra clave struct cuando declare una instancia de una estructura previamente definida.

DEBE Declarar instancias de estructuras con las mismas reglas de alcance que otras variables. (En el Día 12, "Alcance de las variables", se trata este tema a profundidad.)

Inicialización de estructuras

De manera similar a otros tipos de variables del C, las estructuras pueden ser inicializadas cuando son declaradas. El procedimiento es similar al que se usa para inicializar arreglos. La declaración de estructura es seguida por un signo de igual y una lista de valores de inicialización, separados por comas y encerrados entre llaves. Por ejemplo, vea los siguientes enunciados:

```
1: struct venta {
2:     char cliente[20];
3:     char concepto[20];
4:     float importe;
5: } miventa = { "Industrias Acme",
6:               "Escritorio",
7:               1000.00
8:             };
```

Cuando estos enunciados se ejecutan, ejecutan las siguientes acciones:

1. Define un tipo de estructura llamado `venta` (líneas 1 a 5).
2. Declara una instancia de la estructura tipo venta llamada `miventa` (línea 5).
3. Inicializa el miembro `miventa.cliente` de la estructura a la cadena "Industrias Acme" (línea 5).
4. Inicializa el miembro miventa.concepto de la estructura a la cadena "Escritorio" (línea 6).
5. Inicializa el miembro `miventa.importe` de la estructura al valor 1000.00 (línea 7).

Para una estructura que contiene estructuras como miembros, liste los valores de inicialización en orden. Ellos son puestos en las estructuras miembro en el orden en que son listados los miembros en la definición de la estructura. A continuación se presenta un ejemplo que expande ligeramente al anterior:

```
1: struct cliente {
2:    char empresa[20];
3:    char representante[25];
4: };
5:
6: struct venta {
7:    struct cliente comprador;
8:    char concepto[20];
9:    float importe;
10: } miventa = { { "Industrias Acme", "Juan Pérez"},
11:               "Escritorio",
12:               1000.00
13:             };
```

Estos enunciados ejecutan las siguientes inicializaciones:

1. El miembro `miventa.comprador`.empresa de la estructura es inicializado a la cadena "Industrias Acme" (línea 10).
2. El miembro `miventa.comprador.representante` de la estructura es inicializado a la cadena "Juan Pérez" (línea 10).
3. El miembro `miventa.concepto` de la estructura es inicializado a la cadena "Escritorio" (línea 11).
4. El miembro `miventa.importe` de la estructura es inicializado a la cantidad 1000.00 (línea 12).

También se puede inicializar arreglos de estructuras. Los datos de inicialización que se proporcionan son aplicados en orden a las estructuras en el arreglo. Por ejemplo, para declarar un arreglo de estructuras de tipo venta e inicializar los primeros dos elementos del arreglo (esto es, las dos primeras estructuras), se podría escribir:

```
1: struct cliente {
2:    char empresa[20];
3:    char representante[25];
4: };
5:
6: struct venta {
7:    struct cliente comprador;
8:    char concepto[20];
9:    float importe;
10: };
```

```
11:
12:
13: struct venta y1990[100] = {
14:     {{ "Industrias Acme", "Juan Pérez"},
15:              "Escritorio",
16:              1000.00
17:     }
18:     {{ "Compañía Wilson", "Pedro Wilson"},
19:        "Silla modelo 12",
20:         290.00
21:     }
22: };
```

1. El miembro `y1990[0].comprador.empresa` de la estructura es inicializado a la cadena `"Industrias Acme"` (línea 14).
2. El miembro `y1990[0].comprador.representante` de la estructura es inicializado a la cadena `"Juan Pérez"` (línea 14).
3. El miembro `y1990[0].concepto` de la estructura es inicializado a la cadena `"Escritorio"` (línea 15).
4. El miembro `y1990[0].importe` de la estructura es inicializado a la cantidad `1000.00` (línea 16).
5. El miembro `y1990[1].comprador.empresa` de la estructura es inicializado a la cadena `"Compañía Wilson"` (línea 18).
6. El miembro `y1990[1].comprador.representante` de la estructura es inicializado a la cadena `"Pedro Wilson"` (línea 18).
7. El miembro `y1990[1].concepto` de la estructura es inicializado a la cadena `"Silla modelo 12"` (línea 19).
8. El miembro `y1990[1].importe` de la estructura es inicializado a la cantidad `290.00` (línea 20).

Estructuras y apuntadores

Debido a que los apuntadores son una parte tan importante del C, no debe sorprenderle que puedan usarse con las estructuras. Se pueden usar apuntadores como miembros de la estructura, y también se pueden declarar apuntadores a las estructuras. Estos se tratan, a su vez, en los siguientes párrafos.

Apuntadores como miembros de estructuras

Se tiene una flexibilidad completa para usar apuntadores como miembros de estructura. Los miembros de apuntador son declarados en la misma forma que los apuntadores que no son miembros de estructura, esto es, usando el operador de indirección (*). A continuación se presenta un ejemplo:

```
struct datos {
   int *valor;
   int *tasa;
} primera;
```

Estos enunciados definen y declaran una estructura cuyos dos miembros son apuntadores a tipo `int`. Como sucede con todos los apuntadores, no es suficiente declararlos. Se debe, asignándoles la dirección de una variable, inicializarlos para que apunten a algo. Si `costo` e `interés` han sido declarados para ser variables de tipo `int`, se podría escribir

```
primera.valor = &costo;
primera.tasa = &interes;
```

Ahora que los apuntadores han sido inicializados, se puede usar el operador de indirección (*) como se explicó en el Día 9, "Apuntadores". La expresión `*primera.valor` evalúa al valor de `costo`, y la expresión `*primera.tasa` evalúa al valor de `interés`.

Tal vez el tipo más frecuente de apuntador usado como miembro de estructura es un apuntador a tipo `char`. Recuerde del Día 10, "Caracteres y cadenas", que una cadena es una secuencia de caracteres delineada por un apuntador que apunta al primer carácter de la cadena y un carácter nulo que indica el fin de la cadena. Por si lo ha olvidado, se puede declarar un apuntador a tipo `char` e inicializarlo para que apunte a una cadena de la manera siguiente:

```
char *p_mensaje;
p_mensaje = "Aprenda C por usted mismo en 21 días";
```

Se puede hacer lo mismo con apuntadores a tipo `char` que son miembros de estructuras:

```
struct msg {
   char *p1;
   char *p2;
} misptr;

misptr.p1 = "Aprenda C por usted mismo en 21 días";
misptr.p2 = "Por Prentice Hall";
```

La figura 11.4 ilustra el resultado de la ejecución de los enunciados anteriores. Cada apuntador miembro de la estructura apunta al primer byte de una cadena, guardada en cualquier lugar de memoria. Compare esto con la figura 11.3, que muestra la manera en que los datos son guardados en una estructura que contiene arreglos de tipo `char`.

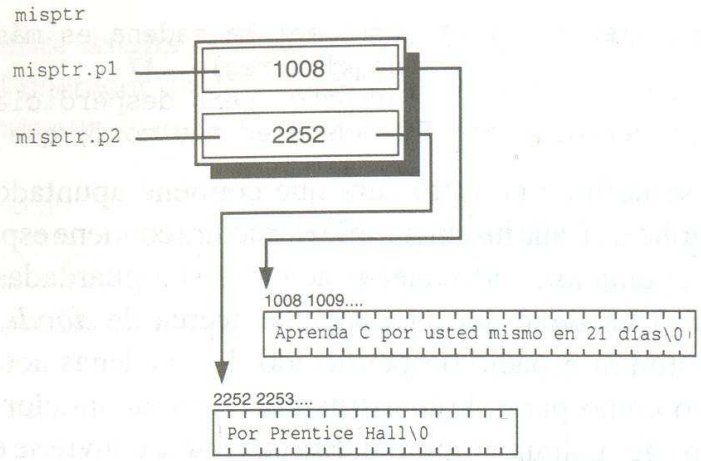


Figura 11.4. *Una estructura que contiene apuntadores a tipo* `char`.

Se puede usar apuntadores miembros de estructura en cualquier lugar en que pueda ser utilizado un apuntador. Por ejemplo, para imprimir las cadenas apuntadas se podría escribir

```
printf( "%s %s, misptr.p1, misptr.p2);
```

¿Cuál es la diferencia entre usar un arreglo de tipo `char` como miembro de estructura y usar un apuntador a tipo `char`? Ambos son métodos para "guardar" una cadena en una estructura, como se muestra aquí en la estructura msg que emplea ambos métodos:

```
struct msg {
   char p1[10];
   char *p2;
} misptr;
```

Recuerde que un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Por lo tanto, puede usar estos dos miembros de estructura en forma similar:

```
strcpy(misptr.p1, "Aprenda C por usted mismo en 21 días");
strcpy(misptr.p2, "Por Prentice Hall");
/* aquí va código adicional */
puts(misptr.p1);
puts(misptr.p2);
```

¿Cuál es la diferencia entre estos métodos? Es ésta: si se define una estructura que contiene un arreglo de tipo `char`, cada instancia de ese tipo de estructura contiene espacio de almacenamiento para el arreglo del tamaño especificado. Es más, se está limitado al tamaño especificado y no se puede guardar una cadena más grande en la estructura. Este es un ejemplo:

```
struct msg {
   char p1[10];
   char p2[10];
} misptr;
...
```

```
strcpy(p1, "Aguascalientes"); /* ¡Erróneo! La cadena es más larga */
                              /* que el arreglo. */
strcpy(p2, "Ags");            /* Correcto, pero desperdicia espacio */
          /* debido a que la cadena es más corta que el arreglo.*/
```

Si, por otro lado, se define una estructura que contiene apuntadores a tipo `char`, estas restricciones no se aplican. Cada instancia de la estructura contiene espacio de almacenamiento solamente para el apuntador. Las cadenas actuales son guardadas en cualquier lugar de memoria (pero usted no tiene que preocuparse acerca de *dónde* en memoria). No hay restricción de longitud ni espacio desperdiciado. Las cadenas actuales son guardadas en cualquier lado y no como parte de la estructura. Cada apuntador en la estructura puede apuntar a una cadena de cualquier longitud. Esa cadena se convierte en parte de la estructura, incluso aunque no esté guardada en la estructura.

Apuntadores a estructuras

Un programa en C puede declarar y usar apuntadores a estructuras, de manera similar a los apuntadores que apuntan a cualquier otro tipo de almacenamiento de dato. Los apuntadores a estructuras se usan frecuentemente cuando se pasa una estructura como argumento a una función. Los apuntadores a estructuras también se usan en un método de almacenamiento de datos muy poderoso, conocido como *listas encadenadas*. Ambos temas se tratan posteriormente en este capítulo.

Por ahora, veamos la manera en que el programa puede crear y usar apuntadores a estructuras. Primero defina una estructura.

```
struct parte {
   int número;
   char nombre[10];
};
```

Ahora declare un apuntador al tipo `parte`.

```
struct parte *p_parte;
```

Recuerde, el operador de indirección (*) en la declaración dice que `p_parte` es un apuntador al tipo `parte` y no una instancia de tipo `parte`.

¿Puede ser inicializado ahora el apuntador? No, debido a que ha sido definida la estructura `parte` pero no ha sido declarada ninguna instancia de ella. Recuerde que es la declaración, y no la definición, la que reserva espacio de almacenamiento en memoria para los objetos de datos. Debido a que un apuntador necesita una dirección de memoria a la cual apuntar, se debe declarar una instancia de tipo `parte` antes de que cualquier cosa pueda apuntar a ella. Por lo tanto, aquí está la declaración:

```
struct parte rueda;
```

Ahora se puede ejecutar la inicialización del apuntador.

```
p_parte = &rueda;
```

El enunciado anterior asigna la dirección de rueda a `p_parte`. (Recuerde del Día 9, "Apuntadores", el operador de dirección de `(&)`.) La relación entre una estructura y un apuntador a la estructura se muestra en la figura 11.5.

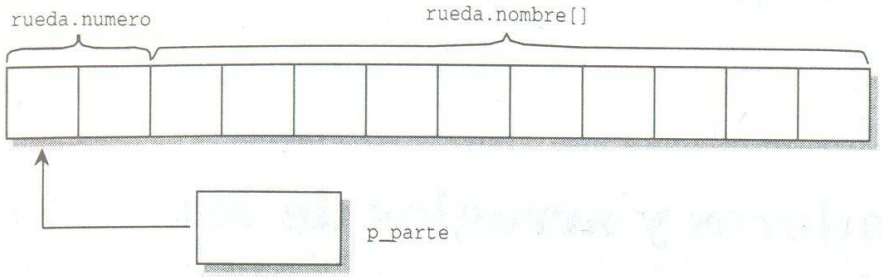


Figura 11.5. *Un apuntador a una estructura apunta al primer byte de la estructura.*

Ahora que ya tiene un apuntador a la estructura rueda, ¿cómo lo usa? Un método usa el operador de indirección (*). Recuerde del Día 9, "Apuntadores", que si

```
ptr
```

es un apuntador a un objeto de dato, la expresión

```
*ptr
```

se refiere al objeto apuntado. Aplicando esto al ejemplo actual, se sabe que `p_parte` es un apuntador a la estructura rueda y, por lo tanto, `*p_parte` se refiere a rueda. Luego se aplica el operador de miembro de estructura (.) para accesar miembros individuales de rueda. Para asignar el valor de 100 a `rueda.numero` se podría escribir

```
(*p_parte).número = 100;
```

`*p_parte` debe ser encerrado entre paréntesis, debido a que el operador (.) tiene una precedencia mayor que el operador (*).

Un segundo método para accesar miembros de estructura usando un apuntador a la estructura es usar el *operador de membresía indirecta,* que consiste en los símbolos `->`, (un guión seguido por el símbolo de mayor que). (Observe que usándolos de esta manera, el C los trata como un solo operador y no como dos.) El símbolo es puesto entre el nombre del apuntador y el nombre del miembro. Para accesar el miembro número de rueda con el apuntador `p_parte`, se podría escribir

```
p_parte ->número
```

Viendo otro ejemplo, si `str` es una estructura, `p_str` es un apuntador a `str` y `miem` es un miembro de `str`, se puede accesar a `str.miem` escribiendo

```
p_str->miem
```

Por lo tanto, hay tres maneras de accesar un miembro de estructura: una, usando el nombre de la estructura, otra, usando un apuntador a la estructura con el operador de indirección (*) y la tercera, usando un apuntador a la estructura con el operador de membresía indirecta (->). Si `p_str` es un apuntador a la estructura `str`, las siguientes expresiones son equivalentes:

```
str.miem
(*p_str).miem
p_str->miem
```

Apuntadores y arreglos de estructuras

Ya ha visto que los arreglos de estructuras pueden ser una herramienta de programación muy poderosa y, de la misma forma, pueden ser los apuntadores a estructuras. Se pueden combinar los dos usando apuntadores para accesar estructuras que son elmentos de arreglo.

Para ilustrarlo, aquí se encuentra una definición de estructura de un ejemplo anterior:

```
struct parte {
   int numero;
   char nombre[10];
};
```

Después de que ha sido definida la estructura parte, se puede declarar un arreglo de tipo `parte`.

```
struct parte datos[100];
```

Luego se puede declarar un apuntador al tipo `parte`, e inicializarlo para que apunte a la primera estructura en el arreglo `datos`:

```
struct parte *p_parte;
p_parte = &datos[0];
```

Recuerde que el nombre de un arreglo sin corchetes es un apuntador al primer elemento del arreglo, por lo que la segunda línea también podría haber sido escrita

```
p_parte = datos;
```

Ahora tenemos un arreglo de estructuras de tipo `parte`, y un apuntador al primer elemento de arreglo (esto es, la primera estructura en el arreglo). Se podría, por ejemplo, imprimir el contenido del primer elemento con el enunciado

```
printf("%d %s", p_parte->número, p_parte->nombre);
```

¿Qué pasa si se quiere imprimir todos los elementos del arreglo? Probablemente se usaría un ciclo `for`, imprimiendo un elemento de arreglo en cada iteración del ciclo. Para accesar los miembros usando notación de apuntadores, se debe cambiar el apuntador `p_parte` para que en cada iteración del ciclo apunte al siguiente elemento del arreglo (esto es, la siguiente estructura en el arreglo). ¿Cómo se hace esto?

La aritmética de apuntadores del C viene en su auxilio. El operador unario de incremento (++) tiene un significado especial cuando es aplicado a un apuntador: significa "incremente el apuntador por el tamaño del objeto al que apunta". Dicho de otra forma, si se tiene un apuntador `ptr` que apunta a un objeto de datos de tipo `obj`, el enunciado

```
ptr++;
```

tiene el mismo efecto que

```
ptr += sizeof(obj);
```

Este aspecto de la aritmética de apuntadores es particularmente relevante para los arreglos, de la siguiente manera: los elementos de arreglo se encuentran guardados secuencialmente en memoria. Si un apuntador apunta al elemento de arreglo n, al incrementar el apuntador con el operador (++) se logra que apunte al elemento `n + 1`. Esto se ilustra en la figura 11.6, que muestra un arreglo llamado `x[]` que consiste en elementos de cuatro bytes (por ejemplo, una estructura que contiene dos miembros tipo `int`, cada uno de dos bytes de largo). El apuntador `ptr` fue inicializado para que apuntara a `x[0]`. Cada vez que es incrementado, `ptr` apunta al siguiente elemento de arreglo.

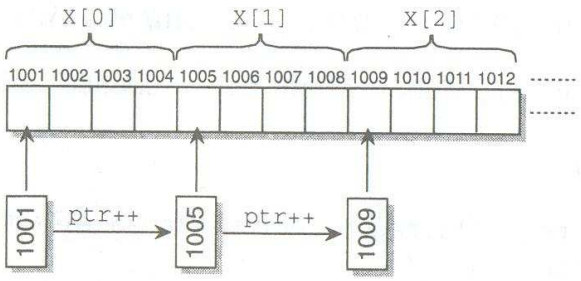


Figura 11.6. *Con cada incremento, un apuntador "avanza" al siguiente elemento de arreglo.*

Lo que esto significa es que el programa puede avanzar por un arreglo de estructuras (o un arreglo de cualquier otro tipo de datos) incrementando un apuntador. Este tipo de notación es, por lo general, más fácil de usar y más concisa que usar subíndices de arreglo para ejecutar la misma tarea.

El programa del listado 11.4 le muestra la manera de hacer esto. A continuación se presenta la salida del programa.

Captura

Listado 11.4. Acceso de elementos de arreglos sucesivos incrementando un apuntador.

```
1: /* Demuestra el avance paso a paso por un arreglo de estructuras */
2: /* usando notación de apuntadores. */
3:
4: #include <stdio.h>
```

Listado 11.4. continuación

```
5:
6: #define MAX 4
7:
8: /* Define una estructura, y luego declara e inicializa */
9: /* un arreglo de cuatro estructuras */
10:
11: struct part {
12:     int number;
13:     char name[10];
14: } data[MAX] = {1, "Smith",
15:                2, "Jones",
16:                3, "Adams",
17:                4, "Wilson"
18:                };
19:
20: /* Declara un apuntador al tipo part y una variable de contador. */
21:
22: struct part *p_part;
23: int count;
24:
25: main()
26: {
27:     /* Inicializa el apuntador al primer elemento del arreglo. */
28:
29:     p_part = data;
30:
31:     /* Hace ciclo por el arreglo incrementando el apuntador */
32:     /* en cada iteración. */
33:
34:     for (count = 0; count < MAX; count++)
35:     {
36:         printf("\nAt address %d: %d %s", p_part, p_part->number,
37:                 p_part->name);
38:         p_part++;
39:     }
40:
41: }
```

Salida

La salida del programa se muestra aquí:

```
At address 96: 1 Smith
At address 108: 2 Jones
At address 120: 3 Adams
At address 132: 4 Wilson
```

Análisis

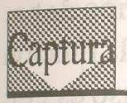
Primero, este programa declara e inicializa un arreglo de estructuras en las líneas 11 a 18, llamado `data`. Un apuntador, llamado `p_part`, es definido luego para que apunte a la estructura `data`. La primera tarea de la función `main()` es hacer que el apuntador `p_part` apunte a la estructura `part` que fue declarada. Luego se imprimen todos los

elementos, usando un ciclo `for` que incrementa el apuntador al arreglo en cada iteración. El programa despliega la dirección de cada elemento.

Vea detalladamente las direcciones desplegadas. Los valores exactos pueden diferir en su sistema, pero tienen incrementos del mismo tamaño, el tamaño justo de la estructura part (la mayoría de los sistemas tendrán un incremento de 12). Esto ilustra claramente que al incrementar un apuntador se le incrementa por una cantidad igual al tamaño del objeto de datos al que apunta.

Paso de estructuras como argumentos a funciones

De manera similar a otros tipos de datos, una estructura puede ser pasada como argumento a una función. El programa del listado 11.5 muestra la manera de hacerlo. Este programa es una modificación del programa del listado 11.2, usando una función para desplegar datos en la pantalla (a diferencia del listado 11.2 que usa enunciados que son parte de `main()`).



Listado 11.5. Paso de una estructura como argumento a una función.

```
1: /* Demuestra el paso de una estructura a una función. */
2:
3: #include <stdio.h>
4:
5: /* Declara y define una estructura para guardar los datos. */
6:
7: struct data{
8:     float amount;
9:     char fname[30];
10:    char lname[30];
11: } rec;
12:
13: /* El prototipo de función. La función no tiene valor de retorno, */
14: /* y toma una estructura de tipo data como su único argumento. */
15:
16: void print_rec(struct data x);
17:
18: main()
19: {
20:     /* Recibe los datos del teclado */
21:
22:     printf("Enter the donor's first and last names,\n");
23:     printf("separated by a space: ");
24:     scanf("%s %s", rec.fname, rec.lname);
25:
26:     printf("\nEnter the donation amount: ");
27:     scanf("%f", &rec.amount);
28:
29:     /* Llama la función de desplegado. */
```

Listado 11.5. continuación

```
30:
31:     print_rec( rec );
32: }
33:
34: void print_rec(struct data x)
35: {
36:     printf("\nDonor %s %s gave $%.2f.", x.fname, x.lname,
37:             x.amount);
38: }
```

Salida

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

Análisis

Viendo la línea 16, se ve el prototipo de función para la función que va a recibir la estructura. Como se haría con cualquier otro tipo de dato que va a ser pasado, se necesita incluir los argumentos adecuados. En este caso es una estructura de tipo `data`. Esto es repetido en el encabezado para la función, en la línea 34. Cuando se llama a la función se necesita solamente pasar el nombre de instancia de la estructura, que en este caso es `rec` (línea 31). Esto es todo. Pasar una estructura a una función no es muy diferente que pasarle una simple variable.

También se puede pasar una estructura a una función pasando la dirección de la estructura (esto es, un apuntador a la estructura). En versiones antiguas del C, ésta era, de hecho, la única manera de pasar una estructura como argumento. Ya no es necesario, pero tal vez vea programas antiguos que usan todavía este método. Si se pasa un apuntador a una estructura como argumento, recuerde que debe usar el operador de membresía indirecta (`->`) para accesar miembros de estructura en la función.

DEBE

NO DEBE ¡Confundir los arreglos con las estructuras!

DEBE Aprovechar la declaración de apuntadores a estructuras, especialmente cuando use arreglos de estructuras.

NO DEBE Olvidar que cuando se incrementa un apuntador, se mueve una distancia equivalente al tamaño del dato al que apunta. En el caso de un apuntador a una estructura, este es el tamaño de la estructura.

DEBE Usar el operador de membresía indirecta (->) cuando trabaje con un apuntador a una estructura.

NO DEBE

Uniones

Las *uniones* son similares a las estructuras. Una unión es declarada y usada en la misma forma que una estructura. Una unión se diferencia de una estructura en que, en un momento dado, solamente puede ser usado uno de sus miembros. La razón de esto es simple. Todos los miembros de la unión ocupan la misma área de memoria. Están puestos uno encima de otro.

Definición, declaración e inicialización de uniones

Las uniones son definidas y declaradas de manera parecida a las estructuras. La única diferencia en la declaración es que se usa la palabra clave `unión` en vez de `struct`. Para definir una unión simple de una variable `char` y una variable entera, se escribiría lo siguiente:

```
unión compartida {
char c;
int i;
};
```

Esta unión, `compartida`, puede ser usada para crear instancias de una unión que puede guardar un valor de carácter `c` o un valor entero `i`. Esta es una condición `O`. A diferencia de una estructura, que podría contener ambos valores, la unión sólo puede contener un valor a la vez.

Una unión puede ser inicializada en su declaración. Debido a que solamente un miembro puede ser usado a la vez, solamente uno puede ser inicializado. Para evitar confusiones, sólo puede ser inicializado el primer miembro de la unión. Lo siguiente muestra una instancia de la unión `compartida`, siendo declarada e inicializada:

```
unión compartida variable_genérica = {'@');
```

Observe que la unión `variable_genérica` fue inicializada de manera similar a como sería inicializado el primer miembro de una estructura.

Acceso de miembros de la unión

Los miembros individuales de la unión pueden usarse de la misma forma en que se usan los miembros de la estructura, con el operador de miembro (.). Hay una diferencia importante en el acceso a miembros de unión. Solamente un miembro debe ser accesado a la vez. Debido a que una unión guarda a sus miembros uno encima de otro, es importante accesar solamente un miembro a la vez. Esto se muestra mejor con un ejemplo.

Captura

Listado 11.6. Un ejemplo del uso erróneo de uniones.

```
1:  /* Ejemplo del uso de más de un miembro de unión a la vez */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      union shared_tag {
7:          char   c;
8:          int    i;
9:          long   l;
10:         float  f;
11:         double d;
12:     } shared;
13:
14:     shared.c = '$';
15:
16:     printf("\nchar c   = %c",  shared.c);
17:     printf("\nint i    = %d",  shared.i);
18:     printf("\nlong l   = %ld", shared.l);
19:     printf("\nfloat f  = %f",  shared.f);
20:     printf("\ndouble d = %f",  shared.d);
21:
22:     shared.d = 123456789.8765;
23:
24:     printf("\n\nchar c   = %c",  shared.c);
25:     printf("\nint i    = %d",  shared.i);
26:     printf("\nlong l   = %ld", shared.l);
27:     printf("\nfloat f  = %f",  shared.f);
28:     printf("\ndouble d = %f",  shared.d);
29: }
```

Salida

```
char c   = $
int i    = 4900
long l   = 437785380
float f  = 0.000000
double d = 0.000000

char c   = 7
int i    = -30409
long l   = 1468107063
float f  = 284852666499072.000000
double d = 123456789.876500
```

Análisis

En este listado se puede ver que una unión, llamada `shared`, fue definida y declarada en las líneas 6 a 12. `shared` contiene 5 miembros, cada uno de diferente tipo. Las líneas 14 y 22 inicializan miembros individuales de `shared`. Las líneas 16 a 20 y 24 a 28 presentan luego los valores de cada miembro usando enunciados `printf()`.

Observe que a excepción de `char c = $` y double `d = 123456789.876500`, la salida no puede ser la misma en su computadora. Debido a que la variable de carácter, `c`, fue inicializada en la línea 14, es el único valor que debe ser usado hasta que un miembro diferente sea inicializado. Los resultados de la impresión de las otras variables miembro de la unión (`i, l, f y d`) pueden ser impredecibles (líneas 16 a 20). La línea 22 pone un valor en la variable `double, d`. Observe que la impresión de las variables nuevamente es impredecible para todas, a excepción de `d`. El valor dado para `c` en la línea 14 se ha perdido, debido a que ha sido sobreescrito cuando se dio el valor de `d` en la línea 22. Esta es una evidencia de que todos los miembros ocupan el mismo espacio.

Sintaxis

La palabra clave *unión*

```
unión etiqueta {
miembros_de_la_unión
/* aquí pueden ir enunciados adicionales */
}instancia;
```

La palabra clave `unión` es usada para declarar uniones. Una unión es una colección de una o más variables (`miembros_de_la_unión`) que han sido agrupados bajo un solo nombre. Además, cada uno de estos miembros de la unión ocupa la misma área de memoria.

La palabra clave `unión` identifica el inicio de una definición de unión. Es seguida por una etiqueta, que es el nombre dado a la unión. A continuación de la etiqueta se encuentran los miembros de la unión encerrados en llaves. Una `instancia`, la declaración actual de la unión, también puede ser definida. Si se define la estructura sin la instancia es simplemente una plantilla, que puede ser usada posteriormente en un programa para declarar estructuras. A continuación se presenta un formato de plantilla:

```
unión etiqueta {
miembros_de_la_unión
/* aquí pueden ir enunciados adicionales */
};
```

Para usar la plantilla se podría usar el siguiente formato:

```
unión etiqueta instancia;
```

Para usar este formato se debe haber definido previamente una unión con la etiqueta dada.

Ejemplo 1

```
/* Declara una plantilla de unión llamada ??? */
unión etiqueta {
   int num;
   char carácter;
}
/* Usa la plantilla de unión */
union etiqueta variables_mezcladas;
```

Ejemplo 2

```
/* Declara al mismo tiempo una unión y una instancia */
unión etiqueta_tipo_genérico {
   char c;
   int i;
   float f;
   double d;
} genérico;
```

Ejemplo 3

```
/* Inicializa una unión. */
unión etiq_fecha {
   char fecha_completa[9];
   struct etiq_fecha_en_partes {
      char día[2];
      char valor_corte1;
      char mes[2];
      char valor_corte2;
      char ano[2];
   } fecha_en_partes;
} fecha = {"01/01/97"};
```

El listado 11.7 muestra un uso más práctico de una unión. Este listado muestra un uso simplista de una unión. Aunque es simplista, es uno de los más comunes de las uniones.



Listado 11.7. Un uso práctico de una unión.

```
1:    /* Ejemplo de un uso típico de una unión */
2:
3:    #include <stdio.h>
4:
5:    #define CHARACTER    'C'
6:    #define INTEGER      'I'
7:    #define FLOAT        'F'
8:
9:    struct generic_tag{
10:       char type;
11:       union shared_tag {
12:            char   c;
13:            int    i;
14:            float  f;
15:       } shared;
16:   };
17:
18:   void print_function( struct generic_tag generic );
19:
20:   main()
21:   {
```

```
22:         struct generic_tag var;
23:
24:         var.type = CHARACTER;
25:         var.shared.c = '$';
26:         print_function( var );
27:
28:         var.type = FLOAT;
29:         var.shared.f = 12345.67890;
30:         print_function( var );
31:
32:         var.type = _x_;
33:         var.shared.i = 111;
34:         print_function( var );
35:     }
36:
37:     void print_function( struct generic_tag generic )
38:     {
39:         printf("\n\nThe generic value is...");
40:         switch( generic.type )
41:         {
42:             case CHARACTER: printf("%c",  generic.shared.c);
43:                             break;
44:             case INTEGER:   printf("%d",  generic.shared.i);
45:                             break;
46:             case FLOAT:     printf("%f",  generic.shared.f);
47:                             break;
48:             default:        printf("an unknown type: %c",
49:                                    generic.type);
50:                             break;
51:         }
52:     }
```

Salida

```
The generic value is...$

The generic value is...12345.678711

The generic value is...an unknown type: x
```

Análisis

Este programa es una versión muy simplista de lo que podría ser hecho con una unión. Este programa proporciona una manera de guardar varios tipos de datos en un solo espacio de almacenamiento. La estructura `generic_tag` le permite guardar un carácter o un entero o un número de punto flotante dentro de la misma área. Esta área es una unión llamada `shared`, que funciona de manera similar a los ejemplos del listado 11.6. Observe que la estructura `generic_tag` también añade un campo adicional llamado `type`. Este campo es usado para guardar información sobre el tipo de variable contenida en `shared`. `type` ayuda a prevenir que `shared` sea usado en forma equivocada y, por lo tanto, ayuda a evitar datos erróneos, como se presentaron en el listado 11.6.

Una vista formal del programa muestra que las línea 5, 6 y 7 definen las constantes `CHARACTER`, `INTEGER` y `FLOAT`. Estas se usan posteriormente en el programa para hacer más legible el listado. Las líneas 9 a 16 definen una estructura, `generic_tag`, que será usada posteriormente. La línea 18 presenta un prototipo para la función `print_function()`. La estructura var es declarada en la línea 22, e inicializada por primera vez en las líneas 24 y 25 para que guarde un valor de carácter. Una llamada a la función `print_function()`, en la línea 26, permite que se impriman los valores. Las líneas 28 a 30 y 32 a 40 repiten este proceso con otros valores.

La función `print_function()` es la parte medular de este listado. Aunque esta función es usada para imprimir el valor de una variable de `generic_tag`, se podría haber usado una función similar para inicializarla. `print_function()` evaluará la variable `type` para imprimir un enunciado con el tipo de variable adecuado. Esto impide el que se obtengan datos erróneos, como los del listado 11.6.

DEBE	NO DEBE
NO DEBE Tratar de inicializar más que el primer miembro de la unión.	
DEBE Recordar qué miembro de la unión se está usando. Si se llena un miembro de un tipo y se trata de usar de otro tipo diferente, se pueden obtener resultados impredecibles.	
NO DEBE Olvidar que el tamaño de una unión es igual al de su miembro más grande.	
DEBE Observar que las uniones son un tema avanzado del C.	

Listas encadenadas

El último tema de este capítulo es una breve introducción a las listas encadenadas. El término *listas encadenadas* se refiere a una clase general de métodos de almacenamiento de datos, en la cual cada concepto de información está encadenado a uno o más conceptos diferentes, usando apuntadores.

Hay varias clases de listas encadenadas, incluyendo listas con encadenamiento simple, listas con encadenamiento doble y árboles binarios. Cada tipo es adecuado para determinadas tareas de almacenamiento de datos. Lo que tienen en común es que los encadenamientos entre conceptos de datos están definidos por información que se encuentra en los mismos conceptos. En esto se distinguen de los arreglos, donde los encadenamientos entre los conceptos de datos son resultado de la estructura del arreglo. Este capítulo explica el tipo más simple de lista encadenada, la lista con encadenamiento sencillo (a la cual se hace referencia simplemente como lista encadenada).

La organización de una lista encadenada

Para ilustrar la manera en que son construidas las listas encadenadas comencemos con una definición de estructura simple:

```
struct datos {
   char nombre[10];
   struct datos *siguiente;
};
```

¿No ve algo raro en la definición? El segundo miembro del tipo `datos` es un apuntador al tipo `datos`. En otras palabras, la estructura incluye un apuntador a su propio tipo. Esto significa que una instancia de tipo `datos` puede apuntar a otra instancia del mismo tipo. Esta es la manera en que se crean los encadenamientos en una lista encadenada. Cada estructura apunta a la siguiente estructura de la lista. Esto se ilustra en la figura 11.7.

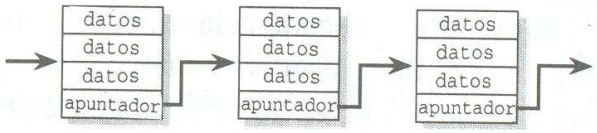


Figura 11.7. *La manera en que los encadenamientos se forman en una lista encadenada.*

Cada lista debe tener un inicio y un final. ¿Cómo son representados éstos en C? El inicio de la lista está marcado por el *apuntador de cabeza,* que apunta a la primera estructura de la lista. El apuntador de cabeza no es una estructura, sino simplemente un apuntador al tipo de dato que forma la lista. El final de la lista está marcado por una estructura que tiene un valor de apuntador de `NULL`. Debido a que todas las demás estructuras de la lista contienen un apuntador que no es `NULL`, que apunta al siguiente concepto de la lista, un valor de apuntador de `NULL` es una manera inequívoca para indicar el final de la lista. La estructura de una lista encadenada, con su apuntador de cabeza y su último elemento, se muestra en la figura 11.8.

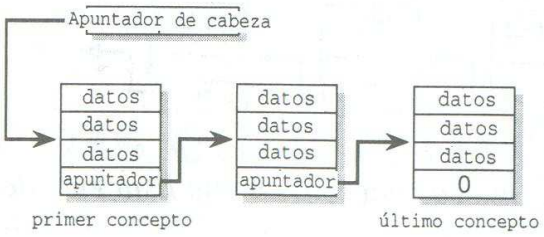


Figura 11.8. *El comienzo de una lista encadenada está indicado por un apuntador de cabeza, y el fin de la lista está indicado por un valor de apuntador de* `NULL`.

Pareciera que la lista encadenada no presenta ninguna ventaja real sobre un arreglo de estructuras. Sin embargo, hay varias ventajas significativas. Una tiene que ver con la inserción y borrado de elementos en una lista ordenada (que es una tarea común de

programación). Imagine que el programa mantiene una lista de 1,000 conceptos de datos en orden alfabético. Para mantener el orden los nuevos conceptos deben ser insertados en la posición adecuada de la lista: `Baez` entre `Arreola` y `Cervantes`, por ejemplo.

Si se está usando un arreglo, se debe hacer un hueco en la posición adecuada antes de poder insertar los nuevos datos. Digamos, por ejemplo, que `Arreola` está en la posición 5 y `Cervantes` en la posición 6. Para insertar a `Baez` en la posición adecuada (posición 6), `Cervantes` y todos los elementos de arreglo superiores deben ser movidos un espacio hacia arriba. ¡Esto requiere una gran cantidad de procesamiento! De manera similar, para borrar un concepto de en medio del arreglo, todos los elementos superiores necesitan ser movidos hacia abajo para llenar el hueco.

Las listas encadenadas facilitan la inserción y el borrado de conceptos de datos. Todo lo que se requiere es un poco de manipulación simple de apuntadores. No se necesita, de hecho, mover los datos. Continuando con el ejemplo anterior, se necesita insertar a `Baez` entre `Arreola` y `Cervantes`. En una lista encadenada, inicialmente el registro de `Arreola` apunta al registro de `Cervantes`. Para insertar el nuevo concepto, `Baez`, todo lo que se necesita es hacer que el registro de `Arreola` apunte al registro de `Baez` y que el registro de `Baez` apunte al registro de `Cervantes`. Este procedimiento es ilustrado en la figura 11.9. (El procedimiento para borrar un concepto es igualmente simple, ¡pero, imaginarse cómo, queda como un ejercicio para usted!).

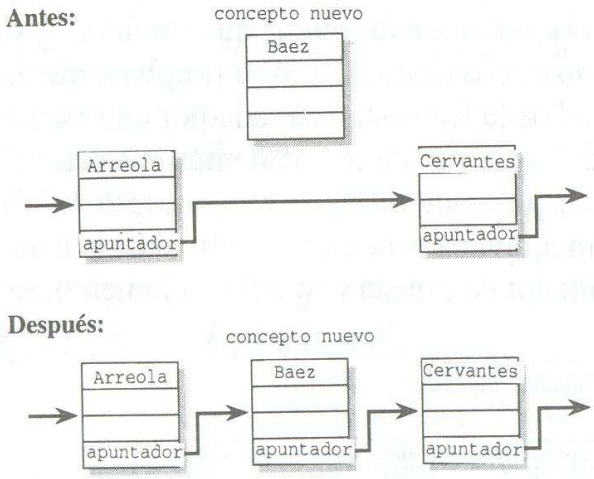


Figura 11.9. *Inserción de un nuevo concepto en una lista encadenada ordenada.*

Otra ventaja de las listas encadenadas se refiere al espacio de almacenamiento. Para usar un arreglo, su tamaño (cantidad de elementos) debe ser definido cuando el programa es compilado. Si, durante la ejecución, el programa se queda sin espacio de memoria, no hay nada que pueda hacerse. Sin embargo, con una lista encadenada el programa puede asignar espacio de almacenamiento para las estructuras conforme lo necesite, llegando hasta el límite impuesto por el hardware. Esto se hace con la función del C `malloc()`, tratada a continuación.

La función *malloc()*

Ya aprendió acerca de `malloc()`, la función del C para asignación de memoria, en el Día 10, "Caracteres y cadenas". El uso de `malloc()` para asignar espacio para una estructura es esencialmente idéntico al uso para asignar espacio para tipo `char`. Si se ha definido un tipo de estructura llamado `datos`, primero se declara un apuntador al tipo

```
struct datos *ptr;
```

y luego se llama a `malloc()`

```
ptr = malloc(sizeof(struct datos));
```

Si `malloc()` funciona satisfactoriamente, asigna un bloque de memoria del tamaño adecuado y regresa un apuntador a él. Si `malloc()` falla, es decir, que no hay suficiente memoria disponible, regresa `NULL` (y un programa real debe hacer prueba de ello).

El valor regresado por `malloc()` ha sido asignado a `ptr`, un apuntador al tipo `datos`. Usando este apuntador y el operador de membresía indirecta (`->`) se puede accesar la memoria asignada de acuerdo con los miembros de datos. Por lo tanto, si el tipo `datos` tiene un miembro de tipo `float` llamado `valor`, se podría escribir

```
ptr ->valor = 1.205;
```

Las estructuras asignadas con `malloc()` no tienen nombre *por sí mismas,* por lo que nunca puede ser usado el operador de membresía (.) para accesar sus miembros. Se debe utilizar siempre un apuntador y el operador de membresía indirecta (`->`).

Implementación de una lista encadenada

La información dada en este capítulo debe serle suficiente para que implemente un programa de lista encadenada. Este podría ser un ejercicio de programación excelente. Si usted tiene éxito en la escritura de un programa que usa listas encadenadas para guardar datos, se puede sentir muy satisfecho, ¡ya que va por buen camino para llegar a ser un programador de C eficiente!

typedef y las estructuras

Se puede usar la palabra clave `typedef` para crear un sinónimo para un tipo de estructura o unión. Por ejemplo, los enunciados

```
typedef struct {
   int x;
   int y;
} coord;
```

definen `coord` como sinónimo para la estructura indicada. Luego, se pueden declarar instancias de esta estructura usando el identificador `coord`.

```
coord arribaizquierda, abajoderecha;
```

Observe que `typedef` no es lo mismo que una etiqueta de estructura, como se describió anteriormente en este capítulo. Si se escribe

```
struct coord {
   int x;
   int y;
};
```

el identificador `coord` es una etiqueta para la estructura. Se puede usar la etiqueta para declarar instancias de la estructura, pero, a diferencia del `typedef`, se debe incluir la palabra clave `struct`:

```
struct coord arribaizquierda, abajoderecha;
```

El que se use `typedef` o la etiqueta de estructura para declarar a las estructuras casi no tiene diferencia práctica. El usar `typedef` da como resultado un código ligeramente más conciso, debido a que no se necesita usar la palabra clave `struct`. Por otro lado, usar una etiqueta y tener explícita la palabra clave `struct` hace más claro que se está declarando una estructura.

Resumen

Este capítulo le mostró la manera de usar estructuras, un tipo de dato que el programador diseña para satisfacer las necesidades de su programa. Una estructura puede contener cualquiera de los tipos de datos del C, incluyendo otras estructuras, apuntadores y arreglos. Cada concepto de datos dentro de una estructura, llamado un miembro, es accesado usando el operador de miembro de estructura (.) entre el nombre de la estructura y el nombre del miembro. Las estructuras pueden usarse individualmente y también pueden usarse en arreglos.

Los apuntadores a estructuras abren mayores posiblidades. Se pueden usar apuntadores para crear listas encadenadas de estructuras, en las cuales cada elemento de la lista está encadenado al siguiente por medio de un apuntador. Combinando listas encadenadas con la función `malloc()`, un programa puede asignar dinámicamente espacio de almacenamiento conforme lo necesite.

Las uniones fueron presentadas como similares a las estructuras. La principal diferencia entre una unión y una estructura es que la unión guarda a todos sus miembros en la misma área. Esto significa que sólo un miembro individual de la unión puede ser usado en un momento dado.

Preguntas y respuestas

1. ¿Tiene algún objeto declarar una estructura sin ninguna instancia?

   Se mostraron dos maneras de declarar estructuras. La primera fue declarar un cuerpo de estructura, una etiqueta y una instancia al mismo tiempo. La segunda fue declarar un cuerpo de estructura y una etiqueta sin una instancia. Posteriormente puede ser declarada una instancia usando la palabra clave struct, la etiqueta y un nombre para la instancia. Es una práctica común de programación usar el segundo método. Muchos programadores declararán el cuerpo de la estructura y su etiqueta sin ninguna instancia. Las instancias serán declaradas posteriormente en el programa. En el siguiente capítulo se describe el alcance de las variables. El alcance se aplica a la instancia, pero no a la etiqueta o al cuerpo de la estructura.

2. ¿Qué es más común: usar un `typedef` o una etiqueta de estructura?

   Muchos programadores usan `typedef` para hacer que su código sea más fácil de leer, mas, sin embargo, tiene poca diferencia práctica. Se pueden comprar muchas bibliotecas que contienen funciones. Estos productos adicionales tiene muchos `typedef` para hacer único al producto. Esto es especialmente cierto sobre los productos de bases de datos adicionales.

3. ¿Puedo simplemente asignar una estructura a otra con el operador de asignación?

   ¡Sí y no! Las versiones más recientes de los compiladores de C le permitirán asignar una estructura a otra, aunque puede ser que las versiones antiguas no lo permitan. ¡En las versiones antiguas del C tal vez necesite asignar cada miembro de la estructura individualmente! Esto también es cierto para las uniones.

4. ¿Qué tan grande es una unión?

   Debido a que cada uno de los miembros en una unión está guardado en la misma posición de memoria, la cantidad de espacio requerido para guardar la unión es igual al de su miembro más grande.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Qué tan diferente es una estructura de un arreglo?
2. ¿Cuál es el operador de miembro de estructura y para qué sirve?
3. ¿Qué palabra clave se usa en C para crear una estructura?
4. ¿Cuál es la diferencia entre una etiqueta de estructura y una instancia de estructura?
5. ¿Qué hace el siguiente fragmento de código?

```
struct address {
    char name[31];
    char add1[31];
    char add2[31];
    char city[11];
    char state[3];
    char zip[11];
} myaddress = { "Bradley Jones",
                "RTSoftware",
                "P.O. Box 1213",
                "Carmel", "IN", "46032-1213"};
```

6. Suponga que ha declarado un arreglo de estructuras y que ptr es un apuntador al primer elemento del arreglo (esto es, la primera estructura del arreglo). ¿Cómo podría cambiar ptr para que apuntara al segundo elemento del arreglo?
7. ¿Cuál es el único miembro necesario en un tipo de estructura que es usado en una lista encadenada?
8. ¿Cuáles son las dos ventajas de usar una lista encadenada en vez de un arreglo?
9. ¿Cuál es el argumento y el valor de retorno de malloc()?
10. ¿Con qué método se borra un concepto de una lista encadenada?

Ejercicios

1. Escriba el código que define a una estructura llamada tiempo que contiene tres miembros int.
2. Escriba el código que: a) define una estructura llamada datos, que contiene un miembro tipo int y dos miembros tipo float y b) declara una instancia de tipo datos llamada info.

3. Continuando con la pregunta 2, ¿cómo asignaría el valor de 100 al miembro int de la estructura `info`?

4. Escriba el código que declara e inicializa un apuntador a info.

5. Continuando con la pregunta 4, mueste dos maneras de usar notación de apuntadores para asignar el valor 5.5 al primer miembro float de info.

6. Escriba la definición para un tipo de estructura llamado datos, que pueda guardar una sola cadena de hasta 20 caracteres y que pueda ser usado en una lista encadenada.

7. Cree una estructura que contenga cinco cadenas, `dirección1`, `dirección2`, `ciudad`, `estado` y `código_postal`. Cree un `typedef`, llamado REGISTRO, que pueda ser usado para crear instancias de esta estructura.

8. Usando el `typedef` del ejercicio 7, asigne e inicialice un elemento llamado `midirección`.

9. BUSQUEDA DE ERRORES: ¿Qué hay de erróneo en el siguiente fragmento de código?

```
struct {
    char zodiac_sign[21];
    int month;
} sign = "Leo", 8;
```

10. BUSQUEDA DE ERRORES: ¿Qué hay de erróneo en el siguiente fragmento de código?

```
/* setting up a union */
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW", 1000 };
```

11. BUSQUEDA DE ERRORES: ¿Qué hay de erróneo en el siguiente fragmento de código?

```
/* a structure to be used in a linked list */
struct data{
    char firstname[10];
    char lastname[10];
    char middlename[10];
    char *next_name;
}
```

DIA 12

Alcance de las variables

En el Día 5, "Funciones: lo básico", se vio que una variable definida dentro de una función es diferente de una variable definida fuera de una función. Sin que usted se diera cuenta, se le presentó el concepto de *alcance de las variables*, un aspecto importante de la programación en C. Hoy aprenderá

- Acerca del alcance y el porqué es importante.
- Qué son las variables externas y por qué se debe, por lo general, evitarlas.
- Acerca de las variables locales.
- La diferencia entre variables estáticas y automáticas.
- Acerca de las variables locales y los bloques.
- La manera de seleccionar una clase de almacenamiento.

¿Qué es el alcance?

El *alcance* de una variable se refiere a la extensión sobre la cual partes diferentes de un programa tienen acceso a la variable, o donde es *visible* la variable. Cuando se hace referencia a las variables del C, los términos *accesibilidad* y *visibilidad* se usan indistintamente. Cuando se habla acerca del alcance, el término *variable* se refiere a todos los tipos de datos del C: variables simples, arreglos, estructuras, apuntadores, etc. También se refiere a las constantes simbólicas definidas con la palabra clave `const`.

El alcance también afecta el tiempo de vida de la variable: la cantidad de tiempo que la variable persiste en memoria, o cuando es asignado y desasignado el espacio de almacenamiento de la variable. En primer lugar este capítulo examina la visibilidad.

Una demostración del alcance

Véase el programa del listado 12.1. En él se define la variable `x` en la línea 5, se usa `printf()` para desplegar el valor de `x` en la línea 11 y luego se llama a la función `print_value()` para desplegar el valor de `x` nuevamente. Observe que a la función `print_value()` no se le pasa el valor de `x` como argumento. Simplemente usa `x` como argumento para `printf()`.

Captura

Listado 12.1. La variable `x` es accesible desde adentro de la función `print_value`.

```
1: /* Ilustra el alcance de variables. */
2:
3: #include <stdio.h>
4:
```

```
5: int x = 999;
6:
7: void print_value(void);
8:
9: main()
10: {
11:     printf("%d\n", x);
12:     print_value();
13: }
14:
15: void print"value(void)
16: {
17:     printf("%d\n", x);
18: }
```

Salida

```
999
999
```

Análisis

El programa en el listado 12.1 compila y ejecuta sin problemas. Ahora haga un modificación pequeña en el programa, moviendo la definición de la variable x a una posición dentro de la función main(). El nuevo código fuente se muestra en el listado 12.2.

Captura

Listado 12.2. La variable x no es accesible desde adentro de la función `print_value`.

```
1: /* Ilustra el alcance de variables. */
2:
3: #include <stdio.h>
4:
5: void print_value(void);
6:
7: main()
8: {
9:     int x = 999;
10:
11:     printf("%d\n", x);
12:     print_value();
13: }
14:
15: void print_value(void)
16: {
17:     printf("%d\n", x);
18: }
```

Análisis

Si se trata de compilar el listado 12.2, el compilador genera un mensaje de error similar al siguiente:

```
list1202.c(17) : Error: undefined identifier 'x',
```

Recuerde que en un mensaje de error el número en paréntesis se refiere a la línea de programa donde encontró el error. La línea 17 es la llamada a `printf()` dentro de la función `print_value()`.

Este mensaje de error le dice que dentro de la función `print_value()` la variable `x` está "indefinida" o, en otras palabras, no visible. Observe, sin embargo, que la llamada a `printf()` en la línea 11 no genera un mensaje de error, ya que en esta parte del programa la variable `x` es visible.

La única diferencia entre el listado 11.1 y el listado 11.2 es la posición donde está definida la variable `x`. Moviendo la definición de `x` se cambia su alcance. En el listado 12.1 `x` es una variable *externa* y su alcance es el programa completo. Es accesible desde la función `main()` y desde la función `print_value()`. En el listado 12.2 `x` es una variable *local* y su alcance está limitado al interior de la función `main()`. Por lo que toca a `print_value()`, `x` no existe. Posteriormente, en este capítulo, aprenderá más acerca de las variables locales y externas, pero primero necesita entender la importancia del alcance.

¿Por qué es importante el alcance?

Para entender la importancia del alcance de las variables, necesita recordar lo que se dijo de la programación estructurada en el Día 5, "Funciones: lo básico". El enfoque estructurado, tal vez recuerde, divide al programa en funciones independientes que ejecutan una tarea específica. Aquí la palabra clave es *independiente*. Para que haya una verdadera independencia es necesario que las variables de cada función estén aisladas de interferencias causadas por otras funciones. Solamente mediante el aislamiento de los datos de cada función se puede estar seguro de que la función hará su trabajo sin que otra parte del programa la interfiera.

Tal vez piense que el aislamiento completo de los datos entre funciones no es siempre deseable, y tiene razón. Pronto se dará cuenta de que mediante la especificación del alcance de las variables, un programador tiene un gran control sobre el nivel de aislamiento de los datos.

Variables externas

Una variable *externa* es aquella que está definida fuera de cualquier función. Esto significa también fuera de `main()`, debido a que `main()` también es una función. Hasta ahora la mayoría de las definiciones de variables en este libro han sido externas, poniéndolas en el código fuente antes del inicio de `main()`. Las variables externas son llamadas, algunas

veces, variables *globales*. Si no se inicializa explícitamente a una variable externa cuando es definida, el compilador la inicializa a 0.

Alcance de las variables externas

El alcance de las variables externas es el programa completo. Esto significa que una variable externa es visible a todo lo largo de `main()` y de cualquier otra función que se encuentra en el programa. Por ejemplo, la variable `x` en el listado 12.1 es una variable externa. Tal como usted vio cuando compiló y ejecutó el programa, `x` es visible dentro de ambas funciones, `main()` y `print_value()`.

Sin embargo, hablando estrictamente, no es correcto decir que el alcance de una variable externa es el programa completo. El alcance es, propiamente, el archivo de código fuente completo que contiene la definición de variable. Si el programa completo está contenido en un solo archivo de código fuente, las dos definiciones de alcance son equivalentes. La mayoría de los programas en C, pequeños y medianos, están contenidos en un archivo, y esto es completamente cierto acerca de los programas que se están escribiendo ahora.

Sin embargo, es posible que el código fuente de un programa esté contenido en dos o más archivos separados. Aprenderá la manera y el porqué se hace esto en el Día 21, "Aprovechando las directivas del preprocesador y más", y qué manejo especial se requiere en estas situaciones para las variables externas.

Cuándo usar variables externas

Aunque los programas de ejemplo hasta este momento han usado variables externas, en la práctica actual debe usarlas rara vez. ¿Por qué? Debido a que, cuando se usan variables externas, se está violando el principio de *independencia modular,* básico para la programación estructurada. La independencia modular se refiere a que cada función o módulo en un programa contiene todo el código y los datos que necesita para hacer su trabajo. Con los relativamente pequeños programas que está escribiendo ahora tal vez no parezca importante, pero cuando avance hacia programas más grandes y complejos la dependencia excesiva en las variables externas puede comenzar a dar problemas.

¿Cuándo se deben usar variables externas? Haga que una variable sea externa solamente cuando todas o la mayoría de las funciones del programa necesiten acceso a la variable. Las constantes simbólicas definidas con la palabra clave `const` son, por lo general, buenos candidatos para ser externas. Si solamente algunas de las funciones necesitan acceso a una variable, pásela como argumento a las funciones en vez de hacerla externa.

DEBE	NO DEBE
DEBE Usar variables locales para conceptos como apuntadores de ciclo.	
NO DEBE Usar variables externas si no son necesarias para la mayoría de las funciones del programa.	
DEBE Usar variables locales para aislar los valores que contienen las variables del resto del programa.	

La palabra clave *extern*

Cuando una función usa una variable externa, es una buena práctica de programación declarar la variable dentro de la función, con la palabra clave `extern`. La declaración toma la forma

```
extern tipo nombre;
```

donde `tipo` es el tipo de la variable, y `nombre` el nombre de la variable. Por ejemplo, se podría añadir la declaración de `x` a las funciones `main()` y `print_value()` del listado 12.1. El programa resultante se muestra en el listado 12.3.

Captura

Listado 12.3. La variable externa `x` es declarada como `extern` dentro de las funciones `main()` y `print_value`.

```
1: /* Ilustra la declaración de variables externas. */
2:
3: #include <stdio.h>
4:
5: int x = 999;
6:
7: void print_value(void);
8:
9: main()
10: {
11:     extern int x;
12:
13:     printf("%d", x);
14:     print_value();
15: }
16:
17: void print_value(void)
18: {
19:
```

```
20:     extern int x;
21:     printf("%d", x);
22: }
```

Salida

```
999999
```

Análisis

Este programa imprime el valor de `x` dos veces, primero en la línea 13, como parte de `main()`, y luego en la línea 21, como parte de `print_value()`. La línea 5 define a `x` como una variable tipo int igual a 999. Las líneas 11 y 20 declaran a `x` como extern `int`. Observe la distinción entre la definición de la variable, que reserva espacio de almacenamiento para la variable, y la declaración `extern`. Esto último dice: "esta función usa una variable externa con tal y cual nombre y tipo, y está definida en cualquier otro lugar".

Variables locales

Una variable local es aquella que está definida dentro de una función. El alcance de una variable local está limitado a la función en la cual se define. El Día 5, "Funciones: lo básico", describe las variables locales dentro de funciones, la manera de definirlas y cuáles son sus ventajas. Las variables locales no son inicializadas automáticamente a 0 por el compilador. Si no se inicializa a una variable local cuando es definida, tiene un valor indefinido, o *basura*. Se debe asignar explícitamente un valor a las variables locales antes de que sean usadas por primera vez.

Una variable también puede ser local a la función `main()`. Este es el caso de `x` en el listado 12.2. Está definida dentro de `main()` y, como lo ilustra la compilación y ejecución de ese programa, también es visible solamente dentro de `main()`.

Variables estáticas versus automáticas

Las variables locales son automáticas por omisión. Esto significa que las variables locales son creadas nuevas cada vez que es llamada la función, y son destruidas cuando la ejecución sale de la función. Lo que esto significa en términos prácticos es que una variable automática no guarda su valor entre llamadas a la función en la cual está definida.

Supongamos que el programa tiene una función que usa una variable local `x`. También supongamos que la primera vez que es llamada la función asigna el valor de 100 a `x`. La ejecución regresa al programa llamador y, posteriormente, nuevamente es llamada la función. ¿Todavía guarda el valor 100 la variable `x`? No. La primera instancia de la variable

x fue destruida cuando la ejecución salió de la función, después de la primera llamada. Cuando la función fue vuelta a llamar fue creada una nueva instancia de x. La x anterior desapareció para siempre.

¿Qué pasa si la función necesita guardar el valor de la variable local entre llamadas? Por ejemplo, una función de impresión tal vez necesite recordar la cantidad de renglones que ya ha enviado a la impresora, para determinar si se necesita un salto de hoja. Para que una variable local mantenga su valor entre llamadas, debe ser definida como *estática*, con la palabra clave `static`. Por ejemplo,

```
void func1(int x)
{
   static int a;
...
}
```

El programa del listado 12.4 ilustra la diferencia entre variables locales automáticas y estáticas.



Listado 12.4. Demuestra la diferencia entre las variables locales automáticas y estáticas.

```
1: /* Demuestra las variables locales automáticas y estáticas. */
2: #include <stdio.h>
3: void func1(void);
4: main()
5: {
6:     int count;
7:
8:     for (count = 0; count < 20; count++)
9:     {
10:        printf("At iteration %d: ", count);
11:        func1();
12:    }
13: }
14:
15: void func1(void)
16: {
17:    static int x = 0;
18:    int y = 0;
19:
20:    printf("x = %d, y = %d\n", x++, y++);
21: }
```




```
At iteration 0: x = 0, y = 0
At iteration 1: x = 1, y = 0
At iteration 2: x = 2, y = 0
At iteration 3: x = 3, y = 0
```

```
At iteration 4: x = 4, y = 0
At iteration 5: x = 5, y = 0
At iteration 6: x = 6, y = 0
At iteration 7: x = 7, y = 0
At iteration 8: x = 8, y = 0
At iteration 9: x = 9, y = 0
At iteration 10: x = 10, y = 0
At iteration 11: x = 11, y = 0
At iteration 12: x = 12, y = 0
At iteration 13: x = 13, y = 0
At iteration 14: x = 14, y = 0
At iteration 15: x = 15, y = 0
At iteration 16: x = 16, y = 0
At iteration 17: x = 17, y = 0
At iteration 18: x = 18, y = 0
At iteration 19: x = 19, y = 0
```

Análisis Este programa tiene una función que define e inicializa una variable de cada tipo. Esta función es `func1()`, en las líneas 15 a 21. Cada vez que es llamada la función, ambas variables son desplegadas en la pantalla e incrementadas (línea 20). La función `main()`, en las líneas 4 a 13, contiene un ciclo `for` (líneas 8 a 12) que imprime un mensaje (línea 10) y luego llama a `func1()` (línea 11). El ciclo `for` repite 20 veces.

En el listado de salida anterior, observe que `x`, la variable estática, se incrementa con cada iteración, debido a que retiene su valor entre llamadas. Por otro lado, la variable automática y es reinicializada a 0 en cada llamada.

Este programa también ilustra una diferencia en la manera en que es manejada la inicialización explícita de variables (esto es, el momento en que una variable es inicializada al momento de la definición). Una variable estática es inicializada solamente la primera vez que la función es llamada. En llamadas posteriores el programa "recuerda" que la variable ya ha sido inicializada y no la vuelve a inicializar. En vez de ello, la variable guarda el valor que tenía cuando la ejecución salió de la función. Por el contrario, una variable automática es inicializada al valor especificado cada vez que la función es llamada.

Si se experimenta un poco con variables automáticas, puede obtener resultados que no concuerdan con lo que se ha leído aquí. Por ejemplo, si se modifica el programa del listado 12.4 en forma tal que las dos variables locales no sean inicializadas cuando se les define, la función `func1()` en las líneas 15 a 21 dirá

```
15: void func1(void)
16: {
17:    static int x;
18:    int y;
19:
20:    printf("x = %d, y = %d\n", x++, y++);
21: }
```

Cuando ejecute el programa modificado tal vez encuentre que el valor de `y` se incrementa en 1 en cada iteración. Esto significa que `y` está guardando su valor entre llamadas a la función. ¿Son mentiras lo que ha leído aquí acerca de que las variables automáticas pierden su valor?

No, lo que ha leído es cierto (¡no dude!). Si obtiene los resultados descritos anteriormente, donde una variable automática guarda su valor en llamadas repetidas a la función, simplemente es por casualidad. Esto es lo que pasa. Cada vez que la función es llamada, se crea una nueva `y`. Tal vez el compilador haya usado la misma posición de memoria para la nueva `y` que la que usó para la `y` la vez anterior que se llamó a la función. Si `y` no es inicializada explícitamente por la función, el espacio de almacenamiento puede contener el valor que `y` tenía durante la llamada anterior. Parece ser que la variable guarda su valor anterior, pero esto es por casualidad. ¡No se puede esperar que esto suceda siempre!

Debido a que las variables locales son automáticas por omisión no necesita ser especificada en la definición de variable. Si lo desea, puede incluir la palabra clave `auto` en la definición, antes de la palabra clave de tipo, como se muestra aquí:

```
void func1(int y)
{
   auto int contador;
   /* Aquí va código adicional */
}
```

El alcance de los parámetros de la función

Una variable que está contenida en la lista de parámetros del encabezado de función tiene un alcance *local*. Por ejemplo, vea la siguiente función:

```
void func1(int x)
{
   int y;
   /* Aquí va código adicional */
}
```

Tanto `x` como `y` son variables locales, con un alcance que es la función completa `func1()`. Por supuesto, `x` inicialmente contiene cualquier valor que haya sido pasado a la función por el programa llamador. Cuando se usa ese valor, se puede usar `x` de manera similar a cualquier otra variable local.

Debido a que las variables de parámetro siempre comienzan con el valor pasado como el argumento correspondiente, no tiene sentido pensar que sean estáticas o automáticas.

Variables estáticas externas

Una variable externa puede ser hecha estática incluyendo en su definición la palabra clave `static`.

```
static float tasa;

main()
{
/* Aquí va código adicional */
}.
```

La diferencia entre una variable externa ordinaria y una variable externa estática reside en el alcance. Una variable externa ordinaria es visible para todas las funciones del archivo, y puede usarse por funciones en otros archivos. Una variable externa estática es visible solamente para las funciones de su propio archivo y por abajo de su punto de definición.

La distinción anterior se aplica obviamente en primer lugar a los programas que tienen código fuente contenido en dos o más archivos. Este tema se trata en el Día 21, "Aprovechando las directivas del preprocesador y más".

Variables de registro

La palabra clave `register` se usa para sugerirle al compilador que una variable local automática sea guardada en un registro del procesador en vez de la memoria regular. ¿Qué es un *registro del procesador* y qué ventajas tiene usarlo?

La unidad central de proceso, o CPU, de la computadora contiene unas cuantas posiciones de almacenamiento de datos, llamados *registros.* Es en estos registros de la CPU donde se efectúan las operaciones de los datos, como la suma y división. Para manejar los datos la CPU, debe moverlos de memoria a sus registros, ejecutar las manipulaciones y luego regresarlas a memoria. Mover datos de y hacia memoria se lleva un cierto tiempo. Si una variable en particular puede ser guardada desde el principio en un registro, el manejo de la variable puede realizarse más rápido.

Con la palabra clave `register` en la definición de una variable automática, se le pide al compilador que guarde esa variable en un registro. Vea el siguiente ejemplo:

```
void func1(void)
{
   register int x;
   /* Aquí va código adicional*/
}
```

Observe que decimos "pedir" y no "decir". Dependiendo de las necesidades del programa, puede ser que un registro no esté disponible para la variable. En este caso el compilador la trata como una variable automática ordinaria. La palabra clave `register` es una sugerencia ¡y no una orden! Los beneficios de la clase de almacenamiento `register` son mayores para las variables que se usan frecuentemente por la función, como la variable de contador de un ciclo.

La palabra clave `register` puede usarse solamente con variables numéricas simples y no con arreglos o estructuras. Tampoco puede usarse con las clases de almacenamiento estático y externo. No se puede definir un apuntador a una variable de registro.

DEBE	NO DEBE
DEBE Inicializar las variables locales o no sabrá qué valor contienen.	
DEBE Inicializar las variables globales, aunque piense que están inicializadas a 0 por omisión. Si siempre inicializa sus variables se evitará problemas, como olvidar la inicialización de variables locales.	
DEBE Pasar variables locales como parámetros de función, en vez de declararlos como globales, si se necesitan solamente en unas cuantas funciones.	
	NO DEBE Usar variables de registro para valores no numéricos, estructuras o arreglos.

Variables locales y la función *main()*

Todo lo que ha sido dicho acerca de las variables locales se aplica a `main()`, así como a todas las demás funciones. Hablando estrictamente, `main()` es una función como cualquier otra. La función `main()` es llamada cuando el programa es arrancado desde el DOS, y el control regresa al DOS desde `main()` cuando el programa termina.

Esto significa que las variables locales definidas en `main()` son creadas cuando el programa se inicia, y su tiempo de vida termina cuando termina el programa. La noción de una variable local estática, reteniendo su valor entre llamadas a `main()`, en realidad no tiene sentido: una variable no puede seguir existiendo entre diferentes ejecuciones del programa. Por lo tanto, dentro de `main()` no hay diferencia entre variables locales estáticas y automáticas. En `main()` se puede definir una variable local como estática, pero no tiene sentido.

DEBE	NO DEBE

DEBE Recordar que main() es, en muchos aspectos, una función similar a cualquier otra función.

NO DEBE Declarar variables estáticas en main(), ya que al hacerlo no se gana nada.

¿Qué clase de almacenamiento se debe usar?

Cuando esté tratando de determinar qué clase de almacenamiento usar para alguna variable del programa en particular, puede ayudarle el ver la tabla 12.1, que resume las cinco clases de almacenamiento disponibles en C.

Tabla 12.1. Las cinco clases de almacenamiento de variables del C.

Clase de almacenam.	Palabra clave	Tiempo de vida	Dónde se define	Alcance
Automático	`Ninguna` [1]	Temporal	En una función	Local
Estático	`static`	Temporal	En una función	Local
Registro	`register`	Temporal	En una función	Local
Externo	`Ninguna` [2]	Permanente	Fuera de una función	Global (para todos los archivos)
Externo Estático	`static`	Permanente	Fuera de una función	Global (para un solo archivo)

[1] La palabra clave `auto` es opcional.

[2] La palabra clave `extern` se usa en funciones para declarar una variable externa estática que está definida en cualquier lado.

Para determinar la clase de almacenamiento, trate de usar la clase de almacenamiento automático siempre que sea posible, y use las otras clases solamente cuando se necesite. A continuación se presentan algunas reglas que seguir:

- Comience dando a cada variable la clase de almacenamiento local automático.
- Si la variable es manejada frecuentemente, añada a su definición la palabra clave register.
- En funciones diferentes a main(), haga que la variable sea estática si su valor debe conservarse entre llamadas a la función.
- Si la variable se usa por la mayoría, o por todas las funciones del programa, defínala con la clase de almacenamiento extern.

Variables locales y bloques

Hasta ahora este capítulo ha tratado solamente las variables que son locales a la función. Esta es la primera forma en que se usan las variables locales, pero se pueden definir variables que son locales a cualquier bloque de programa (cualquier sección encerrada entre llaves). Cuando declare variables dentro de un bloque, debe recordar que la declaración debe ser primero. Para un ejemplo, véase el listado 12.5.



Listado 12.5. Un programa que demuestra la definición de variables locales dentro de un bloque de programa.

```
1: /* Demuestra las variables locales dentro de bloques. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     /* Define una variable local a main(). */
8:
9:     int count = 0;
10:
11:     printf("\nOutside the block, count = %d", count);
12:
13:     /* Inicia un bloque. */
14:     {
15:     /* Define una variable local al bloque. */
16:
17:     int count = 999;
18:     printf("\nWithin the block, count = %d", count);
19:     }
20:
21:     printf("\nOutside the block again, count = %d", count);
22: }
```

La salida del programa es




```
Outside the block, count = 0
Within the block, count = 999
Outside the block again, count = 0
```



En este programa puede ver que el contador count definido dentro del bloque es independiente del `count` definido fuera del bloque. La línea 9 define a `count` como una variable tipo `int` igual a 0. Debido a que es declarada al inicio de `main()` puede usarse por toda la función `main()`. La línea 11 muestra, imprimiendo un mensaje, que `count` ha sido inicializada a 0 . En las líneas 14 a 19 es declarado un bloque, y dentro del bloque está definida otra variable `count` como variable tipo int. Esta variable `count` es inicializada a 999 (línea 17). La línea 18 imprime el valor de 999 de la variable `count` del bloque. Debido a que el bloque termina en la línea 19, el enunciado de impresión de la línea 21 usa el `count` original, declarado inicialmente en la línea 9 de `main()`.

El uso de este tipo de variable local no es común en la programación en C y tal vez nunca encuentre necesidad de hacerlo. Su uso más común es, probablemente, cuando un programador trata de aislar un problema dentro de un programa. Puede aislar temporalmente secciones de código poniéndolas entre llaves, y establecer variables locales para ayudarse en la localización de la falla. Otra ventaja es que la declaración-inicialización de la variable puede ser puesta cercana al punto donde se usa, lo que puede ayudar en la comprensión del programa.

DEBE	NO DEBE

NO DEBE Tratar de poner definiciones de variable en cualquier lugar dentro de una función que no sea al inicio de la función o al inicio de un bloque.

NO DEBE Usar variables al principio de un bloque, a menos que sirvan para clarificar el programa.

DEBE Usar variables al comienzo de un bloque (temporalmente) para ayudarle a encontrar problemas.

Resumen

Este capítulo trató las clases de almacenamiento de variables del C. Cada variable del C, ya sea una variable simple, un arreglo, una estructura o cualquier otra cosa, tiene una clase de almacenamiento específica, que determina: 1) su alcance o visibilidad dentro del programa, y 2) su tiempo de vida o qué tanto tiempo persiste la variable en memoria.

El uso adecuado de las clases de almacenamiento es un aspecto importante de la programación estructurada. Haciendo que la mayoría de las variables sean locales a la función que las usa se mejora la independencia entre funciones. A una variable debe dársele la clase de almacenamiento automático, a menos que haya una razón específica para hacerla externa o estática.

Preguntas y respuestas

1. Si las variables globales pueden usarse en cualquier parte del programa, ¿por qué no hacer que todas las variables sean globales?

   Conforme el programa se hace más grande, usted comenzará a declarar más y más variables. Como se dijo en el capítulo, hay límites sobre la cantidad de memoria disponible. Las variables declaradas como globales consumen memoria durante todo el tiempo que el programa esté ejecutando; sin embargo, las variables locales no lo hacen. En su mayor parte, las variables locales ocupan memoria solamente mientras la función de la que son locales se encuentra activa. (Una variable estática ocupa memoria desde del momento en que se usa por primera vez hasta el final del programa.) Adicionalmente, las variables globales están sujetas a alteración inadvertida por otras funciones. Si esto sucede, las variables tal vez no contengan los valores que se espera cuando se usan en las funciones para las que fueron creadas.

2. El Día 11, "Estructuras", definió que el alcance afecta una instancia de estructura, pero no la etiqueta o el cuerpo de la estructura. ¿Por qué el alcance no afecta la etiqueta o el cuerpo de la estructura?

   Cuando se declara una estructura sin instancias, se está creando una plantilla. De hecho, no se declara ninguna variable. No es sino hasta que se crea una instancia de la estructura cuando se declara una variable. Por esta razón se puede dejar que el cuerpo de la estructura sea externo a cualquier función, sin ningún efecto real sobre la memoria externa. Muchos programadores ponen los cuerpos de estructuras comúnmente usados con etiquetas en los archivos de encabezado, y luego incluyen estos archivos de encabezado cuando necesitan crear una instancia de la estructura. (Los archivos de encabezado se tratan en el Día 20, "Otras funciones".)

3. ¿Cómo sabe la computadora la diferencia entre una variable global y una local?

   La respuesta a esto está más allá del alcance de este capítulo. Lo que se debe saber es que cuando una variable local es declarada con el mismo nombre que una variable global, el programa ignora temporalmente la variable global. Continúa ignorando la variable global hasta que la variable local queda fuera de alcance.

4. ¿Se puede declarar una variable local con un tipo de variable diferente, y usar el mismo nombre de variable como variable global?

   Sí. Cuando se declara una variable local con el mismo nombre que una variable global, es una variable completamente diferente. Esto significa que se le puede hacer de cualquier tipo que se quiera. Sin embargo, se debe tener cuidado cuando se declaren variables globales y locales con el mismo nombre.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿De qué trata el alcance?
2. ¿Cuál es la diferencia más importante entre las clases de almacenamiento local y externo?
3. ¿Cómo afecta su clase de almacenamiento la posición de la definición de una variable?
4. Cuando se define una variable local, ¿cuáles son las dos opciones para el tiempo de vida de la variable?
5. El programa puede inicializar variables locales, tanto automáticas como estáticas, cuando son definidas. ¿Cuándo se efectúa la inicialización?
6. (Cierto o falso): Una variable de registro siempre será puesta en un registro.
7. ¿Qué valor contiene una variable global sin inicializar?
8. ¿Qué valor contiene una variable local sin inicializar?
9. ¿Qué imprimirá la línea 21 del listado 12.5 si se quitan las líneas 9 y 11? Piense acerca de esto y luego pruébelo con el programa para ver qué pasa.
10. Si una función "necesita recordar" el valor de una variable local tipo int entre llamadas, ¿cómo debería ser declarada la variable?
11. ¿Qué es lo que hace la palabra clave `extern`?
12. ¿Qué es lo que hace la palabra clave `static`?

Ejercicios

1. Escriba una declaración para que una variable sea puesta en un registro de la CPU.
2. Cambie el listado 12.2 para impedir el error. Hágalo sin usar ninguna variable externa.
3. Escriba un programa que declare una variable global de tipo int llamada `var`. Inicialice `var` a cualquier valor. El programa debe imprimir el valor de var en una función (que no sea `main()`). ¿Necesita pasar a `var` como parámetro a la función?
4. Cambie el programa del ejercicio 3. En vez de declarar a `var` como variable global, cámbiela a variable local en `main()`. El programa todavía debe imprimir a `var` en una función separada. ¿Necesita pasar a `var` como parámetro a la función?
5. ¿Puede un programa tener una variable global y una local con el mismo nombre? Escriba un programa que use una variable global y una local con el mismo nombre para probar su respuesta.
6. BUSQUEDA DE ERRORES: (la variable no es declarada al principio del bloque).

```
void a_sample_function( void )
{
    int ctr1;

    for ( ctr1 = 0; ctr1 < 25; ctr1++ )
        printf( _*_ );

    puts( "\nThis is a sample function" );
    {
        char star = '*';
        puts( "It has a problem" );
        for ( int ctr2 = 0; ctr2 < 25; ctr2++ )
        {
            printf( _%c_, star);
        }
    }
}
```

7. BUSQUEDA DE ERRORES: ¿qué hay de erróneo en el siguiente código?

```
/* count the number of even numbers from 0 to 100 */
   main()
   {
       int x = 1;
       static long tally = 99;

       for ( x = 0; x < 100; x++ )
           if( x % 2 == 0 )          /* if x is an even number...*/
               tally++;              /* then add 1 to tally!     */
   }
```

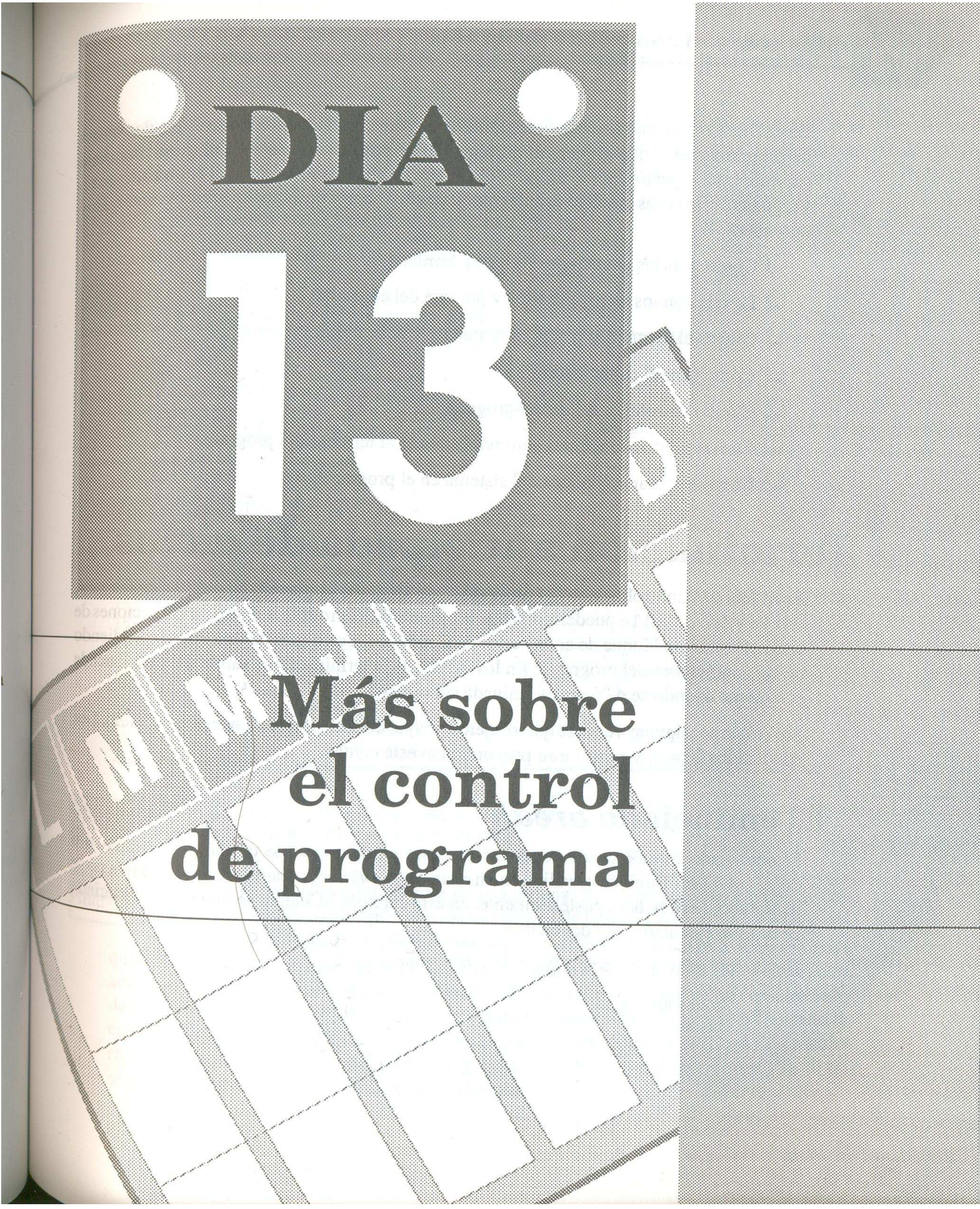
8. BUSQUEDA DE ERRORES: ¿hay algo erróneo en el siguiente programa?

```
#include <stdio.h>

void print_function( char star );

int ctr;

main()
{
    char star;

    print_function( star );
    return 0;
}

void print_function( char star )
{
    char dash;

    for ( ctr = 0; ctr < 25; ctr++ )
    {
        printf( "%c%c", star, dash );
    }
}
```

9. ¿Qué imprime el siguiente programa?

```
#include <stdio.h>
void print_letter2(void);          /* function prototype */

int ctr;
char letter1 = 'X';
char letter2 = '=';

main()
{
    for( ctr = 0; ctr < 10; ctr++ )
    {
        printf( "%c", letter1 )
        print_letter2();
    }
}

void print_letter2(void)
{
    for( ctr = 0; ctr < 2; ctr++ )
        printf( "%c", letter2 );
}
```

10. BUSQUEDA DE ERRORES: ¿Qué error hay en el programa anterior? Vuélvalo a escribir para que sea correcto.



Más sobre el control de programa

En el Día 6, "Control básico del programa", se presentaron algunos enunciados de control de programa del C que gobiernan la ejecución de otros enunciados en el programa. Este capítulo trata aspectos más avanzados de control de programa, incluido el enunciado `goto` y algunas de las cosas más interesantes que se pueden hacer con ciclos en los programas. Hoy aprenderá

- Cómo usar los enunciados `break` y `continue`.
- Lo que son los ciclos infinitos y por qué deben usarse.
- Qué es el enunciado `goto` y por qué se le debe evitar.
- Cómo usar el enunciado `switch`.
- Cómo controlar la salida del programa.
- Cómo ejecutar funciones automáticamente al término del programa.
- Cómo ejecutar comandos del sistema en el programa.

Terminación anticipada de ciclos

En el Día 6, "Control básico del programa", se aprendió la manera en que los ciclos `for`, `while` y `do...while` pueden controlar la ejecución del programa. Estas construcciones de ciclo ejecutan un bloque de enunciados de C ninguna, una o más de una vez, dependiendo de las condiciones del programa. En los tres casos la terminación o salida del ciclo sucede solamente cuando se da una determinada condición.

Sin embargo, algunas veces se quiere ejercer mayor control sobre la ejecución del ciclo. Los enunciados `break` y `continue` proporcionan este control.

El enunciado *break*

El enunciado `break` puede ponerse solamente en el cuerpo de un ciclo `for`, `while` o `do...while`. (También es válido en un enunciado `switch`, aunque ese tema todavía no se ha tratado, pero se hará posteriormente, en este capítulo.) Cuando se llega a un enunciado `break`, la ejecución sale del ciclo.

```
for ( contador = 0; contador < 10, contador++)
{
   if ( contador == 5)
        break;
}
```

Actuando solo, el ciclo `for` ejecutaría 10 veces. Sin embargo, en la sexta iteración contador es igual a 5, y se ejecuta el enunciado `break` haciendo que termine el ciclo `for`. La ejecución luego pasa al enunciado que se encuentra inmediatamente a continuación de la llave que cierra el ciclo `for`. Cuando un enunciado `break` es encontrado en el interior de un ciclo anidado, solamente causa la salida del ciclo más interno.

El programa en el listado 13.1 demuestra el uso de `break`.

Captura

Listado 13.1. Uso del enunciado `break`.

```
1: /* Demuestra un enunciado break. */
2:
3: #include <stdio.h>
4:
5: char s[] = "This is a test string. It contains two sentences.";
6:
7: main()
8: {
9:     int count;
10:
11:     printf("\nOriginal string: %s", s);
12:
13:     for (count = 0; s[count]!='\0'; count++)
14:         if (s[count] == '.')
15:         {
16:             s[count+1] = '\0';
17:             break;
18:         }
19:
20:     printf("\nModified string: %s", s);
21: }
```

Salida

```
Original string: This is a test string. It contains two sentences.
Modified string: This is a test string.
```

Análisis

El programa extrae la primera frase de una cadena. Busca en la cadena, carácter por carácter, el primer punto (que debe marcar el final de la frase). Esto se logra en el ciclo `for` en las líneas 13 a 18. La línea 13 inicia el ciclo `for`, incrementando a `count` para que vaya de carácter en carácter por la cadena `s`. La línea 14 revisa para ver si el carácter actual en la cadena es igual a un punto. Si lo es, se inserta inmediatamente un carácter nulo después del punto (línea 16). Esto, en efecto, corta la cadena. Una vez que se ha cortado la cadena ya no se necesita continuar el ciclo, por lo que un enunciado `break` (línea 17) termina rápidamente el ciclo y envía el control a la primera línea después del ciclo (línea 19). Si no se encuentra algún punto, la cadena no es alterada.

Un ciclo puede contener varios enunciados `break`. Sólo el primer `break` ejecutado (en caso de haber alguno) tiene efecto. Si ninguno se ejecuta, el ciclo termina normalmente (de acuerdo con su condición). La figura 13.1 muestra la operación del enunciado `break`.

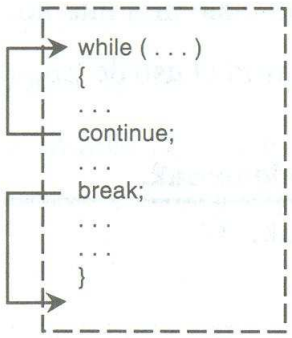


Figura 13.1. *Operación de los enunciados* `break` *y* `continue`.

Sintaxis

El enunciado *break*

```
break;
```

`break` se usa en el interior de un ciclo o de un enunciado `switch`. Hace que el control de programa brinque más allá del final del ciclo actual (`for, while o do...while`) o del enunciado `switch`. No ejecuta ninguna iteración más del ciclo, sino que se ejecuta el primer comando que se encuentre a continuación del ciclo o del enunciado `switch`.

Ejemplo

```
#include <stdio.h>

int x;
printf ( "Contando de 1 a 10\n");
/* al no tener condición en el ciclo for se hará que
   repita para siempre */
for( x = 1; ; x++ )
{
   if( x == 10 ) /* Esto revisa para el valor 10 */
      break;     /* Esto termina el ciclo      */
   printf( "\n%d, x );
}
```

El enunciado *continue*

De manera similar al enunciado `break`, el enunciado `continue` solamente puede ponerse en el cuerpo de un ciclo `for`, `while` o `do...while`. Cuando ejecuta un enunciado `continue`, comienza inmediatamente la siguiente iteración del ciclo que lo contiene. Los

enunciados que están entre el enunciado `continue` y el final del ciclo no se ejecutan. La operación de `continue` se muestra en la figura 13.1. Observe cómo difiere de la operación de un enunciado `break`.

Un programa que usa `continue` se presenta en el listado 13.2. El programa acepta una línea de entrada del teclado, y luego la despliega habiendo quitado todas la vocales minúsculas.

Listado 13.2. Demostración del enunciado `continue`.

```
1: /* Demuestra un enunciado continue. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     /* Declara un buffer para entrada y una variable de contador. */
8:
9:     char buffer[81];
10:     int ctr;
11:
12:     /* Recibe una línea de texto. */
13:
14:     puts("Enter a line of text:");
15:     gets(buffer);
16:
17:     /* Va por la cadena desplegando solamente */
18:     /* los caracteres que no son vocales minúsculas. */
19:
20:     for (ctr = 0; buffer[ctr] !='\0'; ctr++)
21:     {
22:
23:         /* Si el carácter es una vocal minúscula, */
24:         /* regrese al ciclo sin desplegarla. */
25:
26:         if (buffer[ctr] == 'a' || buffer[ctr] == 'e' ||
                buffer[ctr] == 'i'
27:             || buffer[ctr] == 'o' || buffer[ctr] == 'u')
28:             continue;
29:
30:         /* Si no es una vocal la despliega. */
31:
32:         putchar(buffer[ctr]);
33:     }
34: }
```

Salida

```
Enter a line of text:
This is a line of text
Ths s  ln f txt
```

Análisis

Aunque no es un programa muy práctico, hace uso de un enunciado `continue` en forma efectiva. Las líneas 9 y 10 declaran las variables del programa, `buffer[]` guarda la cadena que el usuario teclea en la línea 15. La otra variable, `ctr`, se incrementa por el buffer mientras el ciclo `for` de las líneas 20 a 23 busca vocales. Para cada letra en el ciclo, un enunciado `if` en las líneas 26 y 27 compara las letras contra las vocales minúsculas. Si alguna coincide, se ejecuta un enunciado `continue`, regresando el control a la línea 20, el enunciado `for`. Si la letra no es una vocal, el control pasa al enunciado `if` y se ejecuta la línea 32. La línea 32 contiene una nueva función de biblioteca, `putchar()`, que despliega un solo carácter en la pantalla.

Sintaxis

El enunciado *continue*

```
continue;
```

`continue` se usa en el interior de un ciclo. Hace que el control de programa se salte el resto de la iteración actual del ciclo y comience una nueva iteración.

Ejemplo

```
#include <stdio.h>
int x;
printf("Imprimiendo solamente los números pares de 1 a 10\n");
for( x = 1; x <= 10; x++ )
{
   if( x % 2 != 0 )  /* Ve si el número es NO par */
          continue;  /* Obtiene la nueva instancia de x */
   printf( "\n%d", x );
}
```

El enunciado *goto*

El enunciado `goto` es uno de los enunciados de *salto incondicional,* o *ramificación,* del C. Cuando la ejecución del programa llega a un enunciado `goto`, la ejecución salta inmediatamente, o se ramifica, a la posición especificada por el enunciado `goto`. El enunciado es *incondicional,* debido a que la ejecución siempre ramifica cuando encuentra un enunciado `goto`. La ramificación no depende de ninguna condición de programa (a diferencia de los enunciados `if`, por ejemplo).

La sintaxis del enunciado `goto` es

```
goto destino;
```

destino es un *enunciado de etiqueta,* que identifica la posición del programa a la que debe ramificar la ejecución. Un enunciado de etiqueta consiste en un identificador, seguido por dos puntos y un enunciado C.

```
lugar1: un enunciado C;
```

Si se quiere que la etiqueta esté sola en una línea, se le puede poner a continuación el enunciado nulo (un punto y coma solo).

```
lugar1: ;
```

Un enunciado goto y su etiqueta de destino deben encontrarse en la misma función, aunque pueden estar en diferentes bloques. Véase el listado 13.3, un programa simple que usa un enunciado goto.

Captura **Listado 13.3. Demostración del enunciado goto.**

```
1: /* Demuestra un enunciado goto */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int n;
8:
9: start: ;
10:
11:     puts("Enter a number between 0 and 10: ");
12:     scanf("%d", &n);
13:
14:     if ( n < 0 || n > 10 )
15:         goto start;
16:     else if (n == 0)
17:         goto location0;
18:     else if (n == 1)
19:         goto location1;
20:     else
21:         goto location2;
22:
23: location0: ;
24:     puts("You entered 0.");
25:     goto end;
26:
27: location1: ;
28:     puts("You entered 1.");
29:     goto end;
30:
31: location2: ;
32:     puts("You entered something between 2 and 10.");
```

Listado 13.3. continuación

```
33:
34: end: ;
35: }
```

Salida

```
>list1303
Enter a number between 0 and 10:
1
You entered 1.

>list1303
Enter a number between 0 and 10:
9
You entered something between 2 and 10.
```

Análisis

Este es un programa simple que acepta un número entre 0 y 10. Si el número no está entre 0 y 10, el programa usa un enunciado goto en la línea 15 para ir a `start`, que se encuentra en la línea 9. En caso contrario, el programa revisa en la línea 16 para ver si el número es igual a 0. En caso de serlo, un enunciado `goto` en la línea 17 manda el control a `location0` (línea 23), que imprime un enunciado en la línea 24 y ejecuta otro goto. El `goto` de la línea 25 manda el control a `end`, que se encuentra al final del programa. El programa ejecuta la misma lógica para el valor 1 y para todos los valores del 2 al 10.

El destino de un enunciado `goto` puede encontrarse antes o después de ese enunciado en el código. La única restricción, como se dijo anteriormente, es que tanto el `goto` como el destino deben estar en la misma función. Sin embargo, pueden estar en diferentes bloques. Se puede usar el `goto` para transferir la ejecución, tanto dentro como fuera de los ciclos, como un enunciado `for`. Sin embargo, nunca se debe hacer esto, y le recomendamos cuidadosamente que nunca use los enunciados `goto` en ningún lugar del programa. Hay dos razones.

- No son necesarios. No hay tarea de programación que requiera el enunciado goto. Siempre se puede escribir el código necesario usando los otros enunciados de ramificación del C.
- Es peligroso. Los enunciados goto pueden parecer una solución ideal para determinados problemas de programación, pero es fácil abusar. Cuando la ejecución del programa se ramifica con el enunciado goto no se guarda registro de dónde vino la ejecución, por lo que ésta puede dar brincos por todos lados en el programa. Este tipo de programación es conocida en el medio como *código espagueti.*

No obstante, algunos programadores cuidadosos pueden escribir programas muy buenos que usan `goto`. Puede haber situaciones donde el uso razonable del `goto` es la solución más simple a un problema de programación. Sin embargo, nunca es la única solución. Si va a ignorar esta precaución, ¡por lo menos tenga cuidado!

DEBE	NO DEBE

DEBE Evitar el uso de los enunciados goto tanto como sea posible.

NO DEBE Confundir break y continue. break cierra un ciclo, en tanto que continue inicia la siguiente iteración.

Ciclos infinitos

¿Qué es un ciclo infinito y por qué se querrá usar uno en un programa? Un ciclo infinito es aquel que, actuando solo, ejecutará para siempre. Puede ser un ciclo `for`, `while` o `do...while`. Por ejemplo, si se escribe

```
while (1)
{
   /* aquí va código adicional */
}
```

se crea un ciclo infinito. La condición que revisa el `while` es la constante 1, que siempre será cierta y no puede ser cambiada por la ejecución del programa. Por lo tanto, actuando solo, el ciclo nunca termina.

Sin embargo, en la sección anterior se vio que el enunciado `break` puede ser usado para salir de un ciclo. Sin el enunciado `break` un ciclo infinito no tendría utilidad. Con el `break` se pueden aprovechar los ciclos infinitos.

También se puede crear un ciclo `for` infinito o un ciclo `do...while` infinito de la manera siguiente:

```
for (;;)
{
   /* aquí va código adicional */
}
do
{
   /* aquí va código adicional */
} while (1);
```

El principio sigue siendo el mismo para los tres tipos de ciclo. Los ejemplos de esta sección usan el ciclo `while`.

Un ciclo infinito puede ser usado para revisar varias condiciones y determinar si el ciclo debe terminar. Tal vez sea difícil incluir todas la condiciones de prueba en paréntesis después del enunciado `while`. Quizá sea más fácil revisar las condiciones individualmente en el cuerpo del ciclo, y luego salir ejecutando un `break` cuando se necesita.

Un ciclo infinito también puede crear un sistema de menú que dirija las operaciones del programa. Tal vez recuerde del Día 5, "Funciones: lo básico", que la función `main()` del programa sirve a menudo como un tipo de "agente de tránsito", dirigiendo la ejecución entre las diversas funciones que hacen el trabajo real del programa. Esto a menudo se logra por un menú de algún tipo: se le presenta al usuario una lista de alternativas y él selecciona alguna. Una de las alternativas disponibles debe ser terminar el programa. Una vez que se ha hecho una selección, se usa alguno de los enunciados de decisión del C para dirigir la ejecución del programa según se necesite.

El programa del listado 13.4 muestra un sistema de menú.

Captura

Listado 13.4. Uso de un ciclo infinito para estructurar un sistema de menú.

```
1: /* Demuestra el uso de un ciclo infinito */
2: /* para estructurar un sistema de menú */
3: #include <stdio.h>
4: #define DELAY  150000       /* Usado en el ciclo de espera. */
5:
6: int menu(void);
7: void delay(void);
8:
9: main()
10: {
11:     int choice;
12:
13:     while (1)
14:     {
15:
16:     /* Obtiene la selección del usuario. */
17;
18:     choice = menu();
19:
20:     /* Ramifica en base a la entrada */
21:
22;     if (choice == 1)
23:     {
24:         puts("\nExecuting choice 1.");
25:         delay();
26:     }
27:     else if (choice == 2)
28:          {
29:              puts("\nExecuting choice 2.");
30:              delay();
31:          }
32;     else if (choice == 3)
33:          {
34:              puts("\nExecuting choice 3.");
```

```
35:             delay();
36:         }
37:     else if (choice == 4)
38:         {
39:             puts("\nExecuting choice 4.");
40:             delay();
41:         }
42:     else if (choice == 5)       /* Sale del programa. */
43:         {
44:             puts("Exiting program now...");
45:             delay();
46:             break;
47:         }
48:         else
49:         {
50:             puts("Invalid choice, try again.");
51:             delay();
52:         }
53:     }
54: }
55:
56: int menu(void)
57: /* Despliega un menú y recibe la selección del usuario. */
58: {
59:     int reply;
60:
61:     puts("\nEnter 1 for task A.");
62:     puts("Enter 2 for task B.");
63:     puts("Enter 3 for task C");
64:     puts("Enter 4 for task D.");
65:     puts("Enter 5 to exit program.");
66:
67:     scanf("%d", &reply);
68:
69:     return reply;
70: }
71:
72: void delay( void )
73: {
74:     long x;
75:     for ( x = 0; x < DELAY; x++ )
76:         ;
77: }
```

Salida

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
1
```

```
Executing choice 1.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

Análisis En el listado 13.4 es llamada una función llamada `menu()` en la línea 18, y definida en las líneas 56 a 70. `menu()` despliega un menú en la pantalla, acepta entrada del usuario y regresa la entrada al programa principal. En `main()` una serie de enunciados `if` anidados revisa el valor regresado y dirige la ejecución de acuerdo con él. La única cosa que hace el programa es desplegar mensajes en la pantalla. En un programa real el código podría llamar diversas funciones y ejecutar la tarea seleccionada.

Este programa también usa una segunda función, llamada `delay()`. `delay()` está definida en las líneas 72 a 77 y en realidad no hace mucho. Dicho simplemente, el enunciado `for` de la línea 75 efectúa ciclo haciendo nada (línea 76). El enunciado hace ciclo la cantidad de veces que indica `DELAY`. Este es un método efectivo para hacer que el programa haga una pausa. Si la pausa es demasiado corta o demasiado larga, se puede ajustar el valor de `DELAY`.

Tanto Borland como Zortech proporcionan una función similar a `delay()`, llamada `sleep()`. Esta función hace una pausa en la ejecución del programa, con una duración de la cantidad de segundos que se le pasan como argumento. Para usar a `sleep()` el programa debe incluir el archivo de encabezado TIME.H, en caso de estar usando el compilador Zortech. Se debe usar DOS.H si se usa el compilador Borland. Si no se está usando ninguno de estos compiladores, o un compilador que soporte a `sleep()`, en vez de ello se puede usar a `delay()`.

El enunciado *switch*

El enunciado de control de programa más flexible del C es el enunciado `switch`, que permite que el programa ejecute diferentes enunciados basado en una expresión que puede tener más de dos valores. Los enunciados de control anteriores, como `if`, se encuentran limitados a la evaluación de una expresión que solamente puede tener dos valores: cierto o falso. Para controlar el flujo del programa con base en más de dos valores se tienen que usar varios

enunciados `if` anidados, como lo muestra el listado 13.4. El enunciado `switch` hace innecesario este anidado.

La forma general del enunciado `switch` es la siguiente:

```
switch (expresion)
{
   case plantilla_1: enunciado(s);
   case plantilla_2: enunciado(s);
   ...
   case plantilla_n: enunciado(s);
   default: enunciado(s);
}
```

En este enunciado `expresion` es cualquier expresión que evalúa a un valor entero: tipo `long`, `int` o `char`. El enunciado `switch` evalúa la `expresión`, y compara el valor contra las plantillas que están a continuación de cada etiqueta `case`. Luego

- Si hay concordancia entre la *expresión* y alguna de las plantillas, la ejecución es transferida al enunciado que está a continuación de la etiqueta `case`.
- Si no hay concordancia, la ejecución es transferida al enunciado que está a continuación de la etiqueta opcional `default`.
- Si no hay concordancia ni etiqueta `default`, la ejecución pasa al primer enunciado que está a continuación de la llave de cierre del enunciado `switch`.

El enunciado `switch` es demostrado por el programa simple del listado 13.5, que despliega un mensaje con base en la entrada del usuario.

Listado 13.5. Demostración del enunciado `switch`.

```
1: /* Demuestra un enunciado switch. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int reply;
8:
9:     puts("Enter a number between 1 and 5:");
10:     scanf("%d", &reply);
11:
12:     switch (reply)
13:     {
14:         case 1:
15:             puts("You entered 1.");
16:         case 2:
17:             puts("You entered 2.");
```

Listado 13.5. continuación

```
18:         case 3:
19:             puts("You entered 3.");
20:         case 4:
21:             puts("You entered 4.");
22:         case 5:
23:             puts("You entered 5.");
24:         default:
25:             puts("Out of range, try again.");
26:     }
27: }
```

Salida

Compile y ejecute el programa, tecleando 2 cuando lo pida el programa. La salida es

```
Enter a number between 1 and 5:
2
You entered 2.
You entered 3.
You entered 4.
You entered 5.
Out of range, try again.
```

Análisis

Bueno, esto no es correcto, ¿o si? Parece que el enunciado `switch` encuentra la primera plantilla que concuerda y luego ejecuta todo lo que se encuentra a continuación (y no sólo los enunciados asociados con la plantilla). Esto es exactamente lo que sucede. Esta es la manera en que se supone que debe trabajar el `switch`. De hecho, efectúa un `goto` a la plantilla que concuerda. Para asegurarse de que sólo sean ejecutados los enunciados asociados con la plantilla que concuerda, incluya un enunciado `break` donde sea necesario. El listado 13.6 muestra al programa reescrito con enunciados `break`. Ahora funciona correctamente.

Captura

Listado 13.6. Uso correcto de `switch` incluyendo enunciados `break` donde se necesitan.

```
1: /* Demuestra un enunciado switch correctamente. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int reply;
8:
9:     puts("Enter a number between 1 and 5:");
10:    scanf("%d", &reply);
11:
12:    switch (reply)
13:    {
```

```
14:       case 0:
15:           break;
16        case 1:
17:           {
18:           puts("You entered 1.");
19:           break;
20:           }
21:       case 2:
22:           {
23:           puts("You entered 2.");
24:           break;
25:           }
26:       case 3:
27:           {
28:           puts("You entered 3.");
29:           break;
30:           }
31:       case 4:
32:           {
33:           puts("You entered 4.");
34:           break;
35:           }
36:       case 5:
37:           {
38:           puts("You entered 5.");
39:           break;
40:           }
41:       default:
42:           puts("Out of range, try again.");
43:           }
44: }
```

Salida

```
>list1306
Enter a number between 1 and 5:
1
You entered 1.

>list1306
Enter a number between 1 and 5:
6
Out of range, try again.
```

Análisis

Compile y ejecute esta versión; funciona correctamente.

Es común que con el enunciado `switch` se implemente el tipo de menús que se muestran en el listado 13.4. El uso de `switch` es mucho mejor que el uso de enunciados `if` anidados, como se empleó en la versión anterior.

El programa en el listado 13.7 usa switch, en vez de `if`.



Listado 13.7. Uso del enunciado `switch` para ejecutar un sistema de menú.

```
1: /* Demuestra el uso de un ciclo infinito */
2: /* y el enunciado switch para estructurar un sistema de menú. */
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: #define DELAY 150000
7:
8: int menu(void);
9: void delay(void);
10:
11: main()
12: {
13:
14:    while (1)
15:     {
16:     /* Obtiene la selección del usuario y ramifica con base en ella. */
17:
18:          switch(menu())
19:          {
20:             case 1:
21:                 {
22:                 puts("\nExecuting choice 1.");
23:                 delay();
24:                 break;
25:                 }
26:             case 2:
27:                 {
28:                 puts("\nExecuting choice 2.");
29:                 delay();
30:                 break;
31:                 }
32:             case 3:
33:                 {
34:                 puts("\nExecuting choice 3.");
35:                 delay();
36:                 break;
37:                 }
38:             case 4:
39:                 {
40:                 puts("\nExecuting choice 4.");
41:                 delay();
42:                 break;
43:                 }
44:             case 5:     /* Salida del programa */
45:                 {
46:                 puts("Exiting program now...");
```

```
47:                 delay();
48:                 exit(0);
49:                 }
50:             default:
51:                 {
52:                 puts("Invalid choice, try again.");
53:                 delay();
54:                 }
55:         }
56:     }
57: }
58:
59: int menu(void)
60:/* Despliega un menú y recibe la selección del usuario. */
61: {
62:     int reply;
63:
64:     puts("\nEnter 1 for task A.");
65:     puts("Enter 2 for task B.");
66:     puts("Enter 3 for task C.");
67:     puts("Enter 4 for task D.");
68:     puts("Enter 5 to exit program.");
69:
70:     scanf("%d", &reply);
71:
72:     return reply;
73: }
74:
75: void delay( void )
76: {
77:     long x;
78:     for( x = 0; x < DELAY; x++ )
79:         ;
80: }
```

Salida

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
1

Executing choice 1.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
```

```
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

Análisis Hay otro nuevo enunciado en esta versión: la función de biblioteca `exit()` en los enunciados asociados con `case` 5:, en la línea 48. Aquí no se puede usar `break`, como se hizo en la versión del listado 13.4. La ejecución de un `break` simplemente cortaría al enunciado `switch`, pero no al ciclo `while` infinito. Como aprenderá en la siguiente sección, la función `exit()` termina el programa.

Sin embargo, algunas veces puede ser útil hacer que la ejecución "pase por todas" las partes de una construcción `switch` puede ser útil. Digamos, por ejemplo, que se quiere que el mismo grupo de enunciados sea ejecutado si alguno de varios valores es encontrado. Simplemente omita los enunciados `break` y liste todas las plantillas `case` antes de los enunciados. Esto está ilustrado por el programa del listado 13.8.

Captura **Listado 13.8. Otra manera de usar el enunciado `switch`.**

```
1: /* Otro uso del enunciado switch. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: main()
7: {
8:     int reply;
9:
10:     while (1)
11:     {
12:         puts("Enter a value between 1 and 10, 0 to exit: ");
13:         scanf("%d", &reply);
14:
15:         switch (reply)
16:         {
17:             case 0:
18:                 exit(0);
19:             case 1:
20:             case 2:
21:             case 3:
22:             case 4:
23:             case 5:
24:                 {
25:                 puts("You entered 5 or below.");
26:                 break;
```

```
27:                 }
28:             case 6:
29:             case 7:
30:             case 8:
31:             case 9:
32:             case 10:
33:                 {
34:                 puts("You entered 6 or higher.");
35:                 break;
36:                 }
37:             default:
38:                 puts("Between 1 and 10, please!");
39:         }
40:     }
41: }
```

Salida

```
Enter a value between 1 and 10, 0 to exit:
11
Between 1 and 10, please!
Enter a value between 1 and 10, 0 to exit:
1
You entered 5 or below.
Enter a value between 1 and 10, 0 to exit:
6
You entered 6 or higher.
Enter a value between 1 and 10, 0 to exit:
0
```

Análisis

Este programa acepta un valor de teclado y luego determina si el valor es 5 o menor, o 6 o mayor, o no se encuentra entre 1 y 10. Si el valor es 0, la línea 18 ejecuta una llamada a la función `exit()` y, con ello, da por terminado el programa.

Sintaxis

El enunciado *switch*

```
switch (expresion)
{
   case plantilla_1: enunciado(s);
   case plantilla_2: enunciado(s);
   ...
   case plantilla_n: enunciado(s);
   default: enunciado(s);
}
```

El enunciado `switch` permite múltiples ramificaciones con una sola expresión. Es más eficiente y fácil de seguir que un enunciado `if` de varios niveles. Un enunciado `switch` evalúa una expresión, y luego ramifica al enunciado `case` que contiene la plantilla que concuerda con el resultado de la expresión. Si ninguna plantilla concuerda con el resultado

de la expresión, el control pasa al enunciado `default`. Si no hay enunciado `default`, el control pasa al final del enunciado `switch`.

El flujo del programa continúa a partir del enunciado `case`, a menos que se encuentre un enunciado `break`. Si se encuentra un enunciado `break`, el control pasa al final del enunciado `switch`.

Ejemplo 1

```
switch( letra )
{
   case 'A':
   case 'a':
            printf( "Se tecleó A" );
         break;
   case 'B':
   case 'b':
            printf( "Se tecleó B");
         break;
   ...
   ...
   default:
            printf( "No hay case para %c", letra );
}
```

Ejemplo 2

```
switch( cifra )
{
   case 0:    puts( "El número es 0 o menor.");
   case 1:    puts( "El número es 1 o menor.");
   case 2:    puts( "El número es 2 o menor.");
   case 3:    puts( "El número es 3 o menor.");
   ...
   ...
   case 99:   puts( "El número es 99 o menor.");
              break;
   default:   puts( "El número es mayor que 99.");
}
```

Debido a que no hay enunciados `break` para los enunciados del primer `case`, este ejemplo encuentra el `case` que concuerda con el número, e imprime cada `case` desde ese punto hasta el `break` que se encuentra en `case` 99. Si el número fuera 3, se podría decir que el número es igual a o menor que 3, igual o menor que 4, igual o menor que 5... igual o menor que 99. El programa continúa imprimiendo hasta que llega al enunciado `break` en `case` 99.

Terminación del programa

Un programa en C normalmente termina cuando la ejecución llega a la llave de cierre de la función `main()`. Sin embargo, se puede terminar el programa en cualquier momento llamando la función de biblioteca `exit()`. También se puede especificar una o más funciones que sean ejecutadas automáticamente a la terminación.

La función *exit()*

La función `exit()` termina la ejecución del programa y regresa control al sistema operativo. Esta función toma un solo argumento de tipo `int`, que es regresado al sistema operativo para indicar el éxito o la falla del programa. La sintaxis de la función `exit()` es

```
exit(estado);
```

Si `estado` tiene un valor de 0, indica que el programa ha terminado normalmente. Un valor de 1 indica que el programa ha terminado con algún tipo de error. Por lo general, el valor de retorno es ignorado. En un sistema DOS se puede probar el valor de retorno con un archivo por lotes del DOS y el enunciado `if errorlevel`. Sin embargo, este no es un libro acerca del DOS, por lo que necesitará consultar la documentación correspondiente, si quiere hacer uso del valor de retorno del programa.

Para usar la función `exit()` el programa debe incluir el archivo de encabezado STDLIB.H. Este archivo de encabezado también define dos constantes simbólicas, para usarse como argumentos de la función `exit()` de la manera siguiente:

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

Por lo tanto, para salir con un valor de retorno de `0` llame a `exit(EXIT_SUCCESS)`, y para un valor de retorno de `1` llame a `exit(EXIT_FAILURE)`.

DEBE	NO DEBE

NO DEBE Olvidar usar enunciados `break` si el enunciado `switch` los necesita.

DEBE Usar un caso `default` en un enunciado `switch`, aunque piense que ha cubierto todos los casos posibles.

DEBE Usar un enunciado `switch`, en vez de un enunciado `if`, si hay más de dos condiciones que están siendo evaluadas para la misma variable.

DEBE Alinear los enunciados `case` para que sean fáciles de leer.

La función *atexit()* (sólo para el DOS)

La función `atexit()` se usa para especificar, o registrar, una o más funciones que se ejecutan automáticamente cuando el programa termina. Esta función tal vez no esté disponible en sistemas que no sean el DOS. Se puede registrar un máximo de 32 funciones, y al momento de terminación se ejecutan en orden inverso. La última función registrada es la primera ejecutada. Cuando han sido ejecutadas todas las funciones registradas por `atexit()` el programa termina y el control regresa al sistema operativo.

El prototipo de la función `atexit()` se encuentra en el archivo de encabezado STDLIB.H, que debe ser incluido en cualquier programa que use a `atexit()`. El prototipo dice lo siguiente:

```
int atexit(void (*)(void));
```

Tal vez no reconozca este formato. Significa que `atexit()` toma un apuntador a función como su argumento. (En el Día 15, "Más sobre apuntadores", se tratan los apuntadores a función con mayor precisión.) Las funciones registradas con `atexit()` deben tener un tipo de retorno de `void`. Digamos que se quiere que las funciones `limpieza1()` y `limpieza2()` sean ejecutadas en ese orden a la terminación. Así es como lo haría:

```
1: #include <stdlib.h>
2:
3: void limpieza1(void);
4: void limpieza2(void);
5:
6: main()
7: {
8:    atexit(limpieza2);
9:    atexit(limpieza1);
10:
11:
12: }   /* fin de main() */
13:
14:
15: void limpieza1(void)
16: {
17: /* aquí va código adicional */
18: }
19:
20: void limpieza2(void)
21: {
22:    /* aquí va código adicional */
23: }
```

Cuando este programa ejecuta, la línea 8 registra a `limpieza2()` con la función `atexit()` y la línea 9 registra a `limpieza1()`. Cuando el programa termina, las funciones registradas son llamadas en orden inverso: primero es llamada `limpieza1()` y a continuación `limpieza2()`, y luego el programa termina. Las funciones registradas con `atexit()` se

ejecutan cuando la terminación del programa es causada por la ejecución de la función `exit()` o cuando se llega al final de `main()`.

El programa del listado 13.9 muestra la manera en que se usan las funciones `exit()` y `atexit()`.

Captura Listado 13.9. Uso de las funciones `exit()` y `atexit()`.

```
1: /* Demuestra las funciones exit() y atexit(). */
2: #include <stdio.h>
3: #include <stdlib.h>
4: #include <time.h>
5: #define DELAY 150000
6:
7: void cleanup( void );
8: void delay( void );
9:
10: main()
11: {
12:     int reply;
13:
14:    /* Registra la función que será llamada a la salida. */
15:
16:     atexit( cleanup );
17:
18:     puts("Enter 1 to exit, any other to continue.");
19:     scanf("%d", &reply);
20:
21:     if (reply == 1)
22:         exit( EXIT_SUCCESS );
23:
24:    /* Pretende hacer algún trabajo. */
25:
26:     for (reply = 0; reply < 5; reply++)
27:     {
28:         puts("Working...");
29:         delay();
30:     }
31: }       /* Fin de main() */
32:
33: void cleanup( void )
34: {
35:     puts("\nPreparing for exit...");
36:     delay();
37: }
38:
39: void delay(void)
40: {
41:     long x;
```

Listado 13.9. continuación

```
42:     for( x = 0; x < DELAY; x++ )
43:         ;
44: }
```

Salida

```
>list1309
Enter 1 to exit, any other to continue.
1

Preparing for exit...

>list1309
Enter 1 to exit, any other to continue.
8
Working...
Working...
Working...
Working...
Working...

Preparing for exit...
```

Análisis

Este programa registra la función `cleanup()`, en la línea 16, con la función `atexit()`. Cuando el programa termina, ya sea con la función `exit()` en la línea 22 o por llegar al final de `main()`, ejecuta `cleanup()`.

¿Por qué habría de necesitar una función de "limpieza" a la terminación del programa? Ciertamente el ejemplo en el listado 13.9 no hace nada útil, ya que sirve solamente para mostrar la función. Sin embargo, en programas reales a veces hay tareas que necesitan ser ejecutadas a la terminación: por ejemplo, el cierre de archivos y flujos, o la liberación de memoria asignada dinámicamente. (Los archivos y flujos se tratan en los Días 14 y 16.) Tal vez en este momento estas cosas no signifiquen nada para usted, pero se tratan posteriormente en el libro. Entonces podrá ver la manera de usar `atexit()` en una situación real.

DEBE	NO DEBE
NO DEBE Abusar de la función `atexit()`. Limite su uso a aquellas funciones que deban ser ejecutadas en caso de que haya un problema cuando el programa termina, como el cierre de un archivo abierto.	
DEBE Usar el comando `exit()` para salir del programa en caso de que haya algún problema.	

Ejecución de comandos del sistema operativo en un programa

La biblioteca estándar del C incluye una función, `system()`, que le permite ejecutar comandos del sistema operativo dentro de un programa C en ejecución. Esto puede ser útil, permitiéndole leer el listado de un directorio del disco o formatear un disco sin salir del programa. Para usar la función `system()`, un programa debe incluir el archivo de encabezado STDLIB.H. El formato de `system()` es

```
system(comando);
```

El argumento `comando` puede ser una constante de cadena o un apuntador a una cadena. Por ejemplo, para obtener un listado de directorio en el DOS, se podría escribir

```
system("dir");
```

o

```
char *comando = "dir";
system(comando);
```

Después de que el comando del sistema operativo se ejecuta, la ejecución regresa al programa en la posición inmediatamente a continuación de la llamada a `system()`. Si el comando que se pasa a `system()` no es un comando válido del sistema operativo, se obtiene un mensaje de error `Bad command or file name` (Comando o nombre de archivo incorrecto) antes de regresar al programa. El uso de `system()` está ilustrado en el listado 13.10.



Listado 13.10. Uso de la función `system()` para ejecutar comandos de sistema.

```
1: /* Demuestra la función system(). */
2:
3: #include <stdlib.h>
4:
5: main()
6: {
7:     /* Declara un buffer para guardar la entrada. */
8:
9:     char input[40];
10:
11:     while (1)
12:     {
13:         /* Obtiene el comando del usuario. */
14:
15:         puts("\nInput the desired DOS command, blank to exit");
16:         gets(input);
```

Listado 13.10. continuación

```
17:
18:         /* Sale si se teclea una línea en blanco */
19:
20:         if (input[0] == '\0')
21:             exit(0);
22:
23:         /* Ejecuta el comando. */
24:
25:         system(input);
26:     }
27: }
```

Salida

```
Input the desired DOS command, blank to exit
dir *.bak

Volume in drive E is STACVOL_000
Directory of E:\BOOK\LISTINGS
LIST1414 BAK      1416 08-22-92   5:18p
        1 file(s)       1416 bytes
                    24068096 bytes free

Input the desired DOS command, blank to exit
```

Análisis

El listado 13.10 ilustra el uso de `system()`. Con un ciclo `while`, en las líneas 11 a 26, este programa permite los comandos del sistema operativo. Las líneas 15 y 16 le piden al usuario que teclee el comando del sistema operativo. Si se dio Enter sin teclear ningún comando, las líneas 20 y 21 llaman a `exit()` para que termine el programa. La línea 25 llama a `system()`, con el comando tecleado por el usuario.

Los comandos que se pasan a `system()` no están limitados a comandos simples del sistema operativo, como el listado de directorios o el formateo de discos. También se puede pasar el nombre de cualquier archivo ejecutable o por lotes, claro, y el programa se ejecuta normalmente. Por ejemplo, si se pasa el argumento LIST1309, se podría ejecutar el programa llamado LIST1309. Cuando termina el programa, la ejecución regresa a donde fue hecha la llamada a `system()`.

La única restricción sobre el uso de `system()` se refiere a la memoria. Cuando `system()` se ejecuta, el programa original permanece cargado en la RAM de la computadora, y se carga también una nueva copia del procesador de comandos del sistema operativo y cualquier programa que se ejecute. Esto funciona solamente si la computadora tiene suficiente memoria. En caso de no ser suficiente, se obtiene un mensaje de error.

Resumen

Este capítulo trató varios temas relacionados con el control de programa. Se aprendió acerca del enunciado `goto` y el porqué debe evitarse su uso en los programas. Se vio que los enunciados `break` y `continue` dan control adicional sobre la ejecución de ciclos, y que estos enunciados pueden usarse en conjunción con ciclos infinitos para ejecutar tareas de programación útiles.

Este capítulo también explicó la manera de usar la función `exit()` para controlar la terminación del programa, y la manera en que puede ser usada `atexit()` para registrar funciones que serán ejecutadas automáticamente cuando termine el programa. Por último se vio la manera de usar la función `system()`, para ejecutar comandos del sistema desde dentro del programa.

Preguntas y respuestas

1. ¿Es mejor usar un enunciado `switch` que un ciclo anidado?

   Si se está revisando una variable, por lo general un enunciado `switch` es más eficiente. Muchas veces también es más fácil de leer. Como regla general, si sólo hay dos opciones, use un enunciado `if`. Si hay más de dos, un enunciado `switch` es más eficiente.

2. ¿Por qué debo evitar el enunciado `goto`?

   Cuando se ve por primera vez un enunciado `goto` es fácil pensar que puede ser útil. Sin embargo, `goto` puede causar más problemas que los que resuelve. Un enunciado `goto` es un comando no estructurado que le lleva a otro punto en un programa. Muchos depuradores (software que ayuda a trazar problemas de los programas) no pueden interrogar adecuadamente al `goto`. Los enunciados `goto` también lo llevan al código espagueti (código que está desperdigado por todos lados).

3. ¿Por qué no todos los compiladores tienen las mismas funciones?

   En este capítulo debe haberse dado cuenta de que no se tienen disponibles todas las funciones de todos los compiladores o para todos los sistemas de computadoras. La función `atexit()` se encuentra disponible solamente en los sistemas DOS. La función `sleep()` es proporcionada por Borland y Zortech, pero no por Microsoft. Cada compilador es diferente, pero la mayoría tienen la misma funcionalidad y características generales. Cuando las funciones no están disponibles, por lo general es posible crearlas por uno mismo.

   La respuesta a la pregunta se basa en lo que cada compañía productora del compilador considera que es importante. Esta misma respuesta puede ser aplicada

a los procesadores de palabras o a las hojas de cálculo. No todos los procesadores de palabras u hojas de cálculo tienen la misma funcionalidad que los demás paquetes o funciones similares.

4. ¿Es conveniente usar la función `system()` para ejecutar funciones del sistema?

   La función `system()` pudiera parecer que es una manera fácil de hacer cosas como listar los archivos de un directorio. Sin embargo, se debe tener precaución. La mayoría de los comandos del sistema operativo son específicos de un sistema operativo en particular. Si se usa una llamada a `system()`, probablemente el código no sea portable. Si se quiere ejecutar otro programa (y no un comando del sistema operativo), no se deben tener problemas de portabilidad.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Cuándo es aconsejable usar el enunciado goto en los programas?
2. ¿Cuál es la diferencia entre el enunciado break y el continue?
3. ¿Qué es un ciclo infinito y cómo se hace uno?
4. ¿Cuáles dos eventos hacen que termine la ejecución del programa?
5. ¿Qué tipos de variable puede evaluar `switch`?
6. ¿Qué hace el enunciado `default`?
7. ¿Qué hace la función `atexit()`?
8. ¿Qué hace la función `system()`?

Ejercicios

1. Escriba un enunciado que haga que el control de programa vaya a la siguiente iteración de un ciclo.
2. Escriba el (los) enunciado(s) que envíen el control de un programa al final del ciclo.
3. Escriba código que haga que las funciones `f1()`, `f2()` y `f3()` ejecuten, en ese orden, cuando termine el programa.

4. Escriba una línea de código que despliegue un listado de todos los archivos del directorio actual.

5. BUSQUEDA DE ERRORES: ¿hay error en el siguiente código? Si lo hubiera, ¿cuál es?

```
switch( answer )
{
    case 'Y': printf("You answered yes");
              break;
    case 'N': printf( "You answered no");
}
```

6. BUSQUEDA DE ERRORES: ¿hay error en el siguiente código? Si lo hubiera, ¿cuál es?

```
switch( choice )
{
    default:
          printf("You did not choose 1 or 2");
    case 1:
          printf("You answered 1");
          break;
    case 2:
          printf( "You answered 2");
          break;
}
```

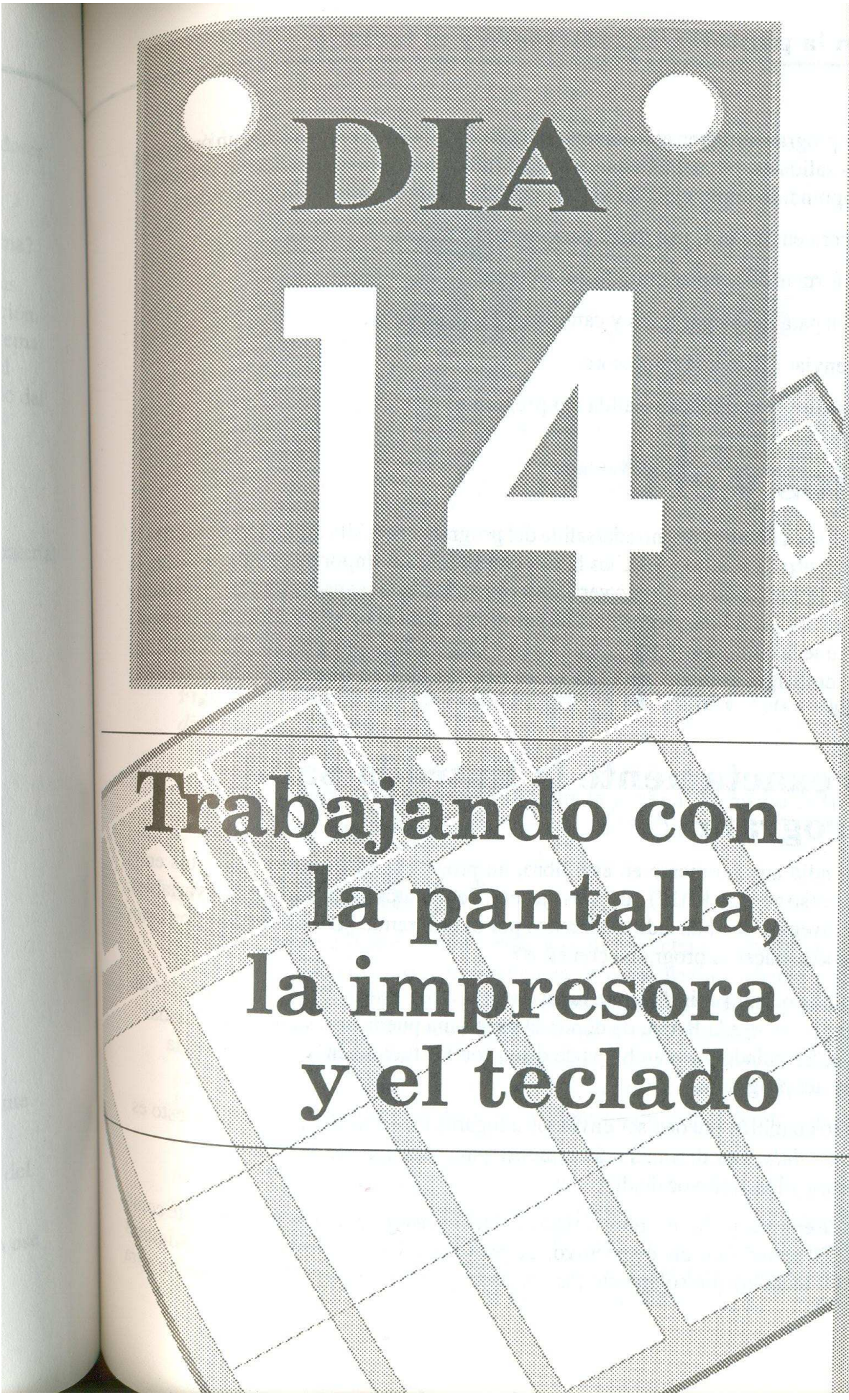
7. Reescriba el ejercicio 6 con enunciados `if`.

8. Escriba un ciclo `do...while` infinito.

Debido a la multitud de posibles respuestas a los siguientes dos ejercicios no se proporcionan respuestas.

9. Escriba un programa que funcione como una calculadora. El programa debe permitir suma, resta, multiplicación y división.

10. Escriba un programa que proporcione un menú con cinco opciones diferentes. La quinta opción debe terminar el programa. Cada una de las demás opciones debe ejecutar un comando de sistema usando la función `system()`.



DIA 14

Trabajando con la pantalla, la impresora y el teclado

Casi todos los programas deben ejecutar entrada y salida. Qué tan bien maneje un programa la entrada y la salida es, frecuentemente, el mejor indicador de la utilidad de un programa. Usted ya ha aprendido cómo ejecutar alguna entrada y salida básica. Hoy aprenderá

- La manera en que el C usa *flujos* para entrada y salida.
- Varias formas de aceptar entrada del teclado.
- Métodos para desplegar texto y cantidades en la pantalla.
- Cómo enviar salida a la impresora.
- Cómo redirigir la entrada y salida del programa.

Los flujos y el C

Antes de llegar a los detalles de entrada/salida del programa necesita aprender acerca de los *flujos*. Toda la entrada y salida del C es hecha con flujos, sin importar de dónde venga la entrada o a dónde vaya la salida. Como verá posteriormente, esta manera estándar de manejar todas las entradas y las salidas tiene ventajas definitivas para el programador. Por supuesto que esto hace que la comprensión de lo que son los flujos y la manera en que funcionan sea esencial. Sin embargo, primero necesita saber exactamente lo que significa el término *entrada/salida.*

¿Qué es exactamente la Entrada/Salida de un programa?

Como se aprendió anteriormente en este libro, un programa en C guarda los datos en la memoria de acceso al azar (RAM) mientras ejecuta. Estos datos están en forma de variables, estructuras y arreglos que han sido declarados por el programa. ¿De dónde vinieron estos datos y qué puede hacer el programa con ellos?

- Los datos pueden venir de algún lugar externo al programa. Los datos movidos de un lugar externo a la RAM, de donde el programa puede accesarlos, es llamada *entrada.* El teclado y los archivos de disco son las fuentes más comunes de la entrada del programa.
- Los datos también pueden ser enviados a lugares externos del programa, y esto es llamado *salida.* Los destinos más comunes para la salida son la pantalla, la impresora y los archivos de disco.

A las fuentes de entrada y a los destinos de salida se les hace referencia colectivamente como *dispositivos.* El teclado es un dispositivo, la pantalla es un dispositivo, etc. Algunos dispositivos (el teclado) son solamente para entrada, y otros (la pantalla), solamente para

salida. Otros más (archivos de disco) son tanto para entrada como para salida. Esto está ilustrado en la figura 14.1.

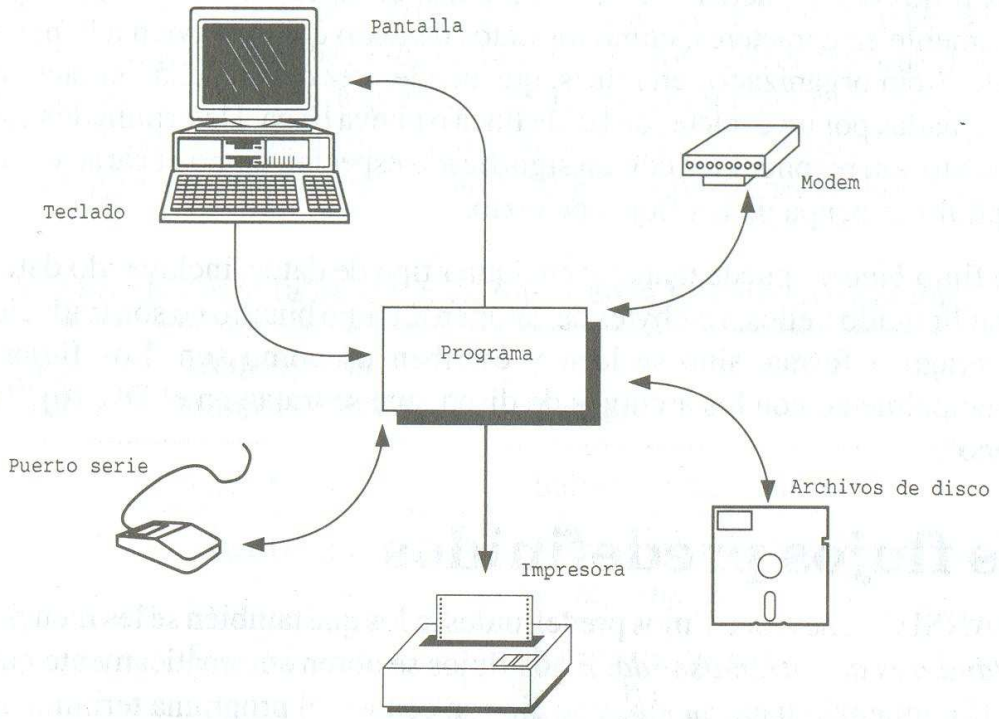


Figura 14.1. *La entrada y la salida pueden efectuarse entre el programa y una variedad de dispositivos externos.*

El C ejecuta todas las operaciones de entrada y salida por medio de flujos, sin importar el dispositivo, y sin importar si es de entrada o salida.

¿Qué es un flujo?

Un *flujo* es sencillamente una secuencia de caracteres. Con mayor exactitud, es una secuencia de bytes de datos. Una secuencia de bytes que fluyen hacia un programa es un flujo de entrada, y una secuencia de bytes que salen de un programa es un flujo de salida. La ventaja principal de los flujos es que la programación de entrada/salida es *independiente del dispositivo*. Los programadores no necesitan escribir funciones especiales de entrada/salida para cada dispositivo (teclado, disco, etc.). El programa "ve" a la entrada/salida como un flujo continuo de bytes, sin importar de dónde vienen ni a dónde van.

Cada flujo del C está conectado a un archivo. En este contexto, el término *archivo* no se refiere a un archivo de disco. En vez de ello, es un paso intermedio entre el flujo con el que trata el programa y el dispositivo físico actual que está siendo utilizado para la entrada o salida. En su mayor parte, el programador principiante no necesita preocuparse de estos archivos, ya que los detalles de la interacción entre flujos, archivos y dispositivos son manejados automáticamente por la biblioteca de funciones del C y el sistema operativo.

Flujos de texto contra flujos binarios

Los flujos del C pueden ser de dos modos: de texto y binarios. Un flujo de *texto* consiste solamente en caracteres, como los datos de texto que se envían a la pantalla. Los flujos de texto están organizados en líneas, que pueden ser de hasta 255 caracteres de largo y están terminadas por un carácter de fin de línea o nueva línea. Determinados caracteres en el flujo de texto son reconocidos con un significado especial, como el carácter de nueva línea. Este capítulo se ocupa de los flujos de texto.

Un flujo *binario* puede manejar cualquier tipo de datos, incluyendo datos de texto pero sin estar limitado a ellos. Los bytes de datos en un flujo binario no son traducidos o interpretados en ninguna forma, sino se leen y escriben tal como son. Los flujos binarios se usan principalmente con los archivos de disco, que se tratan en el Día 16, "Uso de archivos de disco".

Los flujos predefinidos

El ANSI C tiene cinco flujos predefinidos, a los que también se les menciona como *archivos estándares de entrada/salida.* Estos flujos se abren automáticamente cuando un programa en C comienza su ejecución, y se cierran cuando el programa termina. El programador no necesita hacer ninguna acción especial para que estos flujos estén a su disposición. Los flujos estándares, y los dispositivos a los que están conectados, están listados en la tabla 14.1. Los cinco flujos estándares son flujos de modo texto.

Tabla 14.1. Los cinco flujos estándares.

Nombre	Flujo	Dispositivo
`stdin`	Entrada estándar	Teclado
`stdout`	Salida estándar	Pantalla
`stderr`	Error estándar	Pantalla
`stdprn`	Impresora estándar	Impresora (LPT1:)
`stdaux`	Auxiliar estándar	Puerto serie (COM1:)

Cada vez que ha usado las funciones `printf()` o `puts()` para desplegar texto en la pantalla, ha usado el flujo `stdout`. De la misma forma, cuando usa `gets()` o `scanf()` para leer la entrada del teclado, se usa el flujo `stdin`. Los flujos estándares se abren automáticamente, pero otros flujos, como los usados para manejar información guardada en disco, deben abrirse explícitamente. Se aprenderá la manera de hacer esto en el Día 16, "Uso de archivos de disco". El resto de este capítulo se ocupa de los flujos estándares.

Funciones de flujo del C

La biblioteca estándar del C tiene una variedad de funciones que se ocupan de la entrada y salida de flujos. La mayoría de estas funciones vienen en dos variedades: una que siempre usa uno de los flujos estándares y otra que requiere que el programador especifique el flujo. Estas funciones se listan en la tabla 14.2. Esta tabla no lista todas las funciones de entrada/salida del C, y tampoco se tratan en este capítulo todas las funciones de la tabla.

Tabla 14.2. Las funciones de entrada/salida para flujos de la biblioteca estándar.

Usa uno de los flujos estándares	Requiere un nombre de flujo	Acción
`printf()`	`fprintf()`	Salida formateada
`vprintf()`	`vfprintf()`	Salida formateada con lista variable de argumentos
`puts()`	`fputs()`	Salida de cadena
`putchar()`	`putc(), fputc()`	Salida de carácter
`scanf()`	`fscanf()`	Entrada formateada
`gets()`	`fgets()`	Entrada de cadena
`getchar()`	`getc(), fgetc()`	Entrada de carácter
`perror()`		Solamente para salida de cadena a stderr

La función `perror()` puede requerir a STDLIB.H. Todas las otras funciones requieren a STDIO.H. `vprintf()` y `vfprintf()` también requieren a STDARGS.H. Unas cuantas otras funciones requieren a CONIO.H.

Un ejemplo

El programa corto del listado 14.1 demuestra la equivalencia de flujos. En la línea 10 la función `gets()` se usa para recibir una línea de texto del teclado `(stdin)`. Debido a que `gets()` regresa un apuntador a la cadena, puede ser usada como argumento para `puts()`, que despliega la cadena en la pantalla `(stdout)`. Cuando el programa ejecuta, recibe una línea de texto del usuario y luego despliega inmediatamente la cadena en la pantalla.

Captura **Listado 14.1. La equivalencia de flujos.**

```
1: /* Demuestra la equivalencia de flujos de entrada y salida. */
2: #include <stdio.h>
3:
4: main()
5: {
6:     char buffer[81];;
7:
8:     /* Recibe una línea, y luego inmediatamente le da salida. */
9:
10:     puts(gets(buffer));
11: }
```

DEBE	NO DEBE
DEBE Aprovechar los cinco flujos estándares de entrada/salida que proporciona el C.	
	NO DEBE Renombrar o cambiar los cinco flujos estándares innecesariamente.
	NO DEBE Tratar de usar un flujo de entrada, como stdin, para una función de salida, como fprintf().

Aceptando entrada del teclado

Casi todos los programas del C requieren alguna forma de entrada del teclado (esto es, desde `stdin`). Las funciones de entrada están divididas en una jerarquía de tres niveles: entrada de caracteres, entrada de líneas y entrada formateada.

Entrada de caracteres

Las funciones para entrada de caracteres leen la entrada de un flujo, carácter por carácter. Cuando son llamadas, cada una de estas funciones regresa el siguiente carácter del flujo, o `EOF`, si se ha llegado al fin de archivo o si ha sucedido algún error. `EOF` es una constante simbólica definida en stdio.h como -1. Las funciones de entrada de caracteres difieren por su forma de almacenado (bufferign) y de replicación (echoing).

- Algunas funciones de entrada de caracteres se *almacenan temporalmente (buffered)*. Esto significa que el sistema operativo guarda todos los caracteres en un espacio de almacenamiento temporal hasta que se oprime Enter, y luego el sistema envía los caracteres al flujo stdin. Otras *no se almacenan temporalmente*, y cada carácter es enviado a `stdin` tan pronto como se oprime la tecla.
- Algunas funciones de entrada replican automáticamente cada carácter a `stdout` en cuanto lo reciben. Otras no replican, sino que el carácter es enviado a `stdin` y no a `stdout`. Debido a que `stdout` es asignado a la pantalla, ahí es donde es replicada la entrada.

Los usos de la entrada de carácter con y sin almacenamiento temporal, replicada y sin replicar, se explican en las siguientes secciones.

La función *getchar()*

La función `getchar()` obtiene el siguiente carácter del flujo `stdin`. Proporciona entrada de caracteres con almacenamiento temporal con réplica, y su prototipo es

```
int getchar(void);
```

El uso de `getchar()` es demostrado en el listado 14.2. Observe que la función `putchar()`, explicada minuciosamente más adelante, en este capítulo, simplemente despliega un solo carácter en la pantalla.

Captura

Listado 14.2. Demostración de la función `getchar()`.

```
1: /* Demuestra la función getchar(). */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int ch;
8:
9:     while ((ch = getchar()) != '\n')
10:         putchar(ch);
11: }
```

Análisis

Esto es lo que pasa cuando ejecuta este programa:

1. En la línea 9 es llamada la función `getchar()`, y espera para recibir un carácter de stdin. Debido a que `getchar()` es una función de entrada con almacenamiento temporal, no se recibe ningún carácter en tanto no se oprima Enter. Sin embargo, cada tecla que se oprime es replicada en la pantalla.

2. Cuando se oprime Enter, todos los caracteres tecleados, incluida la nueva línea, son enviados a `stdin` por el sistema operativo. La función `getchar()` regresa los caracteres de uno en uno, asignando cada uno de ellos en turno a `ch`.
3. Cada carácter es comparado con el carácter de nueva línea, '\n' y, en caso de no ser igual, es deplegado en la pantalla con `putchar()`. Cuando `getchar()` regresa una nueva línea, termina el ciclo `while`.

La función `getchar()` puede ser usada para recibir líneas completas de texto, como se muestra en el listado 14.3. Sin embargo, otras funciones de entrada son más adecuadas para esta tarea, conforme aprenderá posteriormente, en este capítulo.

Captura

Listado 14.3. Uso de la función `getchar()` para recibir una línea de texto completa.

```
1: /* Uso de getchar() para recibir cadenas. */
2:
3: #include <stdio.h>
4:
5: #define MAX 80
6:
7: main()
8: {
9:     char ch, buffer[MAX+1];
10:     int x = 0;
11:
12:     while ((ch = getchar()) != '\n' && x < MAX)
13:         buffer[x++] = ch;
14:
15:     buffer[x] = '\0';
16:
17:     printf("%s", buffer);
18: }
```

Análisis

Este programa es similar al listado 14.2 en la forma en que usa `getchar()`. Se ha añadido una condición adicional al ciclo. Esta vez el ciclo `while` acepta caracteres de `getchar()` hasta que llegue un carácter de nueva línea o se lean 80 caracteres. Cada carácter es asignado a un arreglo, llamado `buffer`. Cuando se da entrada a los caracteres, la línea 15 pone un carácter nulo al final del arreglo, para que la función `printf()` en la línea 17 pueda imprimir la cadena completa.

Una nota adicional sobre este programa. En la línea 9, ¿por qué fue declarado `buffer` con un tamaño de `MAX + 1` en vez de simplemente `MAX`? Al declarar buffer con el tamaño de `MAX + 1` la cadena puede ser de 80 caracteres más el carácter nulo terminal. No olvide incluir un lugar para el carácter nulo terminal al final de las cadenas.

La función *getch()*

La función `getch()` obtiene el siguiente carácter del flujo `stdin`. Proporciona entrada de caracteres sin almacenamiento temporal y sin réplica. El prototipo para `getch()` está en el archivo de encabezado conio.h, de la manera siguiente:

```
int getch(void);
```

Como es sin almacenamiento temporal, `getch()` regresa cada carácter tan pronto se oprime una tecla, sin esperar a que el usuario oprima Enter. Debido a que `getch()` no replica su entrada, los caracteres no son desplegados en la pantalla. El uso de `getch()` se ilustra con el programa del listado 14.4.

Captura

Listado 14.4. Uso de la función `getch()`.

```
1: /* Demuestra la función getch(). */
2:
3: #include <stdio.h>
4: #include <conio.h>
5:
6: main()
7: {
8:     int ch;
9:
10:     while ((ch = getch()) != '\r')
11:         putchar(ch);
12:}
```

Análisis

Cuando este programa ejecuta, `getch()` regresa cada carácter tan pronto se oprime una tecla, y no espera a que se oprima Enter. No hay réplica, por lo que la única razón por la que cada carácter es desplegado en la pantalla es la llamada a `putchar()`. Pero, ¿por qué el programa compara cada carácter con `\r` en vez de `\n`?

El código `\r` es la secuencia de escape para el carácter de retorno. Cuando se oprime Enter, el dispositivo de teclado envía un retorno (CR) a `stdin`. Las funciones de entrada de caracteres con almacenamiento temporal traducen automáticamente el retorno a una nueva línea y, por lo tanto, el programa debe revisar por `\n` para determinar si se ha oprimido Enter. Las funciones de entrada de caracteres sin almacenamiento temporal no traducen, por lo que un retorno es recibido como `\r`, y esto es lo que debe ver el programa.

El uso de `getch()` para recibir una línea completa de texto se ilustra en el listado 14.5. La ejecución de este programa muestra claramente que `getch()` no replica su entrada. A excepción de la sustitución de `getchar()` por `getch()`, este programa es idéntico al del listado 14.3.

Listado 14.5. Uso de la función `getch()` para recibir una línea completa.

```
1: /* Uso de getch() para recibir cadenas. */
2:
3: #include <stdio.h>
4: #include <conio.h>
5:
6: #define MAX 80
7:
8: main()
9: {
10:     char ch, buffer[MAX+1];
11:     int x = 0;
12:
13:     while ((ch = getch()) != '\r' && x < MAX)
14:         buffer[x++] = ch;
15:
16:     buffer[x] = '\0';
17:
18:     printf("%s", buffer);
19:}
```

Precaución: `getch()` no es un comando estándar ANSI. Esto significa que su compilador (u otros compiladores) pueden aceptarlo o no. `getch()` es aceptado por Zortech y Borland. Microsoft acepta `_getch()`. Si se tienen problemas al usarlo, se deberá revisar el compilador y ver si acepta a `getch()`. Si se trata de tener portabilidad, se deben evitar las funciones que no son de ANSI.

La función *getche()*

Esta es una sección corta, debido a que `getche()` es exactamente igual a `getch()`, a excepción de que replica cada carácter a `stdout`. Modifique el programa del listado 14.4 para usar `getche()`, en vez de `getch()`. Cuando el programa ejecuta, cada tecla que se oprime es desplegada dos veces en la pantalla: una por la replicación de `getche()` y otra por `putchar()`.

Precaución: `getche()` no es un comando estándar de ANSI.

Las funciones *getc()* y *fgetc()*

Estas dos funciones de entrada de caracteres no trabajan automáticamente con `stdin`. En vez de ello le permiten al programa especificar el flujo de entrada. Se usan, en primer lugar, para leer caracteres de archivos de disco, lo que el Día 16, "Uso de archivos de disco", trata minuciosamente.

DEBE	NO DEBE
DEBE Entender la diferencia entre entrada replicada y sin replicar.	
DEBE Entender la diferencia entre entrada con almacenamiento temporal y sin almacenamiento temporal.	
	NO DEBE Usar funciones estándares no ANSI si desea tener portabilidad.

Obtención de teclas especiales en una PC

Hasta ahora ha aprendido la manera de recibir todos los caracteres regulares del teclado de la PC de IBM: letras, números y signos de puntuación. El teclado de la PC también tiene varias teclas especiales, como las teclas de función F1 a F10, así como las flechas y otras teclas de dirección en el teclado numérico. También se pueden dar combinaciones de teclas, como Ctrl y PgDn, Alt y 1, y Shift y F1. ¿Cómo hacer para aceptar entrada de estas teclas especiales en los programas de C?

Para hacerlo primero necesita saber la manera en que funcionan estas teclas y combinaciones de teclas. Estas teclas difieren de las teclas de caracteres regulares en que envían un par de valores a `stdin`, en vez de solamente uno. En cada par, el primer valor es siempre el carácter nulo, con un valor numérico de 0. El segundo tiene un valor numérico que identifica la tecla oprimida. Por ejemplo, al oprimir F1 se envía un par que consiste en 0 seguido por 59, y al apretar la tecla Home se envía un par que consiste en 0 seguido por 71. Las teclas que regresan un código de dos caracteres a `stdin` son llamadas las *teclas extendidas,* y sus códigos son llamados *códigos de teclas extendidas.*

¿Cómo maneja un programa la entrada de teclas extendidas? Específicamente, ¿cómo puede un programa aceptar solamente la opresión de una tecla especial, ignorando todos los demás caracteres? Es simple.

1. Acepta caracteres de `stdin`, descartándolos hasta que se recibe un 0 o \0. (Para el compilador C el número 0 y el carácter \0 son lo mismo.) Esto señala el primero de dos valores enviado por alguna de estas teclas.
2. Ve el siguiente valor, que identifica la tecla que fue oprimida.

Una función que ejecuta esta tarea, `ext_key()`, se presenta en el listado 14.6. El programa usa `ext_key()` para aceptar opresiones de teclas extendidas, ignorando todas las demás

entradas. El segundo valor de la tecla es desplegado en la pantalla, y al oprimir F1 se termina el programa. El segundo valor enviado por la tecla F1 es 59, por lo que este es el valor por el que revisa el programa en su enunciado `if`.

Captura

Listado 14.6. Aceptación de la entrada de teclas extendidas.

```
1: /* Demuestra la lectura de teclas extendidas del teclado. */
2:
3: #include <stdio.h>
4: #include <conio.h>
5:
6: int ext_key(void);
7:
8: main()
9: {
10:     int ch;
11:
12:     puts("Press any extended key; press F1 to exit.");
13:
14:     while (1)
15:     {
16:         ch = ext_key();
17:         if (ch == 59)       /* F1? */
18:             break;
19:         else
20:             printf("\nThat key's code has a value of %d.", ch);
21:     }
22: }
23:
24: int ext_key(void)
25: {
26:     int ch;
27:
28:     /* Espera hasta que llegue un byte cero. */
29:
30:     while ((ch = getch()) != 0)
31:         ;
32:
33:     /* Regresa el siguiente carácter. */
34:
35:     return getch();
36: }
```

Análisis

Mientras se tecleen caracteres al programa, `main()` permanece en un ciclo `while` infinito. En el ciclo la línea 16 hace una llamada a `ext_key()`, definida en las líneas 24 a 36. `ext_key()` tiene su propio ciclo `while` que solamente revisa por el carácter `\0`, la señal de que ha sido oprimido un carácter extendido. Una vez que obtiene el carácter

nulo, `ext_key()` regresa el siguiente carácter llamando nuevamente a `getch()`. De regreso en `main()`, la línea 17 evalúa el siguiente valor para ver si es igual a 59 (equivalente a F1). Si es así, un enunciado break hace que se salga del ciclo infinito y el programa termina. Si no fue F1, el programa imprime el valor de la tecla extendida y el ciclo continúa.

La mayoría de los compiladores facilitan la manipulación de la entrada de teclas extendidas, definiendo un juego de constantes simbólicas en el archivo de encabezado. Por lo general, este archivo es CONIO.H. En el compilador Zortech cada una de estas constantes comienza con los caracteres `_KB_`, e identifica a alguna de las teclas extendidas en forma mnemónica. Su valor es igual al valor regresado por la tecla. Por ejemplo, la constante `_KB_F1` está definida como el valor 59. Estas constantes y sus valores se muestran en la tabla 14.3.

Tabla 14.3. Constantes simbólicas para las teclas extendidas definidas en conio.h del compilador Zortech.

#define_KB_F1	59	/* Teclas de función F1 a F10 */
#define_KB_F2	60	
#define_KB_F3	61	
#define_KB_F4	62	
#define_KB_F5	63	
#define_KB_F6	64	
#define_KB_F7	65	
#define_KB_F8	66	
#define_KB_F9	67	
#define_KB_F10	68	
#define_KB_HOME	71	/* Teclas de edición */
#define_KB_UP	72	
#define_KB_PGUP	73	
#define_KB_LEF	75	
#define_KB_RIGHT	77	
#define_KB_END	79	
#define_KB_DOWN	80	
#define_KB_PGDN	81	

Tabla 14.3. continuación

#define_KB_INS	82	
#define_KB_BACK_TAB	15	
#define_KB_SF1	84	/* Mayúsculas F1-F10 */
#define_KB_SF2	85	
#define_KB_SF3	86	
#define_KB_SF4	87	
#define_KB_SF5	88	
#define_KB_SF6	89	
#define_KB_SF7	90	
#define_KB_SF8	91	
#define_KB_SF9	92	
#define_KB_SF10	93	
#define_KB_CF1	94	/* Control F1-F10 */
#define_KB_CF2	95	
#define_KB_CF3	96	
#define_KB_CF4	97	
#define_KB_CF5	98	
#define_KB_CF6	99	
#define_KB_CF7	100	
#define_KB_CF8	101	
#define_KB_CF9	102	
#define_KB_CF10	103	
#define_KB_AF1	104	/* Alt F1-F10 */
#define_KB_AF2	105	
#define_KB_AF3	106	
#define_KB_AF4	107	

```
#define_KB_AF5         108
#define_KB_AF6         109
#define_KB_AF7         110
#define_KB_AF8         111
#define_KB_AF9         112
#define_KB_AF10        113

#define_KB_DEL         83
#define_KB_CPGUP       132     /* Control PgUp */
#define_KB_CLEFT       115     /* Control flecha izquierda */
#define_KB_CRIGHT      116     /* Control flecha derecha */
#define_KB_CEND        117     /* Control End */
#define_KB_CPGDN       118     /* Control PgDn */
#define_KB_CHOME       119     /* Control Home */
#define_KB_A1          120     /* Alt 1 */
#define_KB_A2          121
#define_KB_A3          122
#define_KB_A4          123
#define_KB_A5          124
#define_KB_A6          125
#define_KB_A7          126
#define_KB_A8          127
#define_KB_A9          128
#define_KB_A0          7129    /* Alt 0 */
#define_KB_AMINUS      130     /* Alt '-'.*/
#define_KB_APLUS       131     /* Alt '+'.*/
```

Debe usted revisar los manuales de su compilador para determinar si contienen constantes definidas para las teclas extendidas. Si no es así, añada los enunciados `#define`, al principio

del programa, para las teclas específicas que necesite el programa. Si el compilador no le define las teclas extendidas, tal vez quiera crear un archivo de encabezado con todas las constantes definidas anteriormente. De esta forma podrá incluir el archivo en cualquier programa que necesite los códigos de teclas extendidas (en el Día 21, "Cómo aprovechar las directivas del preprocesador y más", se aprenderán cosas específicas acerca de los archivos de encabezado).

La función `ext_key()` puede ser modificada para aceptar un subjconjunto de las teclas extendidas. Se pueden usar teclas extendidas para crear interfaces, menús y otras cosas parecidas, flexibles y amigables. El programa del listado 14.7 modifica a `ext_key()` para que solamente acepte opresiones de tecla de función, y luego estructura un sistema de menú manejado por teclas de función.



Listado 14.7. Un sistema de menú que responde a la entrada de teclas de función.

```
1: /* Demuestra un menú manejado por teclas de función. */
2:
3: #include <stdio.h>
4: #include <conio.h>
5: #include <time.h>
6:
7: int fnc_key(void);
8: int menu(void);
9:
10: main()
11: {
12:     /* Ajusta un ciclo infinito. */
13:
14:     while (1)
15:     {
16:         /* Switch basado en el valor de retorno de menu(). */
17:
18:         switch  (menu())
19:         {
20:             case _KB_F1:
21:                 puts("Task 1");
22:                 break;
23:             case _KB_F2:
24:                 puts("Task 2");
25:                 break;
26:             case _KB_F3:
27:                 puts("Task 3");
28:                 break;
29:             case _KB_F4:
30:                 puts("Task 4");
31:                 break;
32:             case _KB_F5:
33:                 puts("Task 5");
34:                 break;
```

```
35:             case _KB_F6:
36:                 puts("Task 6");
37:                 break;
38:             case _KB_F7:
39:                 puts("Task 7");
40:                 break;
41:             case _KB_F8:
42:                 puts("Task 8");
43:                 break;
44:             case _KB_F9:
45:                 puts("Task 9");
46:                 break;
47:             case _KB_F10:
48:                 puts("Exiting program...");
49:                 exit(0);
50:         }
51:     }
52: }
53:
54: int menu(void)
55: {
56:     /* Despliega la selecciones de menú. */
57:
58:     puts("\nF1 -> task 1");
59:     puts("F2 -> task 2");
60:     puts("F3 -> task 3");
61:     puts("F4 -> task 4");
62:     puts("F5 -> task 5");
63:     puts("F6 -> task 6");
64:     puts("F7 -> task 7");
65:     puts("F8 -> task 8");
66:     puts("F9 -> task 9");
67:     puts("F10 -> exit\n");
68:
69:     /* Obtiene una opresión de tecla de función. */
70:
71:     return (fnc_key());
72: }
73:
74: int fnc_key(void)
75: {
76:     int ch;
77:
78:     while (1)
79:     {
80:         /* Espera hasta que llegue un byte cero. */
81:
82:         while ((ch = getch()) != 0)
83:             ;
84:         /* Obtiene el siguiente carácter. */
85:
86:         ch = getch();
```

Listado 14.7. continuación

```
87:
88:         /* ¿Es tecla de función? */
89:
90:         if ( ch >= _KB_F1 && ch <= _KB_F10 )
91:             return ch;
92:     }
93: }
```

Salida

```
F1 -> task 1
F2 -> task 2
F3 -> task 3
F4 -> task 4
F5 -> task 5
F6 -> task 6
F7 -> task 7
F8 -> task 8
F9 -> task 9
F10 -> exit

Exiting program...
```

"Desobtención" de un carácter con *ungetc()*

¿Qué significa "desobtener" un carácter? Tal vez quede claro con un ejemplo. Digamos que el programa está leyendo caracteres de un flujo de entrada, y puede detectar el fin de la entrada solamente leyendo un carácter de más. Tal vez esté recibiendo solamente dígitos, por lo que sabe que la entrada ha terminado cuando aparece el primer carácter que no es dígito. Ese primer carácter que no es dígito puede ser parte importante de datos subsecuentes, pero ya ha sido quitado del flujo de entrada. ¿Se ha perdido? No, puede ser "desobtenido" o regresado al flujo de entrada, donde se convierte en el primer carácter que será leído por la siguiente operación de entrada sobre ese flujo.

Para "desobtener" un carácter se usa la función de biblioteca `ungetc()`. Su prototipo es

```
int ungetc(int ch, FILE *fp);
```

El argumento `ch` es el carácter que ha de ser regresado. El argumento `*fp` especifica el flujo al cual será regresado el carácter, que puede ser cualquier flujo de entrada. Por el momento, simplemente especifique `stdin` como el segundo argumento: `ungetc(ch, stdin);`. La notación `FILE *fp` se usa con flujos asociados con archivos de disco. Aprenderá acerca de esto en el Día 16, "Uso de archivos de disco".

Se puede "desobtener" un solo carácter a un flujo entre lecturas, y en ningún momento se puede "desobtener" el `EOF`. La función `ungetc()` regresa a `ch` en caso satisfactorio, y `EOF`, si el carácter no puede ser regresado al flujo. El listado 17.16 usa `ungetc()`.

Entrada de líneas

Las funciones de entrada de líneas leen una línea del flujo de entrada, leen todos los caracteres hasta el siguiente carácter de nueva línea. La biblioteca estándar tiene dos funciones de entrada de línea, `gets()` y `fgets()`.

La función *gets()*

Ya se le presentó la función `gets()` en el Día 10, "Caracteres y cadenas". Esta es una función llana, que lee una línea de `stdin` y la guarda en una cadena. El prototipo de función es

```
char *gets(char *str);
```

Probablemente puede interpretar este prototipo por usted mismo. `gets()` toma un apuntador a tipo `char` como argumento y regresa un apuntador a tipo `char`. La función `gets()` lee caracteres de `stdin` hasta que encuentra una nueva línea `\n` o un fin de archivo. La nueva línea es reemplazada con un carácter nulo, y la cadena, guardada en la posición indicada por `str`.

El valor de retorno es un apuntador a la cadena (el mismo que `str`). Si `gets()` encuentra un error, o lee un fin de archivo antes de recibir cualquier entrada, regresa un apuntador nulo.

Antes de llamar a `gets()` se debe asignar suficiente espacio de memoria para guardar la cadena, con los métodos tratados en el Día 10, "Caracteres y cadenas". La función no tiene forma de saber si el espacio para `ptr` ha sido asignado o no. En cualquier caso la cadena es recibida y guardada comenzando en `ptr`. Si el espacio no ha sido asignado, la cadena puede sobreescribir otros datos y causar errores de programa.

Los listados 10.5 y 10.6 usan `gets()`.

La función *fgets()*

La función de biblioteca `fgets()` es similar a `gets()`, en que lee una línea de texto de un flujo de entrada. Es más flexible, ya que permite que el programador especifique el flujo de entrada específico que debe usar y la cantidad máxima de caracteres que debe ser recibida. Por lo general, la función `fgets()` se usa para recibir texto de archivos de disco (tratado en el Día 16, "Uso de archivos de disco"). Para usarla a fin de recibir entrada de `stdin`, se especifica a `stdin` como el flujo de entrada. El prototipo de `fgets()` es

```
char *fgets(char *str, int n, FILE *fp);
```

El último parámetro `FILE *fp` se usa para especificar el flujo de entrada. Por el momento, simplemente especifique `stdin` como el argumento de flujo.

El apuntador `str` indica dónde será guardada la cadena de entrada. El argumento `n` especifica la cantidad máxima de caracteres que ha de ser recibida. La función `fgets()` lee

caracteres de la corriente de entrada hasta que encuentra una nueva línea, o un fin de archivo, o se han leído `n - 1` caracteres. La nueva línea es incluida en la cadena terminada con un `\0` antes de ser guardada. Los valores de retorno de `fgets()` son los mismos que se describieron anteriormente para `gets()`.

Hablando estrictamente, `fgets()` no recibe una sola línea de texto (en caso de definir una línea como una secuencia de caracteres que termina con una nueva línea). Puede leer menos que una línea completa, si la línea contiene más de `n - 1` caracteres. Cuando se usa con `stdin`, la ejecución no regresa de `fgets()` sino hasta que se oprime Enter, pero solamente son guardados en la cadenas los primeros `n - 1` caracteres. La nueva línea es incluida en la cadena sólo si se encuentra dentro de los primeros `n - 1` caracteres.

El programa del listado 14.8 muestra a `fgets()`. Cuando ejecute el programa, teclee líneas de longitud menor y mayor que `MAXLEN` para ver qué pasa. Si se da una línea mayor que `MAXLEN`, los primeros `MAXLEN - 1` caracteres son leídos por la primera llamada a `fgets()`. Los caracteres restantes permanecen en el buffer del teclado y son leídos por la siguiente llamada a `fgets()`.



Listado 14.8. Uso de la función `fgets()` para la entrada del teclado.

```
1: /* Demuestra la función fgets(). */
2:
3: #include <stdio.h>
4:
5: #define MAXLEN 10
6:
7: main()
8: {
9:     char buffer[MAXLEN];
10:
11:     puts("Enter text a line at a time; enter a blank to \
            exit.");
12:
13:     while (1)
14:     {
15:         fgets(buffer, MAXLEN, stdin);
16:
17:         if (buffer[0] == '\n')
18:             break;
19:
20:         puts(buffer);
21:     }
22: }
```

Salida

```
Enter text a line at a time; enter a blank to exit.
Roses are red
Roses are
 red
Violets are blue
Violets a
re blue

Programming in C
Programmi
ng in C

Is for people like you!
Is for pe
ople like
 you!
```

Entrada formateada

Las funciones de entrada tratadas hasta ahora simplemente han tomado uno o más caracteres de un flujo de entrada y los han puesto en algún lugar en memoria. No se ha hecho ninguna interpretación o formateo de la entrada, y todavía no se tiene algún método para asignar la entrada de varios caracteres a variables numéricas. Por ejemplo, ¿cómo podría recibir el valor 12.86 desde el teclado y asignarlo a una variable tipo `float`? Use las funciones `scanf()` y `fscanf()`. En el Día 7, "Entrada/salida básica", le fue presentada `scanf()`, y esta sección explica su uso más minuciosamente.

Estas dos funciones son idénticas, a excepción de que `scanf()` siempre usa `stdin` y, en cambio, en `fscanf()`, el usuario puede especificar el flujo de entrada. Esta sección trata sobre `scanf()`. Por lo general, `fscanf()` se usa con la entrada de archivos de disco y, se trata en el Día 16, "Uso de archivos de disco".

Los argumentos de la función *scanf()*

La función `scanf()` toma una cantidad variable de argumentos y requiere por lo menos dos. El primer argumento es una cadena de formato, que usa caracteres especiales para decirle a `scanf()` la manera en que debe interpretar la entrada. El segundo y los demás argumentos son las direcciones de la o las variables a las cuales son asignados los datos de entrada. Este es un ejemplo:

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

El primer argumento, `"%d"`, es la cadena de formato. En este caso `%d` le dice a `scanf()` que busque un valor entero con signo. El segundo argumento usa el operador (&) de *dirección de* para decirle a `scanf()` que asigne el valor de entrada a la variable `x`. Ahora podemos ver los detalles de la cadena de formato.

La cadena de formato de `scanf()` puede contener

- Espacios y tabuladores, que son ignorados (pueden usarse para hacer más legible la cadena de formato).
- Caracteres (a excepción de %) que son apareados contra caracteres diferentes de blanco en la entrada.
- Una o más *especificaciones de conversión,* que consisten en el carácter % seguido de caracteres especiales. Por lo general, la cadena de formato contiene una especificación de conversión para cada variable.

La única parte requerida de la cadena de formato es la especificación de conversión. Cada especificación de conversión comienza con el carácter `%`, y contiene componentes opcionales y requeridos en determinado orden. La función `scanf()` aplica en orden, a los campos de entrada, las especificaciones de conversión que se encuentran en la cadena de formato. Un campo de entrada es una secuencia de caracteres no blancos, que termina cuando se encuentra el siguiente espacio en blanco o cuando se alcanza el ancho de campo, en caso de haber sido especificado. Los componentes de la especificación de conversión son:

- El indicador opcional de supresión de asignación (*), que se pone inmediatamente después del %. En caso de estar presente, este carácter le dice a `scanf()` que ejecute la conversión correspondiente al especificador de conversión actual pero que ignore el resultado (que no lo asigne a ninguna variable).
- El siguiente componente es el ancho de campo, también opcional. El ancho de campo es un número decimal que especifica en caracteres, el ancho, del campo de entrada. En otras palabras, el ancho de campo especifica qué tantos caracteres de `stdin` debe examinar `scanf()` para la conversión actual. Si no se especifica un ancho de campo, el campo de entrada se extiende hasta el siguiente espacio en blanco.
- El siguiente componente es el modificador opcional de precisión, un solo carácter que puede ser `h`, `l`, o `L`. En caso de estar presente, el modificador de precisión cambia el significado del siguiente especificador de tipo. Los detalles se dan posteriormente en este capítulo.
- El único componente requerido del especificador de conversión (aparte del %) es el especificador de tipo. El especificador de tipo consiste en uno o más caracteres que le dicen a `scanf()` cómo interpretar la entrada. Los caracteres se listan y explican en la tabla 14.4. La columna Argumento lista los tipos requeridos de las variables correspondientes. Por ejemplo, el especificador de tipo d requiere `int *` (un apuntador a tipo `int`).

Tabla 14.4. Los caracteres especificadores de tipo usados en los especificadores de conversión.

Tipo	Significado	Argumento
d	Un entero decimal.	`int *`
i	Un entero en notación decimal, octal (con `0` inicial) o hexadecimal (con `0X` o `0x` inicial).	`int *`
o	Un entero en notación octal con o sin el `0` inicial.	`int *`
u	Un entero decimal sin signo.	`unsigned int *`
x	Un entero hexadecimal con o sin el `0X` o `0x` inicial.	`int *`
c	Uno o más caracteres son leídos y asignados secuencialmente a la posición de memoria indicada por el argumento. No se añade `\0` terminal. Si no se da el argumento de ancho de campo, se lee un carácter. Si se da un argumento de ancho de campo se leen esa cantidad de caracteres incluidos los espacios en blanco (en caso de haberlos).	`char *`
s	Una cadena de caracteres sin espacio en blanco es leída en la posición de memoria especificada y se añade un `\0` terminal.	`char *`
e,f,g	Un número de punto flotante. Los números pueden ser dados en notación decimal o científica.	`float*`
[...]	Una cadena. Solamente son aceptados los caracteres listados entre corchetes. La entrada termina tan pronto como se encuentra un carácter que no concuerde, se llegue al ancho de campo especificado o se oprima Enter. Para aceptar el carácter], lístelo al principio: `[]...]`. Se añade un `\0` al final de la cadena.	`char *`
[^...]	Similar a `[...]`, a excepción de que sólo son aceptados los caracteres que no están listados entre corchetes.	`char *`
%	Literal `%`: lee el carácter `%`. No se hace ninguna asignación.	ninguno

Antes de ver algunos ejemplos de `scanf()`, es necesario que se comprenda los modificadores de precisión:

`h` Cuando se pone antes de los especificadores de tipo `d`, `i`, `o`, `u` o `x`, el modificador `h` especifica que el argumento es un apuntador a tipo `short`, en vez de tipo `int`. En una PC el tipo `short` es el mismo que tipo `int`, por lo que nunca es necesario el modificador de precisión `h`.

`l` Cuando se pone antes de los especificadores `d`, `i`, `o`, `u` o `x`, el modificador l especifica que el argumento es un apuntador a tipo long. Cuando se pone antes de los especificadores de tipo `e`, `f` o `g`, el modificador l especifica que el argumento es un apuntador a tipo `double`.

`L` Cuando se pone antes de los especificadores de tipo `e`, `f` o `g`, el modificador L especifica que el argumento es un apuntador a tipo `long double`.

Manejo de caracteres adicionales

La entrada de `scanf()` se almacena temporalmente. De hecho, ningún carácter se recibe desde `stdin` sino hasta que el usuario oprime Enter. Luego, la línea completa de caracteres "llega" de `stdin`, y es procesada, en orden, por `scanf()`. La ejecución regresa de `scanf()` sólo cuando se ha recibido suficiente entrada para satisfacer la especificación de la cadena de formato. También `scanf()` procesa solamente los suficientes caracteres de `stdin` para satisfacer su cadena de formato. Los caracteres adicionales que no son necesarios, en caso de haber algunos, permanecen "esperando" en `stdin`. Estos caracteres pueden causar problemas. Veamos más de cerca la operación de `scanf()` para ver qué pasa.

Cuando se ejecuta una llamada a `scanf()` y el usuario ha tecleado una sola línea, se pueden tener tres situaciones. Para estos ejemplos suponga que se está ejecutando `scanf("%d %d", &x, &y);`, en otras palabras, `scanf()` está esperando dos enteros decimales. Las posibilidades son:

1. La línea que el usuario envía se ajusta a la cadena de formato. Por ejemplo, el usuario teclea `12 14` seguido de Enter. En este caso no hay problemas: scanf() queda satisfecho y no quedan caracteres en `stdin`.
2. La línea que envía el usuario tiene muy pocos elementos como para satisfacer la cadena de formato. Por ejemplo, el usuario teclea 12 seguido de Enter. En este caso, `scanf()` continúa esperando la entrada faltante. Una vez que se recibe la entrada la ejecución continúa y no quedan caracteres en `stdin`.
3. La línea que envía el usuario tiene más elementos que los requeridos por la cadena de formato. Por ejemplo, el usuario teclea `12 14 16` seguido de Enter. En este caso `scanf()` lee el `12` y el `14` y luego regresa. Los caracteres adicionales, el `1` y el `6`, quedan "esperando" en `stdin`.

Es esta tercera situación (específicamente aquellos caracteres que han quedado) es la que puede causar problemas. Ellos quedan esperando por todo el tiempo que está ejecutando el programa hasta la siguiente vez que el programa lea la entrada de `stdin`. Luego estos caracteres adicionales son los primeros en ser leídos, antes que cualquier entrada que el usuario haga en ese momento. ¡No hay duda de que esto puede causar errores!

A continuación se presenta un ejemplo. El siguiente código le pide al usuario que teclee un entero y luego que teclee una cadena:

```
puts("Teclee su edad.")
scanf("%d", &edad);
puts("Teclee su nombre.");
scanf("%d", nombre);
```

Digamos por ejemplo, que en respuesta a la primera pregunta el usuario decide ser preciso, y teclea `29.00` y luego oprime Enter. La primera llamada a `scanf()` está buscando un entero, por lo que lee los caracteres `29` de `stdin` y asigna el valor `29` a la variable edad. Los caracteres `.00` quedan en espera en `stdin`. La siguiente llamada a `scanf()` está esperando una cadena. Va a stdin por la entrada y encuentra a `.00` esperando ahí. El resultado es que la cadena `.00` es asignada a `nombre`.

¿Cómo se puede evitar este problema? Si la gente que usa el programa nunca comete errores, esa es una solución, ¡aunque bastante impráctica!

Una mejor solución es asegurarse de que no haya caracteres adicionales esperando en `stdin` antes de que se le pida entrada al usuario. Se puede hacer esto con una llamada a `gets()`, la cual lee cualquier carácter pendiente en `stdin` hasta, incluido, el fin de línea. En vez de llamar a `gets()` directamente desde el programa, se le puede poner en una función separada con el nombre descriptivo `clear_kb()`. Esta función se muestra en el listado 14.9.

Captura

Listado 14.9. Limpieza de caracteres adicionales en `stdin` para evitar errores.

```
1: /* Limpieza de los caracteres adicionales de stdin. /*
2:
3: #include <stdio.h>
4:
5: void clear_kb(void);
6:
7: main()
8: {
9:     int age;
10:     char name[20];
11:
12:     /* Pide la edad del usuario. */
13:
14:     puts("Enter your age.");
```

Listado 14.9. continuación

```
15:     scanf("%d", &age);
16:
17:     /* Limpia a stdin de cualquier carácter adicional. */
18:
19:     clear_kb();
20:
21:     /* Pide ahora el nombre del usuario. */
22:
23:     puts("Enter your first name.");
24:     scanf("%s", name);
25:     /* Display the data. */
26:
27:     printf("Your age is %d.\n", age);
28:     printf("Your name is %s.\n", name);
29: }
30:
31: void clear_kb(void)
32:
33: /* Limpia a stdin de cualquier carácter que esté en espera. */
34: {
35:     char junk[80];
36:     gets(junk);
37: }
```

Salida

```
Enter your age.
29
Enter your first name.
Bradley Jones
Your age is 29.
Your name is Bradley.
```

Análisis

Cuando ejecute el programa del listado 14.9, teclee algunos caracteres adicionales después de la edad, antes de oprimir Enter. Asegúrese de que el programa los ignora y le pide correctamente el nombre. Luego, modifique el programa, quitando la llamada a `clear_kb`, y vuélvalo a ejecutar. Cualquier carácter adicional tecleado en la misma línea que la edad es asignado a `name`.

Ejemplos de *scanf()*

La mejor manera de familiarizarse con la operación de `scanf()` es usarla. Es una función poderosa, pero a veces puede ser un poco confusa. Pruébela y vea lo que pasa. El programa del listado 14.10 muestra algunas de las formas poco usuales de usar `scanf()`. Deberá compilar y ejecutar el programa, y luego hacer algunos experimentos cambiando las cadenas de formato de `scanf()`.

Captura

Listado 14.10. Algunas maneras de usar `scanf()` para la entrada del teclado.

```
/* Demuestra algunos usos de scanf(). */

#include <stdio.h>

void clear_kb(void);

main()
{
    int i1, i2;
     long l1;
     float f1;
     double d1;
     char buf1[80], buf2[80];

     /* Uso del modificador l para recibir enteros largos y
        dobles. */

     puts("Enter an integer and a floating point number.");
     scanf("%ld %lf", &l1, &d1);
     printf("You entered %ld and %lf.\n",l1, d1);
     puts("The scanf() format string used the l modifier to  \
          store");
     puts("your input in a type long and a type double.\n");

     clear_kb();

     /* Usa el ancho de campo para dividir la entrada. */

     puts("Enter a 5 digit integer (for example, 54321).");
     scanf("%2d%3d", &i1, &i2);

     printf("You entered %d and %d.\n", i1, i2);
     puts("Note how the field width specifier in the scanf() \
          format");
     puts("string split your input into two values.\n");

     clear_kb();

     /* Usa un espacio excluido para dividir una línea de entrada */
     /* donde se encuentra el espacio en dos cadenas */

     puts("Enter your first and last names separated by a \
               space.");
     scanf("%[^ ]%s", buf1, buf2);
     printf("Your first name is %s\n", buf1);
     printf("Your last name is %s\n", buf2);
     puts("Note how [^ ] in the scanf() format string, by \
          excluding");
```

Listado 14.10. continuación

```
44:     puts("the space character, caused the input to be split.");
45: }
46:
47: void clear_kb(void)
48:
49: /* Limpia a stdin de cualquier carácter en espera. */
50: {
51:     char junk[80];
52:     gets(junk);
53: }
```

Salida

```
Enter an integer and a floating point number.
123 45.6789
You entered 123 and 45.678900.
The scanf() format string used the l modifier to store
your input in a type long and a type double.

Enter a 5 digit integer (for example, 54321).
54321
You entered 54 and 321.
Note how the field width specifier in the scanf() format
string split your input into two values.

Enter your first and last names separated by a space.
Peter Aitken
Your first name is Peter
Your last name is Aitken
Note how [^ ] in the scanf() format string, by excluding
the space character, caused the input to be split.
```

Análisis

Este listado comienza definiendo diversas variables en las líneas 9 a 13 para la entrada de datos. El programa luego lo lleva por los pasos para recibir varios tipos de datos. Las líneas 15 a 21 le piden que teclee e imprima enteros largos y un doble. La línea 23 llama a la función `clear_kb()`, para quitar cualesquier caracteres no deseados del flujo de entrada. Las líneas 27 a 28 obtienen el siguiente valor, un entero de 5 caracteres. Como hay especificadores de ancho, el entero de 5 dígitos es dividido en dos enteros: uno con dos caracteres y otro con tres caracteres. La línea 34 llama a `clear_kb()` para volver a limpiar el teclado. El ejemplo final, líneas 36 a 44, usa el carácter de exclusión. La línea 40 usa `"%[^ ]"`, que le dice a `scanf()` que reciba una cadena pero que pare en cualquier espacio. Esto, efectivamente, divide la entrada.

Debe tomarse un tiempo modificando este listado y tecleando valores adicionales para ver qué es lo que pasa.

La función `scanf()` puede ser usada para la mayoría de las necesidades de entrada, en particular aquellas involucradas con números (las cadenas son recibidas más fácilmente con `gets()`). Sin embargo, frecuentemente vale la pena escribir las propias funciones de entrada

especializadas. Podrá ver algunos ejemplos de funciones definidas por el usuario en el Día 18, "Cómo obtener más de las funciones".

DEBE	NO DEBE
DEBE Aprovechar los caracteres extendidos en el programa. Cuando use caracteres extendidos, debe tratar de ser consistente con otros programas.	
	NO DEBE Olvidar revisar el flujo de entrada por si hay caracteres adicionales.
DEBE Usar las funciones `gets()` y `scanf()`, en vez de `fgets()` y `fscanf()`, si solamente está usando el archivo de entrada estándar (`stdin`).	

Salida a pantalla

Las funciones de salida a pantalla se dividen en tres categorías generales, siguiendo las mismas líneas que las funciones de entrada: salida de caracteres, salida de líneas y salida formateada. Ya le han sido presentadas algunas de estas funciones en capítulos anteriores. Esta sección trata todas ellas minuciosamente.

Salida de caracteres con *putchar(), putc() y fputc()*

Las funciones de salida de carácter de la biblioteca del C envían un solo carácter a un flujo. La función `putchar()` envía su salida a `stdout` (que, por lo general, es la pantalla). Las funciones `fputc()` y `putc()` envían su salida a un flujo especificado en la lista de argumentos.

Uso de la función *putchar()*

El prototipo para `putchar()` se encuentra en STDIO.H y dice

```
int putchar(int c);
```

La función escribe en `stdout` el carácter guardado en `c`. Aunque el prototipo especifica un argumento tipo `int`, se le puede pasar a `putchar()` un tipo `char`. También se le puede pasar un tipo `int`, siempre y cuando su valor sea adecuado para un carácter (esto es, que se encuentre en el rango de 0 a 255). La función regresa el carácter que acaba de ser escrito, o `EOF`, en caso de que haya habido algún error.

Ya se vio demostrada a `putchar()` en el listado 14.2. El programa del listado 14.11 despliega los caracteres con valores ASCII entre 14 y 127.

Captura **Listado 14.11. La función `putchar()`.**

```
1: /* Demuestra a putchar(). */
2:
3: #include <stdio.h>
4: main()
5: {
6:     int count;
7:
8:     for (count = 14; count < 128; )
9:         putchar(count++);
10: }
```

También puede desplegar cadenas con la función `putchar()` (como se hace en el listado 14.12), aunque otras funciones son más adecuadas para este objeto.

Captura **Listado 14.12. Despliegue de una cadena con `putchar()`.**

```
1: /* Uso de putchar() para desplegar cadenas. */
2:
3: #include <stdio.h>
4:
5: #define MAXSTRING 80
6:
7: char message[] = "Displayed with putchar().";
8: main()
9: {
10:     int count;
11:
12:     for (count = 0; count < MAXSTRING; count++)
13:     {
14:
15:         /* Busca el final de la cadena. Cuando lo encuentre, */
16:         /* escribe un carácter de nueva línea y sale del ciclo. */
17:
18:         if (message[count] == '\0')
19:         {
20:             putchar('\n');
21:             break;
22:         }
23:         else
24:
25:         /* Si no encuentra el final de la cadena,
              escribe el siguiente carácter */
26:
27:             putchar(message[count]);
28:     }
29: }
```




```
Displayed with putchar().
```

Uso de las funciones *putc() y fputc()*

Estas dos funciones ejecutan la misma acción, el envío de un solo carácter a un flujo especificado. `putc()` es una implementación macro de `fputc()`. Aprenderá acerca de las macros en el Día 21, "Cómo aprovechar las directivas del preprocesador y más". Por ahora, apéguese a `fputc()`. Su prototipo es

```
int fputc(int c, FILE *fp);
```

Tal vez le extrañe la parte `FILE *fp` del prototipo. Se le pasa a `fputc()` el flujo de salida en este argumento. Aprenderá más acerca de esto en el Día 16, "Uso de archivos de disco". Si se especifica `stdout` como el flujo, `fputc()` se comporta exactamente igual como `putchar()`. Por lo tanto, los siguientes dos enunciados son equivalentes:

```
putchar('x');
fputc('x', stdout);
```

Uso de *puts() y fputs()* para la salida de flujos

El programa despliega cadenas en la pantalla con más frecuencia que simples caracteres. La función de biblioteca `puts()` despliega cadenas. La función `fputs()` envía una cadena a un flujo determinado, y en lo demás es idéntica a `puts()`. El prototipo para `puts()` es

```
int puts(char *cp);
```

siendo `*cp` un apuntador al primer carácter de la cadena que se quiere desplegar. La función `puts()` despliega la cadena completa hasta el carácter nulo terminal, pero sin incluirlo, añadiendo una nueva línea al final. Luego, `puts()` regresa un valor positivo, si el resultado es satisfactorio, y `EOF` cuando hay error (recuerde que `EOF` es una constante simbólica con el valor -1 y está definida en STDIO.H).

La función `puts()` puede ser usada para desplegar cualquier tipo de cadena. Su uso se demuestra en el listado 14.13.

Captura

Listado 14.13. Uso de la función `puts()` para desplegar cadenas.

```
1: /* Demuestra a puts(). */
2:
3: #include <stdio.h>
4:
5:/* Declara e inicializa un arreglo de apuntadores. */
6:
7: char *messages[5] = { "This", "is", "a", "short", "message." };
8:
9: main()
10: {
11:     int x;
12:
13:     for (x=0; x<5; x++)
14:         puts(messages[x]);
15:
16:     puts("And this is the end!");
17: }
```

Salida

```
This
is
a
short
message.
And this is the end!
```

Análisis

Este listado declara un arreglo de apuntadores, un tema que todavía no se ha tratado (pero se tratará mañana, Día 15, "Más sobre apuntadores"). Las líneas 13 y 14 imprimen cada una de las cadenas guardadas en el arreglo de mensajes.

Uso de *printf()* y *fprintf()* para la salida formateada

Hasta ahora las funciones de salida han desplegado solamente caracteres y cadenas. ¿Qué hay de los números? Para desplegar números se deben usar las funciones de salida formateada de la biblioteca del C `printf()` y `fprintf()`. Estas funciones también pueden desplegar cadenas y caracteres. Ya fue presentado oficialmente con `printf()` en el Día 7, "Entrada/salida básica", y la ha usado en casi todos los capítulos. Esta sección proporciona los detalles faltantes.

Las dos funciones `printf()` y `fprintf()` son idénticas, a excepción de que `printf()` siempre envía la salida a `stdout`, mientras que a `fprintf()` se le especifica el flujo de salida. Generalmente, `fprintf()` se usa para la salida a archivos de disco, y se trata en el Día 16, "Uso de archivos de disco".

La función `printf()` toma un número variable de argumentos con un mínimo de uno. El primer y único argumento requerido es la cadena de formato, que le dice a `printf()` la manera de formatear la salida. Los argumentos opcionales son variables y expresiones cuyos valores se quieren desplegar. Dé una vista a unos cuantos ejemplos simples, para que se ambiente con `printf()` antes de que en realidad profundice en ella.

- El enunciado `printf("Hola, mundo.");` despliega el mensaje `Holá, mundo.` en la pantalla. Este es un ejemplo del uso de `printf()` con un solo argumento, la cadena de formato. En este caso, la cadena de formato contiene solamente una cadena literal que ha de ser desplegada en la pantalla.
- El enunciado `printf("%d", i);` despliega el valor de la variable entera `i` en la pantalla. La cadena de formato contiene solamente el especificador de formato %d, que le dice a `printf()` que despliegue un solo entero decimal. El segundo argumento, `i`, es el nombre de la variable cuyo valor será desplegado.
- El enunciado `printf("%d más %d igual a %d.", a, b, a+b);` despliega en pantalla `2 más 3 igual a 5` (suponiendo que a y b son variables enteras con los valores `2` y `3` respectivamente). Este uso de `printf()` tiene cuatro argumentos: una cadena de formato que contiene el texto literal así como especificadores de formato, dos variables y una expresión cuyos valores serán desplegados.

Ahora veamos más minuciosamente la cadena de formato de `printf()`. Puede contener

- Uno o más comandos de conversión, que le dicen a `printf()` la manera de desplegar un valor en su lista de argumentos. Un comando de conversión consiste en un %, seguido por uno o más caracteres.
- Caracteres que no son parte del comando de conversión y son desplegados tal cual.

La cadena de formato del tercer ejemplo es `%d más %d igual a %d`. En este caso, los tres `%d` son comandos de conversión, y el resto de la cadena, incluidos los espacios, son los caracteres literales que son desplegados directamente.

Ahora puede analizar componente a componente el comando de conversión. Los componentes del comando se dan aquí y se explican a continuación. Los componentes entre corchetes son opcionales.

`%[indicador][ancho_de_campo][.[precisión]][l]carácter_de_conversión`

El `carácter_de_conversión` es la única parte requerida del comando de conversión (además del `%`). Los caracteres de conversión y su significado se listan en la tabla 14.5.

Tabla 14.5. Los caracteres de conversión para `printf()` y `fprintf()`.

Carácter de conversión	Significado
`d, i`	Despliega un entero con signo en notación decimal.
`u`	Despliega un entero sin signo en notación decimal.
`o`	Despliega un entero en notación octal sin signo.
`x, X`	Despliega un entero en notación hexadecimal sin signo. Use `x` para la salida de minúsculas, `X` para la salida de mayúsculas.
`c`	Despliega un solo carácter (el argumento da el código ASCII y el carácter).
`e, E`	Despliega un `float` o `double` en notación científica (por ejemplo, `123.45` es desplegado como `1.234500e+002`). Se despliegan seis dígitos a la derecha del punto decimal, a menos que se especifique otra precisión (véase abajo). Use `e` o `E` para controlar la salida en mayúsculas o minúsculas.
`f`	Despliega un `float` o `double` en notación decimal (por ejemplo, `123.45` es desplegado como `123.450000`). Se despliegan seis dígitos a la derecha del punto decimal, a menos que se especifique otra precisión.
`g, G`	Usa formato `e, E, o f`. El formato `e` o `E` se usa si el exponente es menor que `-3` o mayor que la precisión (cuyo valor predeterminado es `6`. En caso contrario se usa el formato `f`). Se eliminan los ceros a la derecha.
`n`	Nada se despliega. El argumento correspondiente a un comando de conversión `n` es un apuntador a tipo `int`. La función `fprintf()` asigna a esta variable la cantidad de caracteres que han sido sacados hasta el momento.
`s`	Despliega una cadena. El argumento es un apuntador a `char`. Los caracteres son desplegados hasta que se encuentra un carácter nulo o se llega a la cantidad de caracteres especificada por precisión (que por omisión es `32767`). El carácter terminal nulo no aparece a la salida.
`%`	Despliega el carácter `%`.

Se puede poner el modificador l inmediatamente antes del carácter de conversión. Este modificador se aplica solamente a los caracteres de conversión `o, u, x, X, i, d, b`. Cuando es aplicado, este modificador especifica que el argumento es de tipo `long`, en vez de ser de tipo `int`. Si el modificador `l` es aplicado a los caracteres de conversión `e, E, f, g, G`, especifica que el argumento es un número de precisión tipo `double`. Si se pone un `l` antes de cualquier otro carácter de conversión, es ignorado.

El especificador de precisión consiste en un punto decimal (.) solo o seguido por un número. Un especificador de precisión se aplica solamente a los caracteres de conversión `e E f g G s`. Especifica la cantidad de dígitos que se han de desplegar a la derecha del punto decimal o, cuando se usa con `s`, la cantidad de caracteres de la salida. Si el punto decimal se usa solo, especifica una precisión de 0.

El especificador de ancho de campo determina la cantidad mínima de caracteres de la salida. El especificador de ancho de campo puede ser:

- Un entero decimal que no comience con 0. La salida es rellenada a la izquierda con espacios para completar el ancho de campo indicado.
- Un entero decimal comenzando con 0. La salida es rellenada a la izquierda con ceros para completar el ancho de campo indicado.
- El carácter *. El valor del siguiente argumento (que debe ser un int) se usa como el ancho de campo. Por ejemplo, si `w` es un tipo `int` con un valor de 10, el enunciado `printf("%*d", w, a);` imprime el valor con un ancho de campo de 10.

Si no se especifica ancho de campo, o si el ancho de campo especificado es más angosto que la salida, el campo de salida es tan ancho como se necesite.

La última parte opcional de la cadena de formato de `printf()` es el indicador que está inmediatamente a continuación del carácter %. Se dispone de cuatro indicadores:

- - significa que la salida es alineada a la izquierda en este campo, en vez de la alineación derecha predeterminada.
- + significa que los números con signo siempre son desplegados con un + o - al principio.
- ' ' (espacio) significa que los números positivos son precedidos por un espacio.
- # se aplica solamente a los caracteres de conversión `x, X` y `o`. Especifica que los números diferentes de cero son desplegados precedidos con un `0X` o `0x` (para `x` y `X`) o con `0` (para o).

Cuando use a `printf()`, la cadena de formato puede ser una cadena literal encerrada en comillas dobles en la lista de argumentos de `printf()`. También puede ser una cadena terminada en nulo guardada en memoria, en cuyo caso se pasa a `printf()` un apuntador a la cadena. Por ejemplo,

```
char *fmt = "La respuesta es %f.";
printf(fmt, x);
```

es equivalente a

```
printf("La respuesta es %f.", x);
```

Como se explicó en el Día 7, "Entrada/salida básica", la cadena de formato de `printf()` puede contener secuencia de escape que proporciona control especial sobre la salida. La tabla 14.6 lista las secuencias de escape usadas más frecuentemente. Por ejemplo, al incluir la secuencia de nueva línea (`\n`) en una cadena de formato, se hace que la salida subsecuente aparezca comenzando en el siguiente renglón de la pantalla.

Tabla 14.6. Las secuencias de escape más frecuentemente usadas.

Secuencia	Significado
`\a`	Campana (alerta)
`\b`	Retroceso
`\n`	Nueva línea
`\t`	Tabulador horizontal
`\\`	Diagonal inversa
`\?`	Signo de interrogación
`\'`	Comilla sencilla
`\"`	Comilla doble

`printf()` es algo complicado. La mejor manera de aprender su uso es ver los ejemplos y luego experimentar. El programa del listado 14.14 demuestra algunas de las formas en que puede usar a `printf()`.

Captura **Listado 14.14. Algunas maneras de usar la función `printf()`.**

```
1: /* Demostración de printf(). */
2:
3: #include <stdio.h>
4:
5: char *m1 = "Binary";
6: char *m2 = "Decimal";
7: char *m3 = "Octal";
8: char *m4 = "Hexadecimal";
9:
10: main()
11: {
```

```
12:     float d1 = 10000.123
13:     int n, f;
14:
15:
16:     puts("Outputting a number with different field widths.\n");
17:
18:     printf("%5f\n", d1);
19:     printf("%10f\n", d1);
20:     printf("%15f\n", d1);
21:     printf("%20f\n", d1);
22:     printf("%25f\n", d1);
23:
24:     getch();
25:
26:     puts("\nUse the * field width specifier to obtain field \
            width");
27:     puts("from a variable in the argument list.\n");
28:
29:     for (n=5; n <=25; n+=5)
30:         printf("%*f\n", n, d1);
31:
32:     getch();
33:
34:     puts("\nInclude leading zeros.\n");
35:
36:     printf("%05f\n", d1);
37:     printf("%010f\n", d1);
38:     printf("%015f\n", d1);
39:     printf("%020f\n", d1);
40:     printf("%025f\n", d1);
41:
42:     getch();
43:
44:     puts("Display in octal, decimal, and hexadecimal.");
45:     puts("Use # to precede octal and hex output with 0 and \
            0X.");
46:     puts("Use - to left-justify each value in its field.");
47:     puts("First display column labels.\n");
48:
49:     printf("%-15s%-15s%-15s", m2, m3, m4);
50:
51:     for (n = 1; n < 20; n++)
52:         printf("\n%-15d%-#15o%-#15X", n, n, n);
53:
54:     getch();
55:
56:     puts("\n\nUse the %n conversion command to count \
            characters.\n");
57:
58:     printf("%s%s%s%s%n", m1, m2, m3, m4, &n);
59:
```

Listado 14.14. continuación

```
60:     printf("\n\nThe last printf() output %d characters.", n);
61:
62:     getch();
63: }
```

Salida

```
Outputting a number with different field widths.
10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047

Use the * field width specifier to obtain field width
from a variable in the argument list.

10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047

Include leading zeros.

10000.123047
10000.123047
00010000.123047
0000000010000.123047
000000000000010000.123047
Display in octal, decimal, and hexadecimal.
Use # to precede octal and hex output with 0 and 0X.
Use - to left-justify each value in its field.
First display column labels.


Decimal         Octal           Hexadecimal
1               01              0X1
2               02              0X2
3               03              0X3
4               04              0X4
5               05              0X5
6               06              0X6
7               07              0X7
8               010             0X8
9               011             0X9
10              012             0XA
11              013             0XB
12              014             0XC
13              015             0XD
```

```
14          016          0XE
15          017          0XF
16          020          0X10
17          021          0X11
18          022          0X12
19          023          0X13

Use the %n conversion command to count characters.
BinaryDecimalOctalHexadecimal
The last printf() output 29 characters.
```

Redirección de la entrada y la salida

Un programa que usa `stdin` y `stdout` puede utilizar una característica del sistema operativo llamada *redirección.* La redirección le permite hacer lo siguiente:

- La salida enviada a `stdout` puede ser enviada a un archivo de disco o a la impresora, en vez de a la pantalla.
- La entrada del programa de `stdin` puede venir de un archivo de disco en vez del teclado.

No se codifica la redirección en los programas, sino que se especifica en la línea de comandos cuando se ejecuta el programa. En el DOS, los símbolos para redirección son < y >. La redirección de la salida se trata primero.

¿Recuerda su primer programa en C: HELLO.C? Usa la función de biblioteca `printf()` para desplegar el `mensaje Hola, mundo en la pantalla`. Como sabe ahora, `printf()` envía la salida a `stdout`, por lo que puede ser redirigido. Cuando teclee el nombre del programa en la línea de comandos del DOS, póngale a continuación el símbolo > y el nombre del nuevo destino.

```
hello >destino
```

Por lo tanto, si se teclea hello `>prn`, la salida del programa va a la impresora, en vez de a la pantalla (`prn` es el nombre del DOS para la impresora conectada al puerto LPT1:). Si se teclea `hello >hello.txt`, la salida es puesta en un archivo de disco con el nombre `hello.txt`.

Tenga cuidado cuando redireccione la salida a un archivo de disco. Si el archivo ya existe, la copia anterior se borrará y será reemplazada con el nuevo archivo. Si el archivo no existe, se creará. Cuando se redirecciona la salida a un archivo, también se puede usar el símbolo >>. Si el archivo de destino especificado ya existe, la salida del programa se añadirá al final del archivo.

El programa del listado 14.15 demuestra la redirección. Este programa acepta una línea de entrada a partir de `stdin`, y luego envía la línea a `stdout`, precedida con la leyenda `The input was:`. Después de compilar y enlazar el programa, ejecútelo sin redirección (suponiendo que el programa se llama LIST1415) tecleando LIST1415 en la línea de comandos del DOS. Si luego teclea `Estoy aprendiendo C por mí mismo`, el programa despliega

```
the input was: Estoy aprendiendo C por mí mismo
```

en la pantalla. Si se ejecuta el programa tecleando `LIST1415 >test.txt` y se teclea lo mismo, no se despliega nada en la pantalla. En vez de ello, se crea en disco un archivo llamado `test.txt`. Use el comando TYPE del DOS (o uno equivalente) para desplegar el contenido del archivo:

```
type test.txt
```

y verá que el archivo contiene la única línea `The input was: Estoy aprendiendo C por mí mismo`. De manera similar, si se ha ejecutado el programa tecleando

```
LIST1415 >prn
```

la línea de salida habrá sido impresa en la impresora. Ejecute el programa nuevamente, pero esta vez redireccione la salida a TEST.TXT con el símbolo `>>`. En vez de reemplazar el archivo, la nueva salida se añade al final de TEXT.TXT.

Captura

Listado 14.15. Programa para demostrar la redirección de entrada y salida.

```
1: /* Puede usarse para demostrar la redirección de stdin y stdout. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:    char buf[80];
8:
9:    gets(buf);
10:   printf("The input was: %s", buf);
11: }
```

Ahora veamos la redirección de la entrada. En primer lugar, se necesita un archivo fuente. Use un editor para crear un archivo que se llame `input.txt` que contenga la única línea `William Shakespeare`. Ahora ejecute el programa anterior, tecleando lo siguiente en la línea de comandos del DOS:

```
list1415 <input.txt
```

El programa no espera a que se le dé ninguna entrada por el teclado. En vez de ello, inmediatamente despliega el mensaje

```
The input was: William Shakespeare
```

en la pantalla. El flujo `stdin` fue redireccionado al archivo de disco `input.txt`, por lo que la llamada del programa a `gets()` lee una línea de texto del archivo, en vez de hacerlo del teclado.

Se puede redirigir la entrada y la salida al mismo tiempo. Haga la prueba ejecutando el programa con el comando

```
list1415 <input.txt >junk.txt
```

para redirigir `stdin` al archivo `input.txt` y redirigir a `stdout` a `junk.txt`.

La redirección de `stdin` y `stdout` puede ser útil en determinadas circunstancias. Por ejemplo, un programa de ordenamiento puede clasificar la entrada del teclado o el contenido de un archivo de disco. De la misma forma, un programa de listas de correo puede desplegar las direcciones en la pantalla, enviarlas a la impresora para imprimir etiquetas o ponerlas en un archivo para algún otro uso.

No olvide que la redirección de `stdin` y `stdout` es una característica del sistema operativo y no del lenguaje C. Sin embargo, proporciona otro ejemplo de la flexibilidad de los flujos.

Cuándo usar *fprintf()*

Como se dijo anteriormente, la función de biblioteca `fprintf()` es idéntica a `printf()`, a excepción de que se puede especificar el flujo al que se envía la salida. El uso principal de `fprintf()` se da con archivos de disco, tratados en el Día 16, "Uso de archivos de disco". Hay otros dos usos, como se explica aquí.

Uso de *stderr*

Uno de los flujos predefinidos del C es `stderr`, error estándar. Tradicionalmente los mensajes de error de un programa son enviados al flujo `stderr` y no a `stdout`. ¿A qué se debe esto?

Como ha aprendido, la salida de `stdout` puede ser redirigida a un destino que no sea la pantalla. Si `stdout` es redireccionado, tal vez el usuario no se dé cuenta de cualquier mensaje de error que el programa envíe a `stdout`. A diferencia de `stdout`, `stderr` no puede ser redirigido, y siempre está conectado a la pantalla. Dirigiendo los mensajes de error a `stderr`, se puede estar seguro de que el usuario siempre los ve. Se puede lograr esto con `fprintf()`:

```
fprintf(stderr, "Ha habido un error. ");
```

Se puede escribir una función para manejar los mensajes de error, y luego llamar a la función cuando sucede el error, en vez de llamar a `fprintf()`:

```
mensaje_de_error( "Ha ocurrido un error. ");

void mensaje_de_error(char *msg)
{
   fprintf(stderr, msg);
}
```

Usando su propia función, en vez de llamar directamente a `fprintf()`, se proporciona flexibilidad adicional (una de las ventajas de la programación estructurada). Por ejemplo, en circunstancias especiales tal vez quiera que los mensajes de error de un programa vayan a la impresora o a un archivo de disco. Todo lo que necesita hacer es modificar la función `mensaje_de_error()` para que la salida se envíe al destino deseado.

Salida a impresora

Se envía salida a la impresora accesando el flujo predefinido `stdprn`. En las PC de IBM y compatibles el flujo `stdprn` está conectado al dispositivo LPT1: (el primer puerto paralelo). El listado 14.16 presenta un ejemplo simple.

Captura

Listado 14.16. Envío de salida a la impresora.

```
1: /* Demuestra la salida a impresora. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     float f = 2.0134;
8:
9:     fprintf(stdprn, "This message is printed.\n\n");
10:    fprintf(stdprn, "And now some numbers:\n\n");
11:    fprintf(stdprn, "The square of %f is %f.", f, f*f);
12:
13:    /* Envía un avance de forma. */
14:
15:    fprintf(stdprn, "\f");
16: }
```

Salida

```
This message is printed.
And now some numbers:
The square of 2.013400 is 4.053780.
```

Nota: La salida anterior es impresa es la impresora. Esta salida no aparecerá en la pantalla.

Análisis Si su sistema tiene una impresora conectada al puerto LPT1: puede compilar y ejecutar el programa del listado 14.16. Imprime tres líneas en la página. La línea 15 envía un `"\f"` a la impresora. `\f` es la secuencia de escape para un avance de forma, el comando que hace que la impresora avance una página (o, en el caso de una impresora láser, que saque la página de trabajo).

DEBE	NO DEBE

NO DEBE Tratar de redirigir a `stderr`.

DEBE Usar fprintf() para crear programas que puedan imprimir `stdout`, `stderr`, `stdprn` o cualquier otro flujo.

DEBE Usar `fprintf()` con `stderr` para imprimir mensajes de error en la pantalla.

NO DEBE Usar `stderr` para propósitos diferentes de la impresión de mensajes de error o avisos.

DEBE Crear funciones, como `mensaje_de_error`, para hacer su código más estructurado y más mantenible.

Resumen

Este fue un largo día, lleno de información importante sobre la entrada y salida de los programas. Usted aprendió la manera en que el C usa los flujos, tratando toda la entrada y salida como una secuencia de caracteres. También aprendió que el C tiene cinco flujos predefinidos, `stdin` (el teclado), `stdout` (la pantalla), `stderr` (la pantalla), `stdprn` (la impresora) y `stdaux` (el puerto de comunicaciones).

La entrada del teclado llega del flujo `stdin`. Con las funciones de biblioteca estándares del C se puede aceptar la entrada del teclado carácter por carácter, línea por línea o como números formateados y cadenas. La entrada de caracteres puede ser con almacenamiento temporal o sin él, con réplica o sin ella.

La salida a la pantalla se hace normalmente con el flujo `stdout`. De manera similar a la entrada, la salida del programa puede ser por carácter, por línea o como números formateados y cadenas. Para salida a la impresora se usa `fprintf()`, para enviar datos al flujo `stdprn`.

Cuando se usa `stdin` y `stdout` se puede redirigir la entrada y salida del programa. La entrada puede venir de un archivo de disco, en vez del teclado, y la salida puede ir a un archivo de disco o a la impresora, en vez de a la pantalla.

Por último, aprendió por qué los mensajes de error deben ser enviados al flujo `stderr`, en vez de a `stdout`. Debido a que `stderr` siempre está conectado a la pantalla, se tiene la seguridad de ver los mensajes de error, incluso cuando la salida del programa esté redireccionada.

Preguntas y respuestas

1. ¿Qué pasa si envío la salida a un dispositivo de entrada?

   Se puede escribir un programa en C que haga esto, ¡sin embargo no funcionará! Por ejemplo, si trata de usar `stdprn` con `fscanf()`, el programa compila en un archivo ejecutable, pero la impresora es incapaz de enviar entrada, por lo que el programa no funciona como se pretende.

2. ¿Qué pasa si redirijo alguno de los flujos estándares?

   Si se redirige alguno de los flujos estándares, puede causar problemas posteriormente en el programa. Si se redirige un flujo, se le debe regresar si lo necesita nuevamente en el mismo programa. Muchas de las funciones descritas en este capítulo usan los flujos estándares. Todas ellas usan los mismos flujos, por lo que si cambia al flujo en un lugar, lo cambia para todas las funciones. Por ejemplo, asigne `stdout` igual a `stdprn` en uno de los listados del capítulo ¡y vea lo que pasa!

3. ¿Hay algún peligro en usar funciones no ANSI en un programa?

   La mayoría de los compiladores vienen con muchas funciones útiles que no son del estándar ANSI. Si planea usar siempre ese compilador y no transportar el código a otros compiladores o plataformas, no hay problema. Cuando vaya a usar otros compiladores y plataformas, debe preocuparse con la compatibilidad ANSI.

4. ¿Por qué no puedo usar siempre `fprintf()`, en vez de `printf()`? (¿O `fscanf()`, en vez de `scanf()`?)

Si se están usando los flujos de entrada y salida estándares, se debe usar `printf()` y `scanf()`. Con estas funciones más simples no tiene por qué preocuparse con los otros flujos.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Qué es un flujo y para qué usan los flujos un programa en C?
2. Los siguientes dispositivos ¿son de entrada o de salida?
   a. Impresora
   b. Teclado
   c. Modem
   d. Monitor
   e. Unidad de disco
3. Liste los cinco flujos predefinidos y los dispositivos con los cuales están asociados.
4. ¿Qué flujos usan las siguientes funciones?
   a. `printf()` o `puts()`
   b. `scanf()` o `gets()`
   c. `fprintf()`
5. ¿Cuál es la diferencia entre entrada de caracteres con almacenamiento temporal y sin él desde `stdin`?
6. ¿Cuál es la diferencia entre entrada de caracteres replicada y sin replicar desde `stdin`?
7. ¿Se puede "desobtener" más de un carácter a la vez con `ungetc()`? ¿Se puede "desobtener" el carácter `EOF`?
8. Cuando se usan las funciones de entrada de línea del C, ¿cómo se determina el fin de línea?

9. ¿Cuáles de los siguientes son especificadores de tipo válidos?

   a. `"%d"`

   b. `"%4d"`

   c. `"%3i%c"`

   d. `"%q%d"`

   e. `"%%%i"`

   f. `"%9ld"`

10. ¿Cuál es la diferencia entre `stderr` y `stdout`?

Ejercicios

1. Escriba un enunciado que imprima `"Hello World"` en la pantalla.
2. Use dos funciones diferentes del C para hacer lo mismo que hicieron las funciones en el ejercicio 1.
3. Escriba un enunciado que imprima `"Hello Auxiliary Port"` en el puerto auxiliar estándar.
4. Escriba un enunciado que obtenga una cadena de 30 caracteres o menos. Si se encuentra un asterisco, trunque la cadena.
5. Escriba un solo enunciado que imprima lo siguiente:

   ```
   Juan preguntó "¿Qué es una diagonal invertida?"

   Pedro dijo, "Es '\'"
   ```

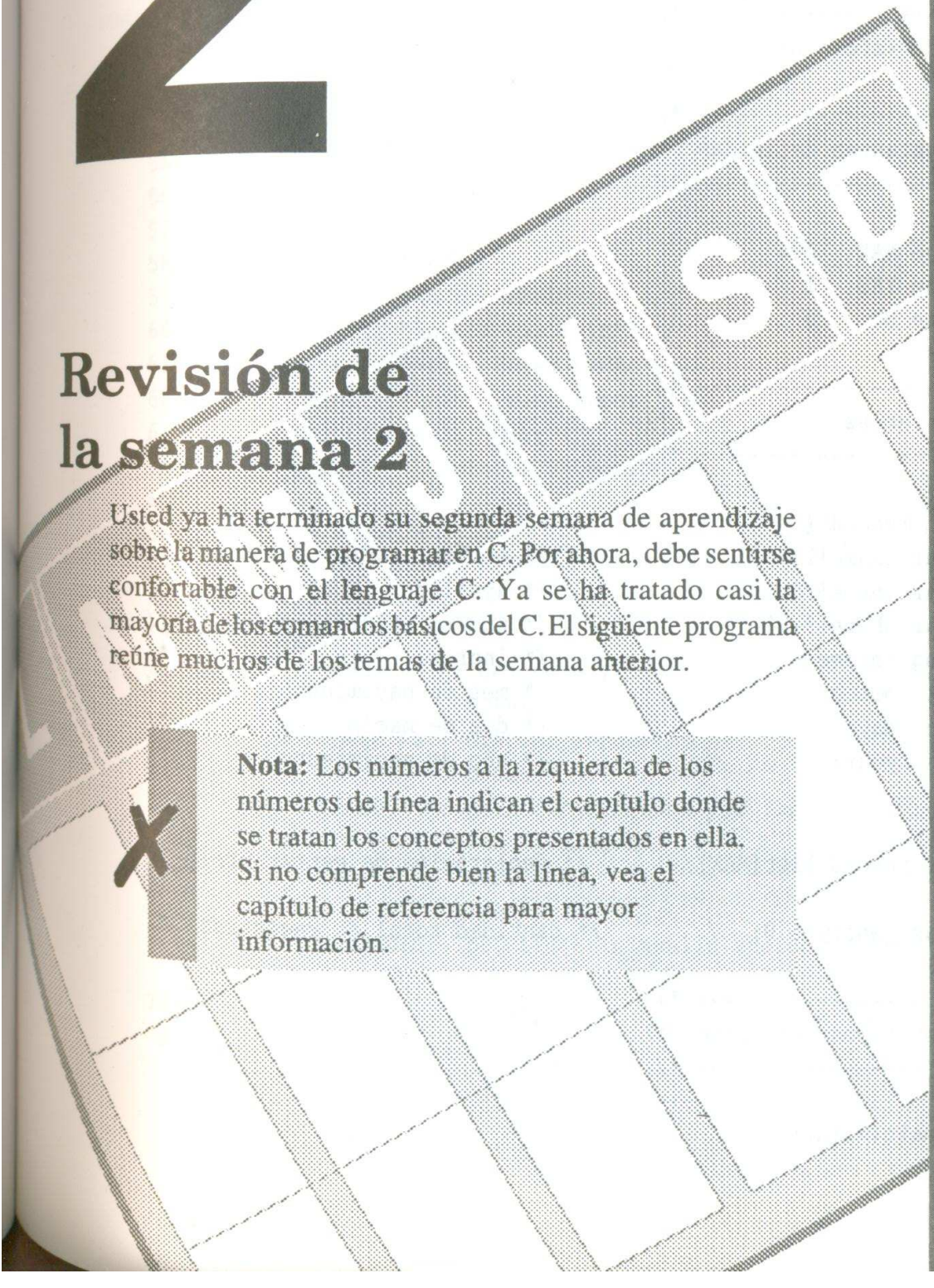
   Debido a multitud de posibilidades, no se proporcionan respuestas para los siguientes ejercicios; sin embargo, debe tratar de hacerlos.

6. Escriba un programa que redireccione un archivo a la impresora carácter por carácter.
7. Escriba un programa que use redirección para aceptar entrada de un archivo de disco, cuente la cantidad de veces que aparece cada carácter ASCII en el archivo y luego despliegue los resultados en la pantalla. (Se da una pista en el apéndice G, "Respuestas".)
8. Escriba un programa que imprima archivos fuente de C. Use redirección para dar el nombre del archivo y use `fprintf()` para hacer la impresión.

9. Escriba un programa de "tecleo" que acepte entrada del teclado, la replique en la pantalla y luego reproduzca esta entrada en la impresora. El programa debe contar las líneas y avanzar el papel en la impresora a una nueva página cuando sea necesario. Use una tecla de función para terminar el programa.

10. Modifique el programa del ejercicio 8 para poner números de línea al inicio del listado cuando lo imprima. (Se da una pista en el apéndice G, "Respuestas".)

2



Revisión de la semana 2

Usted ya ha terminado su segunda semana de aprendizaje sobre la manera de programar en C. Por ahora, debe sentirse confortable con el lenguaje C. Ya se ha tratado casi la mayoría de los comandos básicos del C. El siguiente programa reúne muchos de los temas de la semana anterior.

Nota: Los números a la izquierda de los números de línea indican el capítulo donde se tratan los conceptos presentados en ella. Si no comprende bien la línea, vea el capítulo de referencia para mayor información.

Listado R2.1. Listado de la revisión de la semana dos.

```
1: /* Nombre de programa: week2.c                                    */
2: /* programa para dar información de hasta 100 personas.          */
3: /* El programa imprime un informe                                 */
4: /* basándose en las cantidades tecleadas.                         */
5: /*---------------------------------------------------------------*/
6:  /*--------------------- */
7: /* archivos de inclusión  */
8: /*---------------------- */
9: #include <stdio.h>
10:
11: /*--------------------- */
12: /* constantes definidas  */
13: /*--------------------- */
14:
15: #define MAX   100
16: #define YES   1
17: #define NO    0
18:
19: /*--------------------- */
20: /* variables             */
21: /*--------------------- */
22:
23: struct record {
24:    char fname[15+1];              /* nombre + NULL                */
25:    char lname[20+1];              /* apellido + NULL              */
26:    char phone[9+1];               /* número telefónico + NULL     */
27:    long income;                   /* ingreso                      */
28:    int  month;                    /* mes de nacimiento            */
29:    int  day;                      /* día de nacimiento            */
30:    int  year;                     /* año de nacimiento            */
31: };
32:
33: struct record list[MAX];        /* declaración de estructura     */
34:
35: int last_entry = 0;             /* cantidad total de registros   */
36:
37: /*--------------------- */
38: /* prototipos de función */
39: /*--------------------- */
40:
41: void main(void);
```

```
42: void get_data(void);
43: void display_report(void);
44: int  continue_function(void);
45: void clear_kb(void);
46:
47: /*--------------------- */
48: /* inicio de programa    */
49: /*--------------------- */
50:
51: void main()
52: {
53:     int cont = YES;
54:     int ch;
55:
56:     while( cont == YES )
57:     {
58:        printf( "\n");
59:        printf( "\n     MENU");
60:        printf( "\n   ========\n");
61:        printf( "\n1.  Enter names");
62:        printf( "\n2.  Print report");
63:        printf( "\n3.  Quit");
64:        printf( "\n\nEnter Selection ==> ");
65:
66:        ch = getch();
67:
68:        switch( ch )
69:        {
70:           case _'1': get_data();
71:                      break;
72:           case '2': display_report();
73:                      break;
74:           case '3': printf("\n\nThank you for using this \
                              program!");
75:                      cont = NO;
76:                      break;
77:           default:  printf("\n\nInvalid choice, Please select \
                              1 to 3!");
78:                      break;
79:        }
```

Listado R2.1. continuación

```
80:     }
81: }
82:
83: /*-------------------------------------------------------------------- *
84:  * Función: get_data()                                                 *
85:  * Objetivo: esta función obtiene datos del usuario.                   *
86:  *         continúa recibiendo los datos hasta que se dan              *
87:  *         los de 100 personas o cuando el usuario decide no continuar. *
88:  * Regresa: nada                                                       *
89:  * Notas: esto le permite que se teclee 0/0/0 para las fechas          *
90:  *        de nacimiento en caso de que el usuario no esté seguro.      *
91:  *        También acepta meses de 31 días.                             *
92:  *--------------------------------------------------------------------  */
93:
94: void get_data(void)
95: {
96:    int cont;
97:    int ctr;
98:
99:    for ( cont = YES; last_entry < MAX && cont == YES;last_entry++ )
100:    {
101:       printf("\n\nEnter information for Person %d.",last_entry+1 );
102:
103:       printf("\n\nEnter first name: ");
104:       gets(list[last_entry].fname);
105:
106:       printf("\nEnter last name: ");
107:       gets(list[last_entry].lname);
108:
109:       printf("\nEnter phone in 123-4567 format: ");
110:       gets(list[last_entry].phone);
111:
112:       printf("\nEnter Yearly Income (whole dollars): ");
113:       scanf("%ld", &list[last_entry].income);
114:
115:       printf("\nEnter Birthday:");
116:
117:       do
```

```
118:        {
119:            printf("\n\tMonth (0 - 12): ");
120:            scanf("%d", &list[last_entry].month);
121:        }while ( list[last_entry].month < 0 ||
122:                 list[last_entry].month > 12 );
123:
124:        do
125:        {
126:            printf("\n\tDay (0 - 31): ");
127:            scanf("%d", &list[last_entry].day);
128:        }while ( list[last_entry].day <  0 ||
129:                 list[last_entry].day > 31 );
130:
131:        do
132:        {
133:            printf("\n\tYear (1800 - 1996): ");
134:            scanf("%d", &list[last_entry].year);
135:        }while (list[last_entry].year != 0 &&
136:                (list[last_entry].year < 1800 ||
137:                 list[last_entry].year > 1996 ));
138:
139:        cont = continue_function();
140:    }
141:
142:    if( last_entry == MAX)
143:       printf("\n\nMaximum Number of Names has been \
                 entered!\n");
144: }
145: /*-------------------------------------------------------------------- *
146:  * Función: display_report()                                           *
147:  * Objetivo: esta función despliega un informe en la pantalla         *
148:  * Regresa: nada                                                      *
149:  * Notas: Se puede desplegar más información.                         *
150:  *        Cambie stdout a stdprn para imprimir un informe             *
151:  *-------------------------------------------------------------------- */
152:
153: void display_report()
154: {
155:     long   month_total = 0,
156:            grand_total = 0;       /* Para totales */
157:     int    x, y;
```

Listado R2.1. continuación

```
158:
159:     fprintf(stdout, "\n\n");       /* salta unas cuantas líneas */
160:     fprintf(stdout, "\n             REPORT");
161:     fprintf(stdout, "\n            ========");
162:
163:     for( x = 0; x <= 12; x++ )    /* para cada mes, incluyendo el 0 */
164:     {
165:        month_total = 0;
166:        for( y = 0; y < last_entry; y++ )
167:        {
168:           if( list[y].month == x )
169:           {
170:               fprintf(stdout,"\n\t%s %s %s %ld",
                              list[y].fname,
171:                             list[y].lname, list[y].phone,
                              list[y].income);
172:               month_total += list[y].income;
173:           }
174:        }
175:        fprintf(stdout, "\nTotal for month %d is %ld",
                    x,month_total);
176:        grand_total += month_total;
177:     }
178:     fprintf(stdout, "\n\nReport totals:");
179:     fprintf(stdout, "\nTotal Income is %ld", grand_total);
180:     fprintf(stdout, "\nAverage Income is %ld", grand_total/
                last_entry );
181:
182:     fprintf(stdout, "\n\n* * * End of Report * * *");
183: }
184: /*-------------------------------------------------------------- *
185:  * Función: continue_function()                                  *
186:  * Objetivo: esta función le pregunta al usuario si quiere continuar. *
187:  * Regresa:  YES - Si el usuario desea continuar                 */
188:  *           NO  - Si el usuario desea terminar                  */
189:  *-------------------------------------------------------------- */
190:
191: int continue_function( void )
192: {
```

```
193:     char ch;
194:
195:     printf("\n\nDo you wish to continue? (Y)es/(N)o: ");
196:     ch = getch();
197:
198:     while( ch != 'n' && ch != 'N' && ch != 'y' && ch != 'Y' )
199:     {
200:         printf("\n%c is invalid!", ch);
201:         printf(" n\nPlease enter \'N\' to Quit or \'Y\' to \
                 Continue: ");
202:         ch = getch();
203:     }
204:
205:     clear_kb();
206:
207:     if(ch == 'n' || ch == 'N')
208:         return(NO);
209:     else
210:         return(YES);
211: }
212:
213: /*-------------------------------------------------------   *
214:  * Función:  clear_kb()                                     *
215:  * Objetivo: esta función limpia el teclado de caracteres
   adicionales.                                                 *
216:  * Regresa: nada                                           */
217:  *-------------------------------------------------------   */
218: void clear_kb(void)
219: {
220:     char junk[80];
221:     gets(junk);
222: }
```

Pareciera que, conforme aprende el C, los programas se hacen más grandes. Este programa se parece al que fue presentado después de la primera semana de programación en C, pero cambia algunas de las tareas y añade unas cuantas más. De manera similar al de la revisión de la semana uno, puede teclear hasta 100 conjuntos de información. Los datos que se dan es información acerca de gente. Debe observar que este programa puede desplegar el informe mientras se da la información. Con el otro programa no se podía imprimir el informe sino hasta que se hubiera terminado de dar los datos.

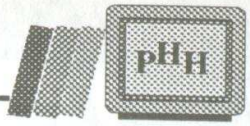
También debe observar la adición de la estructura empleada para guardar los datos. La estructura se define en las líneas 23 a 31 de este programa. Frecuentemente las estructuras se usan para agrupar datos similares (Día 11, "Estructuras"). Este programa agrupa todos los datos para cada persona en una estructura llamada record. Muchos de estos datos deben serle familiares; sin embargo, se registran unos cuantos conceptos nuevos. Las líneas 24 a 26 contienen tres arreglos, o cadenas, de caracteres para guardar el nombre, el apellido y el número de teléfono. Observe que cada una de estas cadenas es declarada con un `+1` en su tamaño de arreglo. Debe recordar del Día 10, "Caracteres y cadenas", que este lugar adicional guarda el carácter `nulo` que significa el final de la cadena.

Este programa demuestra el uso adecuado del alcance de variables (Día 12, "Alcance de las variables"). Las líneas 33 a 35 contienen dos variables globales. La línea 35 usa un `int`, llamado `last_entry`, para guardar la cantidad de gente que ha sido tecleada. Esta es similar a la variable `ctr` usada en la revisión del final de la semana uno. La otra variable global es `list[MAX]`, que es un arreglo de estructuras de registro. Las variables locales se usan en cada una de las funciones a lo largo del programa. De especial interés son las variables `month_total`, `grand_total`, `x` y `y`, en las líneas 155 a 157 en `display_report()`. En la revisión de la semana uno éstas eran variables globales. Debido a que estas variables se refieren solamente a `display_report()`, quedan mejor situadas como variables locales.

Un enunciado adicional de control de programa, el enunciado `switch` (Día 13, "Más sobre el control de programa") se usa en las líneas 68-79. Al usar un enunciado `switch`, en vez de varios enunciados `if...else`, se facilita seguir el código. Las líneas 70 a 79 ejecutan diversas tareas basadas en una selección de menú. Observe que también se incluye el enunciado `default`, para el caso de que se dé un valor que no sea una opción de menú válida.

En la función `get_data()` deberá observar que hay algunos cambios adicionales con respecto a la revisión de la semana uno. Las líneas 103 y 104 le piden una cadena. La línea 104 usa la función `gets()` (Día 14, "Trabajando con la pantalla, la impresora y el teclado") para recibir el nombre de la persona. La función `gets()` recibe una cadena y pone el valor en `list[last_entry].fname`. Debe recordar del Día 11, "Estructuras", que esto pone al nombre en `fname`, que es un miembro de la lista de la estructura.

`display_report()` ha sido modificada, usando `fprintf()` en vez de `printf()` para desplegar la información. La razón de este cambio es simple. Si se quiere que el informe vaya a la impresora en vez de a la pantalla en cada enunciado `fprintf()`, cambie `stdout` a `stdprn`. En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se trató `fprintf()`, `stdout` y `stdprn`. Recuerde que `stdout` y `stdprn` son flujos que envían la salida a la pantalla y a la impresora respectivamente.



`continue_function()`, que se encuentra en las líneas 191 a 211, también ha sido modificada. Ahora se responde a la pregunta con `Y` o `N` en vez de 0 o 1. Esto es más amigable. También observe que la función `clear_kb()` del listado 13.9 ha sido añadida a la línea 205, para quitar cualesquier caracteres adicionales que haya tecleado el usuario.

Este programa usa lo que ha aprendido en las dos primeras semanas de autoprendizaje del C. Como puede ver, muchos de los conceptos de la segunda semana hacen que los programas en C sean más funcionales, y la codificación en C, más fácil. La semana tres continúa avanzando sobre estos conceptos.

3 SEMANA

UN VISTAZO

15

16

17

18

19

20

21

Usted ya ha terminado su segunda semana de aprendizaje sobre cómo programar en C. Por ahora debe sentirse a gusto con el lenguaje C, habiendo tocado la mayoría de las áreas del lenguaje.

Adónde vamos...

En la siguiente semana verá lo que le hemos llamado "Cómo obtener lo máximo del C". Se tocarán muchos de los temas de la primera y segunda semanas y se explorará lo que ellos pueden hacer por usted.

La semana final cierra con broche de oro su aprendizaje del C en 21 días. Cuando termine esta semana final, usted ya debe saber C. El día 15, "Más sobre apuntadores", aborda una de las partes más difíciles del C: apuntadores avanzados. De manera similar al Día 9, "Apuntadores", se debe dedicar un poco de tiempo adicional a los temas tratados. El Día 16, "Uso de archivos de disco", trata uno de los temas más útiles para la creación de aplicaciones: los archivos de disco. Aprenderá la manera de usar archivos de disco para el almacenamiento y la recuperación de datos.

Los Días 17, "Manipulación de cadenas", 18, "Cómo obtener más de las funciones", y 19, "Exploración de la biblioteca de funciones", lo bombardean con una multitud de funciones de biblioteca. Los apéndices E, "Prototipos de función y archivos de encabezado" y F, "Funciones comunes en orden alfabético", listan muchas de las funciones ANSI comunes. Los últimos dos días tratan otras funciones del C, explicando cosas como la manipulación de bits y las directivas del preprocesador.

DIA 15

Más sobre apuntadores

En el Día 9, "Apuntadores", se le presentó lo básico de los apuntadores, una parte importante del lenguaje de programación C. Hoy irá más adelante, explorando algunos temas avanzados sobre los apuntadores que pueden añadir flexibilidad a la programación. Usted aprenderá hoy:

- Cómo declarar un apuntador a un apuntador.
- Cómo usar apuntadores con arreglos multidimensionales.
- Cómo declarar arreglos de apuntadores.
- Cómo declarar apuntadores a funciones.

Apuntadores a apuntadores

Tal como aprendió en el Día 9, "Apuntadores", un apuntador es una variable numérica que contiene un valor que es la dirección de otra variable. Se declara a un apuntador usando al operador de indirección (*). Por ejemplo, la declaración

```
int *ptr;
```

declara un apuntador, llamado `ptr`, que puede apuntar a una variable tipo `int`. Luego se usa el operador de *dirección de* (&) para hacer que el apuntador apunte a una variable específica del tipo correspondiente. Suponiendo que `x` ha sido declarada como variable de tipo `int`, el enunciado

```
ptr = &x;
```

asigna la *dirección de* `x` a `ptr` y hace que `ptr` apunte a `x`. Nuevamente, usando el operador de indirección se puede accesar la variable apuntada usando su apuntador. Ambos de los siguientes enunciados asignan el valor de `12` a `x`:

```
x = 12;
*ptr = 12;
```

Debido a que un apuntador es en sí mismo una variable numérica, es guardada en la memoria de la computadora en una dirección particular. Por lo tanto, se puede crear un apuntador a un apuntador, una variable cuyo valor es la dirección de un apuntador. Esta es la manera:

```
int x = 12;                          /* x es una variable tipo int      */
int *ptr = &x;                       /* ptr es un apuntador a x         */
int **ptr_a_ptr = &ptr;              /* ptr_a_ptr es un apuntador a     */
                                     /* un apuntador a tipo int.        */
```

Observe que se usa un operador de indirección doble (`**`) cuando se declara el apuntador a apuntador. También puede usar al operador de indirección doble cuando accese la variable apuntada con un apuntador a apuntador. Por lo tanto, el enunciado

```
**ptr_a_ptr = 12;
```

asigna el valor `12` a la variable `x`, y el enunciado

```
printf( %d,  **ptr_a_ptr);
```

despliega el valor de `x` en la pantalla. Si por error se usa un operador de indirección sencillo, se obtienen errores. El enunciado

```
*ptr_a_ptr = 12;
```

asigna el valor de `12` a `ptr`, lo que da como resultado que `ptr` apunte a lo que pueda estar guardado en la dirección `12`. Esto evidentemente es un error.

Cuando se declara y usa un apuntador a apuntador, es llamado indirección múltiple. Las relaciones entre una variable, un apuntador y un apuntador a apuntador son ilustradas en la figura 15.1. No hay límite en realidad al nivel posible de indirección múltiple, se puede tener un apuntador a apuntador a apuntador y *hasta el infinito,* pero por lo general no hay gran ventaja de ir más allá de dos niveles.

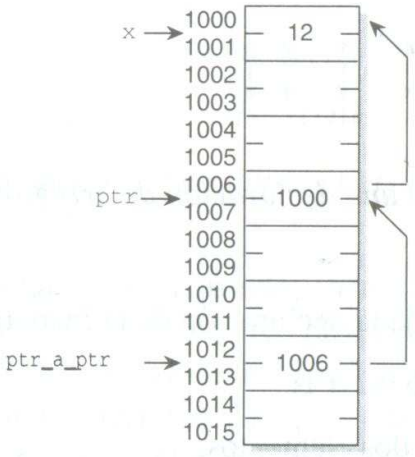


Figura 15.1. *Ilustración de un apuntador a apuntador.*

¿Para qué pueden usar los apuntadores a apuntadores? El uso más común involucra arreglos de apuntadores, que son tratados posteriormente en este capítulo. El Día 19, "Exploración de la biblioteca de funciones", presenta un ejemplo del uso de indirección múltiple en el listado 19.5.

Apuntadores y arreglos de varias dimensiones

El Día 8, "Arreglos numéricos", trata las relaciones especiales entre apuntadores y arreglos. Específicamente, el nombre de un arreglo sin su par de corchetes es un apuntador al primer elemento del arreglo. Por consecuencia, es más fácil usar notación de apuntadores cuando se accesan determinados tipos de arreglos. Sin embargo, en estos primeros ejemplos nos

hemos limitado a arreglos de una sola dimensión. ¿Qué hay acerca de los arreglos multidimensionales?

Recuerde que un arreglo multidimensional es declarado con un juego de corchetes para cada dimensión. Por ejemplo, el enunciado

```
int multi[2][4];
```

declara un arreglo de dos dimensiones que contiene ocho variables tipo `int`. Se puede visualizar al arreglo como si tuviera una estructura de renglones y columnas, en este caso dos renglones y cuatro columnas. Sin embargo, hay otra manera de visualizar un arreglo multidimensional, y una que es más próxima a la manera en que el C maneja, de hecho, los arreglos. Se puede considerar a `multi` como un arreglo de dos elementos, donde cada uno de sus dos elementos es un arreglo de cuatro enteros.

Tal vez esto no sea muy claro para usted. La figura 15.2 diseca el enunciado de declaración de arreglo en sus partes componentes.

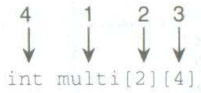


Figura 15.2. *Los componentes de una declaración de arreglo multidimensional.*

Se interpreta a los componentes de la declaración de la manera siguiente:

1. Declara un arreglo llamado `multi`.
2. El arreglo `multi` contiene dos elementos.
3. Cada uno de esos dos elementos contiene cuatro elementos.
4. Cada uno de esos cuatro elementos es de tipo `int`.

Una declaración de arreglo multidimensional se lee comenzando con el nombre del arreglo y moviéndose hacia la derecha por cada juego de corchetes. Cuando se llega al último juego de corchetes (la última dimensión) se toma en consideración el tipo de dato básico del arreglo que se encuentra a la izquierda del nombre.

Bajo el esquema de arreglo de arreglo se visualiza un arreglo multidimensional, tal como se muestra en la figura 15.3.

Ahora, regresemos al tema de los nombres de arreglo como apuntadores. (¡Después de todo éste es un capítulo acerca de apuntadores!) De manera similar a los arreglos de una dimensión, el nombre de un arreglo multidimensional es un apuntador al primer elemento de un arreglo. Continuando con el ejemplo, `multi` es un apuntador al primer elemento del arreglo de dos dimensiones que fue declarado como `int multi[2][4]`. ¿Qué es

exactamente el primer elemento de `multi`? No es la variable tipo `int multi[0][0]` como puede pensar. Recuerde que `multi` es un arreglo de arreglos, por lo que su primer elemento es `multi[0]`, que es un arreglo de cuatro variables tipo `int` (uno de los dos arreglos contenido en `multi`).

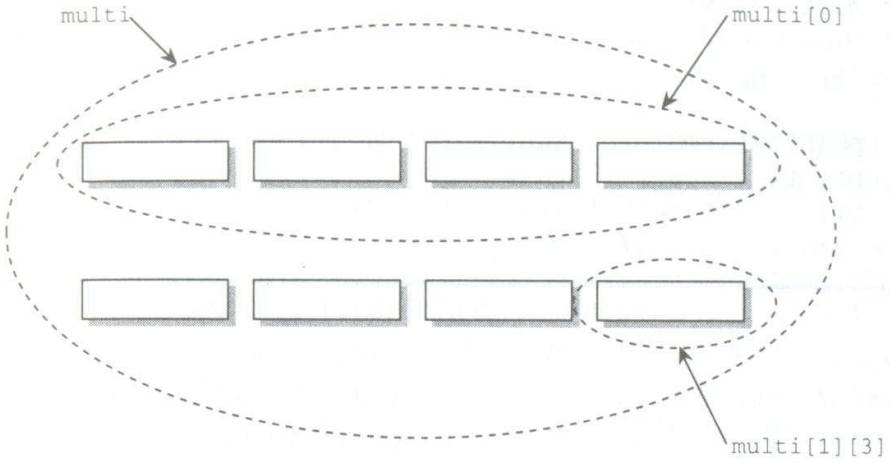


Figura 15.3. *Un arreglo de dos dimensiones es visualizado como un arreglo de arreglos.*

Ahora, si `multi[0]` es también un arreglo, ¿apunta él a algo? ¡Claro que sí! `multi[0]` apunta a su primer elemento, `multi[0][0]`. Tal vez le extrañe el porqué `multi[0]` es un apuntador. Recuerde que el nombre de un arreglo sin corchetes es un apuntador al primer elemento del arreglo. Bien, el término `multi[0]` es el nombre del arreglo `multi[0][0]` al que le falta el último par de corchetes, y por lo tanto califica como un apuntador.

Si está un poco confundido en este momento no se preocupe. Este material es difícil de entender. Tal vez le ayude si recuerda las siguientes reglas para un arreglo de *n* dimensiones:

- El nombre del arreglo seguido por *n* pares de corchetes (conteniendo cada par un índice adecuado, por supuesto) evalúa como dato de un arreglo (esto es, los datos guardados en el elemento de arreglo especificado).
- El nombre del arreglo seguido por menos de *n* pares de corchetes evalúa como un apuntador a un elemento de arreglo.

Por lo tanto, en el ejemplo `multi` evalúa como un apuntador, `multi[0]` evalúa como un apuntador y `multi[0][0]` evalúa como dato de un arreglo.

Ahora veamos dónde apuntan de hecho, todos estos apuntadores. El programa del listado 15.1 declara un arreglo de dos dimensiones, similar al que se ha estado usando en los ejemplos, imprime los apuntadores que hemos estado tratando y también imprime la dirección del primer elemento del arreglo.

Captura

Listado 15.1. La relación entre un arreglo multidimensional y apuntadores.

```
1:   /* Demuestra los apuntadores y arreglos multidimensionales.*/
2:
3:   #include <stdio.h>
4:
5:   int multi[2][4];
6:
7:   main()
8:   {
9:       printf("\nmulti = %u", multi);
10:      printf("\nmulti[0] = %u", multi[0]);
11:      printf("\n&multi[0][0] = %u", &multi[0][0]);
12:  }
```

Salida

```
multi = 1328
multi[0] = 1328
&multi[0][0] = 1328
```

Análisis

El valor actual puede ser que no sea 1328 en su sistema, pero los tres valores deben de ser el mismo. La dirección del arreglo `multi` es la misma que la dirección del arreglo `multi[0]` y ambas son iguales a la dirección del primer entero del arreglo `multi[0][0]`.

Si estos apuntadores tienen el mismo valor, ¿cuál es la diferencia práctica entre ellos en términos del programa? Recuerde del Día 9, "Apuntadores", que el compilador C "sabe" a lo que apunta un apuntador. Para ser más exactos, el compilador sabe el tamaño del concepto al cual está apuntando un apuntador.

¿Cuáles son los tamaños de los elementos que se han estado usando? El listado 15.2 usa el operador `sizeof()` para desplegar los tamaños en bytes de estos elementos.

Captura

Listado 15.2. Determinación del tamaño de los elementos.

```
1: /* Demuestra los tamaños de elementos de arreglos multidimensionales. */
2:
3: #include <stdio.h>
4:#
5: int multi[2][4];
6:
7: main()
8: {
9:     printf("\nThe size of multi = %u", sizeof(multi));
10:      printf("\nThe size of multi[0] = %u", sizeof(multi[0]));
11:      printf("\nThe size of multi[0][0] = %u",
                sizeof(multi[0][0]));
12: }
```

Salida

La salida de este programa (suponiendo que el compilador usa enteros de dos bytes) es la siguiente:

```
The size of multi = 16
The size of multi[0] = 8
The size of multi[0][0] = 2
```

Análisis

Piense acerca de estos valores de tamaño. El arreglo `multi` contiene dos arreglos, y cada uno de ellos contiene cuatro enteros. Cada entero requiere dos bytes de almacenamiento. Con un total de ocho enteros el tamaño de 16 bytes tiene sentido.

A continuación `multi[0]` es un arreglo que contiene cuatro enteros, cada entero ocupa dos bytes, por lo que el tamaño de ocho bytes de `multi[0]` también tiene sentido.

Por último, `multi[0][0]` es un entero, por lo que su tamaño es, por supuesto, de dos bytes.

Ahora, sin olvidar estos tamaños, recuerde la discusión del Día 9, "Apuntadores", sobre la aritmética de apuntadores. El compilador C "sabe" el tamaño del objeto al que está apuntando, y la aritmética de apuntadores toma en cuenta este tamaño. Cuando se incrementa un apuntador su valor es aumentado por la cantidad necesaria para hacer que apunte al "siguiente" de lo que esté apuntando. En otras palabras, es incrementado por el tamaño del objeto al que apunta.

Cuando aplicamos esto al ejemplo, `multi` es un apuntador a un arreglo de cuatro elementos enteros con un tamaño de 8. Si se incrementa a `multi` su valor debe incrementarse en 8 (el tamaño de un arreglo de cuatro elementos enteros). Por lo tanto, si `multi` apunta a `multi[0]`, `(multi+1)` debe apuntar a `multi[1]`. El programa del listado 15.3 comprueba esta teoría.

Captura

Listado 15.3. Este programa demuestra la aritmética de apuntadores con arreglos multidimensionales.

```
1: /* Demuestra la aritmética de apuntadores con */
2: /* apuntadores a arreglos multidimensionales. */
3:#
4: #include < stdio.h>
5:
6:  int multi[2][4];
7:
8: main()
9: {
10:     printf("\nThe value of (multi) = %u", multi);
11:     printf("\nThe value of (multi + 1) = %u", (multi+1));
12:     printf("\nThe address of multi[1] = %u", &multi[1]);
13: }
```

Salida

```
The value of (multi) = 1376
The value of (multi + 1) = 1384
The address of multi[1] = 1384
```

Análisis Los valores precisos pueden ser diferentes en su sistema, pero las relaciones son las mismas. Al incrementar a `multi` en uno se aumenta su valor en 8, y hace que apunte al siguiente elemento del arreglo, `multi[1]`.

En este ejemplo ha visto que `multi` es un apuntador a `multi[0]`. También ha visto que `multi[0]` es en sí mismo un apuntador (a `multi[0][0]`). Por lo tanto, `multi` es un apuntador a apuntador. Para usar la expresión `multi` para accesar datos del arreglo se debe usar indirección doble. Para imprimir el valor guardado en `multi[0][0]` se podría usar cualquiera de los siguientes tres enunciados:

```
printf("%d", mutli[0][0];
printf("%d", *mutli[0];
printf("%d", **mutli;
```

Estos conceptos se aplican igualmente a arreglos con tres o más dimensiones. Por lo tanto, un arreglo tridimensional es un arreglo con elementos que son cada uno arreglos de dos dimensiones y cada uno de estos elementos es en sí mismo un arreglo de arreglos de una dimensión.

Este material sobre ar[redacted]

¿Cómo puede usar apuntadores a elementos de arreglos multidimensionales? De la misma manera que para los arreglos de una sola dimensión, se deben usar apuntadores para pasar un arreglo a una función. Esto es ilustrado para un arreglo multidimensional en el listado 15.4, que usa dos métodos para pasar un arreglo multidimensional a una función.

Captura

Listado 15.4. Paso de un arreglo multidimensional a una función usando un apuntador.

```
1: /* Demuestra el paso a una función de un apuntador a */
2: /* un arreglo multidimensional. */
3:
4: #include <stdio.h>
5:
6: void printarray_1(int (*ptr)[4]);
7: void printarray_2(int (*ptr)[4], int n);
8:
9: main()
10:{
11:     int  multi[3][4] = { { 1, 2, 3, 4 },
12:                          { 5, 6, 7, 8 },
13:                          { 9, 10, 11, 12 } };
14:    /* ptr es un apuntador a un arreglo de 4 enteros.*/
15:
16:      int (*ptr)[4], count;
17:
18:     /* Hace que ptr apunte al primer elemento de multi.*/
19:
20:     ptr = multi;
21:
22:     /* En cada ciclo, ptr es incrementado para que apunte al siguiente */
23:     /* elemento de multi (el siguiente arreglo de enteros de 4 elementos)*/
24:
25:     for (count = 0; count < 3; count++)
26:          printarray_1(ptr++);
27:
28:     puts("\n\nPress a key...");
29:     getch();
30:
31:     printarray_2(multi, 3);
32:
33: }
34:
35: void printarray_1(int (*ptr)[4])
36: {
37:     /* Imprime los elementos de un solo arreglo entero de 4 elementos   */
38:     /* p es un apuntador a tipo int. Se debe usar un modificador de tipo */
39:     /* para hacer que p sea igual a la dirección de ptr. */
40:
```

Listado 15.4. continuación

```
41:     int *p, count;
42:     p = (int *)ptr;
43:
44:     for (count = 0; count < 4; count++)
45:         printf("\n%d", *p++);
46: }
47:
48: void printarray_2(int (*ptr)[4], int n)
49: {
50:     /* Imprime los elementos de un arreglo entero de n por 4 elementos.*/
51:
52:     int *p, count;
53:     p = (int *)ptr;
54:
55:     for (count = 0; count < (4 * n); count++)
56:         printf("\n%d", *p++);
57: }
```

Salida

```
1
2
3
4
5
6
7
8
9
10
11
12

Press a key...

1
2
3
4
5
6
7
8
9
10
11
12
```

Análisis El programa declara y asigna valor inicial a un arreglo de enteros, `multi[3][4]`, en las líneas 11 a 13. Tiene dos funciones, `printarray_1()` y `printarray_2()` que imprimen el contenido del arreglo.

A la función `printarray_1()` (líneas 35 a 46) le es pasado solamente un argumento, un apuntador a un arreglo de 4 enteros. La función imprime los cuatro elementos del arreglo. La primera vez que `main()` llama a `printarray_1()` en la línea 26, le pasa un apuntador al primer elemento (el primer arreglo de cuatro elementos enteros) en `multi`. Luego llama a la función dos veces más, incrementando el apuntador cada vez para que apunte al segundo y luego al tercer elemento de `multi`. Después de que se han hecho estas tres llamadas son desplegados los 12 enteros de `multi`.

La segunda función, `printarray_2()`, usa un diferente método. También le es pasado un apuntador a un arreglo de cuatro enteros, pero también le es pasada una variable entera donde se le da la cantidad de elementos (la cantidad de arreglos de cuatro enteros) que contiene el arreglo multidimensional. Con una sola llamada desde la línea 31, `printarray_2()` despliega el contenido completo de `multi`.

Ambas funciones usan notación de apuntadores para avanzar por los enteros individuales del arreglo. La notación `(int *)ptr` en ambas funciones (líneas 42 a 53) tal vez no sea clara. El `(int *)` es un convertidor específico de tipo, que temporalmente cambia el tipo de dato de la variable de como haya sido declarado a uno nuevo. El convertidor específico de tipo es requerido cuando se asigna el valor de `ptr` a `p`, debido a que hay apuntadores a diferentes tipos (`p` es un apuntador a tipo `int`, y en cambio `ptr` es un apuntador a un arreglo de cuatro enteros). El C no permite que se asigne el valor de un apuntador a otro apuntador de diferente tipo. El convertidor específico de tipo le dice al compilador "solamente para este enunciado trata a `ptr` como un apuntador a tipo `int`". El Día 20, "Otras funciones", trata a mayor detalle los convertidores específicos de tipo.

DEBE

NO DEBE

NO DEBE Olvidar usar al operador de indirección doble (`**`) cuando declare un apuntador a un apuntador.

NO DEBE Olvidar que un apuntador se incrementa por el tamaño del tipo de apuntador (por lo general a lo que apunta).

NO DEBE Olvidar usar paréntesis cuando declare apuntadores a arreglos.

Declaración de un apuntador a un arreglo de caracteres:

```
char (*letras)[26];
```

Declaración de un arreglo de apuntadores a caracteres:

```
char *letras[26];
```

Arreglos de apuntadores

Recuerde del Día 8, "Arreglos numéricos", que un arreglo es una colección de posiciones de almacenamientos de datos, que tienen el mismo tipo de dato y se les hace referencia con el mismo nombre. Debido a que los apuntadores son uno de los tipos de datos del C, se pueden declarar y usar arreglos de apuntadores. Este tipo de construcción de programa puede ser muy poderoso en determinadas situaciones.

Tal vez el uso más común para un arreglo de apuntadores de da con las cadenas. Una cadena, tal como aprendió en el Día 10, "Caracteres y cadenas", es una secuencia de caracteres guardados en la memoria. El inicio de la cadena está indicado por un apuntador al primer carácter (un apuntador a tipo `char`), y el final de la cadena está marcado por un carácter nulo. Declarando y asignando valores iniciales a un arreglo de apuntadores a tipo `char`, se puede accesar y manejar una gran cantidad de cadenas usando el arreglo de apuntadores.

Cadenas y apuntadores: una revisión

Este es un buen momento para revisar un material del Día 10, "Caracteres y cadenas", en relación con la asignación de memoria y valor inicial de cadenas. Si se quiere asignar memoria y dar valor inicial a una cadena, se declara un arreglo de tipo `char` de la manera siguiente:

```
char mensaje[] = "Este es el mensaje.";
```

También se podría declarar un apuntador a tipo `char`

```
char *mensaje = "Este es el mensaje.";
```

Ambas declaraciones son equivalentes. En este caso el compilador asigna suficiente espacio para guardar la cadena con su carácter nulo terminal, y la expresión mensaje es un apuntador al comienzo de la cadena. ¿Qué hay de estas dos declaraciones?

```
char mensaje1[20];
char *mensaje2;
```

La primera línea declara un arreglo tipo `char` que es de 20 caracteres de largo, y `mensaje1` es un apuntador a la primera posición del arreglo. Aunque está asignado el espacio del arreglo, todavía no se le ha asignado valor inicial. La segunda línea declara a `mensaje2`, que es un apuntador a tipo `char`. Con este enunciado no se asigna espacio de almacenamiento para una cadena. Si se quiere crear una cadena y luego hacer que `mensaje2` apunte hacia ella, primero se debe asignar espacio para la cadena. En el Día 10, "Caracteres y cadenas", se aprendió la manera de usar la función `malloc()`, de asignación de memoria, para este objeto. Recuerde que cualquier cadena debe tener espacio asignado a ella, ya sea al momento de compilación en una declaración o al momento de ejecución con `malloc()`.

Arreglos de apuntadores a *char*

Ahora que hemos terminado con la revisión, ¿cómo declararía usted un arreglo de apuntadores? El siguiente enunciado declara un arreglo de 10 apuntadores a tipo `char`:

```
char *mensaje[10];
```

Cada elemento del arreglo `mensaje[]` es un apuntador individual a tipo `char`. Ya debe haber supuesto que se puede combinar la declaración con la la asignación de valor inicial y la asignación de espacio de almacenamiento para las cadenas.

```
char *mensaje[10] = { uno, dos, tres };
```

Esta declaración hace lo siguiente:

- Asigna un arreglo de 10 elementos llamado `mensaje`, donde cada elemento de mensaje es un apuntador a tipo `char`.
- Asigna espacio en algún lugar de memoria (sin importarnos dónde lo hace exactamente) y guarda las tres cadenas de valores iniciales, cada una con su carácter nulo terminal.
- Asigna el valor inicial de `mensaje[0]` para que apunte al primer carácter de la cadena `uno`, `mensaje[1]` para que apunte al primer carácter de la cadena `dos` y `mensaje[2]` para que apunte al primer carácter de la cadena `tres`.

Esto es ilustrado en la figura 15.4, que muestra la relación entre el arreglo de apuntadores y las cadenas. Observe que en este ejemplo no les es asignado valor inicial a los elementos de arreglo `mensaje[3]` hasta `mensaje[9]` para que apunten a algún lugar.

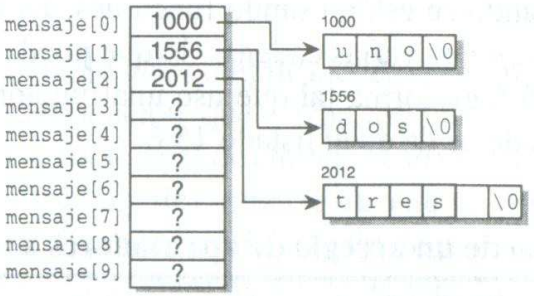


Figura 15.4. *Un arreglo de apuntadores a tipo* `char`.

Ahora veamos un ejemplo sobre el uso de arreglos de apuntadores.

Captura

Listado 15.5. Asignación de valor inicial y uso de un arreglo de apuntadores a tipo `char`.

```
1: /* Asignación de valor inicial a un arreglo de apuntadores a tipo char*/
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     char *message[8] = { "Four", "score", "and", "seven",
8:                          "years", "ago,", "our", "forefathers" };
9:   int count;
10:
11:     for (count = 0; count < 8; count++)
12:         printf("%s ", message[count]);
13: }
```

Salida

```
Four score and seven years ago, our forefathers
```

Análisis

El programa del listado 15.5 declara un arreglo de 8 apuntadores a tipo `char`, y les asigna valores iniciales para que apunten a 8 cadenas (líneas 7 a 8). Luego usa un ciclo `for`, en las líneas 11 y 12, para desplegar cada elemento del arreglo en la pantalla.

Probablemente pueda ver que el manejo del arreglo de apuntadores es más fácil que el manejo de las cadenas por sí mismas. Esta ventaja es obvia en programas más complicados, similares al que se presenta posteriormente en este capítulo. Ya verá en ese programa que la ventaja es mayor cuando se están usando funciones. Es más fácil pasar un arreglo de apuntadores a una función que varias cadenas, lo que puede ser ilustrado reescribiendo el programa del listado 15.5 en forma tal que use una función para desplegar las cadenas. Este programa modificado se da en el listado 15.6.

Captura

Listado 15.6. Paso de un arreglo de apuntadores a una función.

```
1: /* Paso de un arreglo de apuntadores a una función. */
2:
3: #include <stdio.h>
4:
5: void print_strings(char *p[], int n);
6:
7: main()
8: {
9:     char *message[8] = { "Four", "score", "and", "seven",
10:                         "years", "ago,", "our", "forefathers" };
11:
```

```
12:     print_strings(message, 8);
13: }
14:
15: void print_strings(char *p[], int n)
16: {
17:     int count;
18:
19:     for (count = 0; count < n; count++)
20:         printf("%s ", p[count]);
21: }
```

Salida

```
Four score and seven years ago, our forefathers
```

Análisis

Viendo la línea 15 se oberva que la función `print_strings()` toma dos argumentos. Uno es un arreglo de apuntadores a tipo `char` y el otro es la cantidad de elementos del arreglo. Por lo tanto, `print_strings()` podría ser usado para imprimir las cadenas apuntadas por cualquier arreglo de apuntadores.

Tal vez recuerde que en la sección sobre apuntadores a apuntadores se le dijo que vería una demostración posteriormente. Bien, la acaba de ver. El listado 15.6 declaró un arreglo de apuntadores, y el nombre del arreglo es un apuntador a su primer elemento. Cuando se pasa ese arreglo a una función, se está pasando un apuntador (el nombre del arreglo) a un apuntador (el primer elemento del arreglo).

Un ejemplo

Ahora es tiempo para un ejemplo más complicado. El programa del listado 15.7 usa muchas de las habilidades de programación que ya ha aprendido, incluyendo los arreglos de apuntadores. El programa acepta líneas de entrada desde el teclado, asignando espacio para cada línea conforme es tecleada, y llevando registro de las líneas por medio de un arreglo de apuntadores a tipo `char`. Cuando se señala el fin de una entrada, tecleando una línea en blanco, el programa ordena las cadenas en forma alfabética y las despliega en la pantalla.

Debe ver el diseño de este programa desde una perspectiva de programación estructurada. Primero haga una lista de las cosas que debe hacer el programa:

1. Recibe líneas de entrada desde el teclado, de una en una, hasta que se da una línea en blanco.
2. Ordena las líneas en orden alfabético.
3. Despliega en la pantalla las líneas ordenadas.

Esto hace suponer que el programa debe tener por lo menos tres funciones: una para recibir la entrada, otra para ordenar las líneas y otra para desplegar las líneas. Ahora puede diseñar cada función independientemente. ¿Qué necesita que haga la función de entrada, llamada `get_lines()`? Nuevamente haga una lista:

1. Que lleve registro de la cantidad de líneas recibidas, y que regrese ese valor al programa que la llama una vez que todas las líneas han sido tecleadas.
2. Que no permita la entrada de más líneas que el máximo predefinido.
3. Que asigne espacio de almacenamiento para cada línea.
4. Que lleve registro de todas las líneas, guardando apuntadores a cadenas en un arreglo.
5. Que regrese al programa que la llamó cuando se dé una línea en blanco.

Ahora piense acerca de la segunda función, aquella que ordena las líneas. Podría ser llamada `sort()`. (Bastante original, ¿o no?). El método de ordenamiento usado es uno de fuerza bruta, que compara las cadenas adyacentes y las intercambia si la segunda cadena es menor que la primera. Dicho con más precisión, la función compara las dos cadenas cuyos apuntadores se encuentran adyacentes en el arreglo de apuntadores, e intercambia los dos apuntadores en caso de ser necesario.

Para asegurarse de que el ordenamiento está completo se debe revisar el arreglo de principio a fin, comparando cada par de cadenas e intercambiándolas en caso de ser necesario. Para un arreglo de `n` elementos se debe recorrer el arreglo `n - 1` veces. ¿Por qué esto es necesario?

Cada vez que se pasa por el arreglo puede ser movido en una posición, por lo menos, un elemento dado. Por ejemplo, si la cadena que debe encontrarse en primer lugar se encuentra actualmente en la última posición, el primer paso por el arreglo la moverá a la penúltima posición, el segundo paso por el arreglo la moverá hacia arriba una posición más y así sucesivamente. Se requieren `n - 1` pasos para moverla a la primera posición donde le corresponde estar.

Por favor tome en cuenta que éste es un método de ordenamiento muy ineficiente y falto de elegancia. Sin embargo, es fácil de implementar y de entender, y adecuado para las listas pequeñas que ordena el programa de ejemplo.

La última función despliega en la pantalla las líneas ordenadas. Ya ha sido, de hecho, escrita en el listado 15.6, requiriendo solamente modificaciones menores.

Listado 15.7. Un programa que lee líneas de texto del teclado, las ordena alfabéticamente y despliega la lista ordenada.

```
1: /* Recibe una lista de cadenas desde el teclado, las ordena */
2: /* y luego las despliega en la pantalla.                    */
3:
```

```
4: #include <stdio.h>
5: #include <string.h>
6:
7: #define MAXLINES 25
8:
9: int get_lines(char *lines[]);
10: void sort(char *p[], int n);
11: void print_strings(char *p[], int n);
12:
13: char *lines[MAXLINES];
14:
15: main()
16: {
17:     int number_of_lines;
18:
19:     /* Lee las líneas desde el teclado. */
20:
21:     number_of_lines = get_lines(lines);
22:
23:     if ( number_of_lines < 0 )
24:     {
25:         puts(" Memory allocation error");
26:          exit(-1);
27:     }
28:
29:     sort(lines, number_of_lines);
30:      print_strings(lines, number_of_lines);
31:
32: }
33:
34: int get_lines(char *lines[])
35:{
36:      int n = 0;
37:     char buffer[80];  /* Almacenamiento temporal para cada línea. */
38:
39:     puts("Enter one line at time; enter a blank when done.");
40:
41:     while ((n < MAXLINES) && (gets(buffer) != 0) &&
42:            (buffer[0] != '\0'))
43:     {
44:         if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
45:             return -1;
46:         strcpy( lines[n++], buffer );
47:     }
48:     return n;
49:
50: } /* Fin de get_lines(). */
51:
52: void sort(char *p[], int n)
53: {
54:     int a, b;
```

Listado 15.7. continuación

```
55:     char *x;
56:
57:     for (a = 1; a < n; a++)
58:     {
59:         for (b = 0; b < n-1; b++)
60:         {
61:              if (strcmp(p[b], p[b+1]) > 0)
62:             {
63:                  x = p[b];
64:                  p[b] = p[b+1];
65:                  p[b+1] = x;
66:             }
67:         }
68:     }
69: }
70:
71: void print_strings(char *p[], int n)
72: {
73:     int count;
74:
75:     for (count = 0; count < n; count++)
76:         printf('\n%s ', p[count]);
77: }
```

Salida

```
Enter one line at time; enter a blank when done.
dog
apple
zoo
program
merry


apple
dog
merry
program
zoo
```

Análisis Examine algunos de los detalles del programa. Se usan varias funciones de biblioteca nuevas para diversos tipos de manejo de cadenas. A continuación se explican brevemente, y a mayor detalle en el Día 17, "Manipulación de cadenas". El archivo de encabezado STRING.H debe ser incluido en un programa que use estas funciones.

En la función `get_lines()` la entrada es controlada por el enunciado `while` de las líneas 41 y 42, que dicen lo siguiente:

```
while ((n < MAXLINES) && (get(buffer) != 0) &&
        (buffer[0] != '\0'))
```

La condición probada por el `while` tiene tres partes. La primera parte, `n < MAXLINES`, se asegura que no ha sido recibida todavía la cantidad máxima de líneas. La segunda parte, `gets(buffer) != 0` llama a la función de biblioteca `gets()`, para que lea una línea del teclado y la guarde en `buffer`, y verifique que no ha ocurrido el fin de archivo o cualquier otro error. La tercera parte, `buffer[0] != '\0'`, verifica que el primer carácter de la línea que se acaba de recibir no sea el carácter nulo, lo cual indicaría que se dio una línea en blanco.

Si cualquiera de estas tres condiciones se cumple el ciclo `while` termina, y la ejecución regresa al programa que la llamó con la cantidad de líneas dadas como valor de retorno. Si las tres condiciones son satisfechas, se ejecuta el siguiente enunciado `if` de la línea 44:

```
if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
```

La primera parte de la condición llama a `malloc()` para que asigne espacio para la cadena que se acaba de recibir. La función `strlen()` regresa la longitud de la cadena pasada como argumento, y el valor es incrementado en 1 para que de esta forma `malloc()` asigne espacio para la cadena más su carácter nulo terminal.

La función de biblioteca `malloc()`, como debe recordar, regresa un apuntador. El enunciado asigna el valor del apuntador regresado por `malloc()` al elemento correspondiente del arreglo de apuntadores. Si `malloc()` regresa NULL, el ciclo `if` regresa la ejecución al programa que la llamó con un valor de retorno de -1. El código de `main()` revisa el valor de retorno de `get_lines()` y si se le regresa un valor menor que 0. En caso de que suceda, las líneas 23 a 27 reportan un error de asignación de memoria y dan por terminado el programa.

Si la asignación de memoria fue satisfactoria, el programa usa la función `strcpy()`, en la línea 46, para copiar la cadena de la posición del buffer de almacenamiento temporal al espacio de almacenamiento que se acaba de asignar con `malloc()`. Luego repite el ciclo `while`, obteniendo otra línea de entrada.

Una vez que la ejecución regresa a `main()` desde `get_lines()`, ha sucedido lo siguiente (suponiendo que no sucedió un error de asignación de memoria):

- Se han leído una cantidad de líneas de texto del teclado y guardado en memoria como cadenas terminadas con nulo.
- El arreglo lines[] contiene un apuntador a cada cadena. El orden de los apuntadores en el arreglo es el orden en que fueron recibidas las cadenas.
- La variable `numer_of_lines` guarda la cantidad de líneas que fueron recibidas.

Ahora es el momento de ordenar. Recuerde que no se van a mover, de hecho, las cadenas, sino solamente el orden de los apuntadores en el arreglo `lines[]`. Vea el código de la función `sort()`. Contiene un ciclo `for` anidado en otro (líneas 57 a 68). El ciclo externo ejecuta `number_of_lines - 1` veces. Cada vez que ejecuta el ciclo externo, el ciclo

interno avanza paso a paso por el arreglo de apuntadores, comparando a (cadena n) con (cadena n + 1), desde n = 0 hasta n = number_of_lines - 1. La comparación es ejecutada por la función de biblioteca strcmp() de la línea 61, que recibe apuntadores a las dos cadenas. La función strcmp() regresa uno de los siguientes valores:

Un valor mayor que 0 si la primera cadena es mayor que la segunda cadena.

Cero si las dos cadenas son idénticas.

Un valor menor que cero si la segunda cadena es mayor que la primera.

En el programa un valor de retorno de strcmp() mayor que 0 significa que la primera cadena es "mayor que" la segunda y por lo tanto debe ser intercambiada (esto es, sus apuntadores en lines[] deben ser intercambiados). Esto se logra con la ayuda de una variable temporal x. Las líneas 63, 64 y 65 ejecutan el intercambio.

Cuando la ejecución del programa regresa de sort(), los apuntadores en lines[] ya están ordenados adecuadamente: un apuntador a la cadena "mínima" está contenido en lines[0], un apuntador a la siguiente "mínima" está contenido en lines[1], y así sucesivamente. Digamos, por ejemplo, que se dieron las siguientes cinco líneas en este orden:

```
dog
apple
zoo
program
merry
```

La situación antes de llamar a sort() está ilustrada en la figura 15.5 y la situación después del regreso de sort() está ilustada en la figura 15.6.

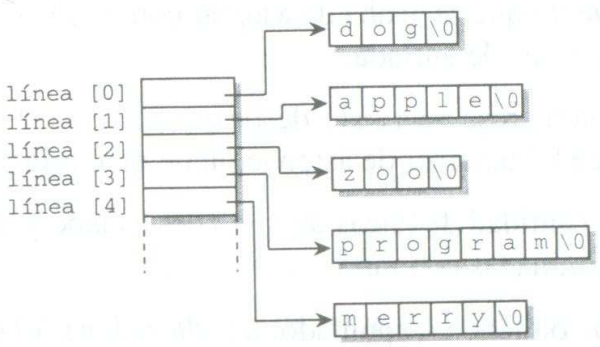


Figura 15.5. *Antes de ordenar los apuntadores están en el mismo orden en que fueron tecleadas las cadenas.*

Por último, el programa llama a la función print_strings(), para desplegar en la pantalla la lista ordenada de cadenas. Esta función debe serle familiar de los ejemplos anteriores en este capítulo.

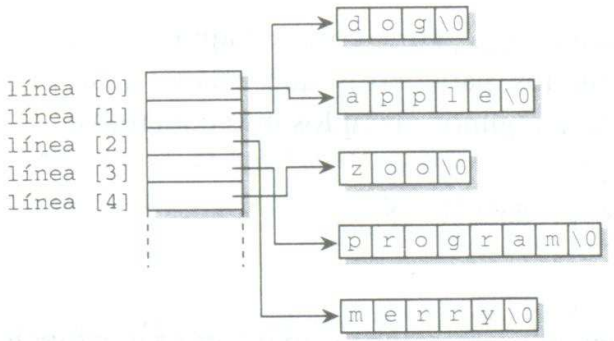


Figura 15.6. *Después de ordenar los apuntadores están ordenados de acuerdo con el orden alfabético de las cadenas.*

El programa en el listado 15.7 es el más complejo que ha visto en este libro. Usa muchas de las técnicas de programación del C que han sido tratadas en capítulos anteriores. Con la ayuda de las explicaciones anteriores debe ser capaz de seguir la operación del programa y comprender cada paso. Si encuentra áreas que no le sean claras, revise la sección correspondiente del libro, para que las comprenda antes de continuar a la siguiente sección.

Apuntadores a funciones

Los apuntadores a funciones proporcionan otro método para llamar funciones. Tal vez diga, "Un momento ¿cómo es que se puede tener un apuntador a una función? Un apuntador guarda la dirección donde se encuentra almacenada una variable, ¿no es así?".

Bueno, si y no. Es cierto que un apuntador guarda una dirección, pero no tiene que ser la dirección donde se encuentre almacenada una variable. Cuando el programa ejecuta, el código para cada función es cargado en memoria, comenzando en una dirección específica. Un apuntador a una función guarda la dirección inicial de una función, es decir, su punto de entrada.

¿Para qué usar un apuntador a una función? Tal como se dijo anteriormente, proporciona una manera más flexible para llamar a una función. Permite que el programa "elija" entre varias funciones, seleccionando la que es adecuada para las circunstancias actuales.

Declaración de un apuntador a una función

De manera similar a los demás apuntadores, se deben declarar los apuntadores a función. La forma general de la declaración es

```
tipo (*prt_a_func)(lista_de_parámetros);
```

Este enunciado declara a `ptr_a_func` como un apuntador a una función que regresa `tipo` y a la que se le pasan los parámetros que están en la `lista_de_parámetros`. A continuación se presentan algunos ejemplos más concretos:

```
int (*func1)(int x);
void (*func2)(double y, double z);
char (*func3)(char *p[]);
void (*func4)();
```

La primera línea declara a `func1` como un apuntador a una función que toma un argumento de tipo `int` y regresa un tipo `int`. La segunda línea declara a `func2` como un apuntador a una función que toma dos argumentos tipo `double` y tiene un tipo de retorno `void` (no hay valor de retorno). La tercera línea declara a `func3` como un apuntador a una función que toma como argumentos a un arreglo de apuntadores a tipo `char` y regresa un tipo `char`. La última línea declara a `func4` como un apuntador a una función que no toma ningún argumento y que tiene un tipo de retorno `void`.

¿Por qué se necesitan los paréntesis alrededor del nombre del apuntador? ¿Por qué no puede escribir:

```
int *func1(int x);
```

en este primer ejemplo?

La razón tiene que ver con la precedencia del operador de indirección, *. Tiene una precedencia relativamente baja, más baja que los paréntesis que rodean la lista de parámetros. La declaración que se acaba de dar, sin el primer juego de paréntesis, declara a `func1` como una función que regresa un apuntador a tipo `int`. (Las funciones que regresan apuntadores son tratadas en el Día 18, "Cómo obtener más de las funciones".) Cuando declare un apuntador a una función, recuerde que siempre hay que incluir un juego de paréntesis alrededor del nombre de apuntador y del operador de indirección, ya que de no hacerlo se meterá en problemas.

Inicialización y uso de un apuntador a una función

Un apuntador a una función no sólo debe ser declarado, sino que se le debe dar un valor inicial para que apunte a algo. Ese "algo" es, por supuesto, una función. No hay nada especial acerca de una función que es apuntada. El único requisito es que el tipo de retorno y la lista de parámetros concuerde con el tipo de retorno y la lista de parámetros de la declaración de apuntador. Por ejemplo, el siguiente código declara y define una función y un apuntador a esa función.

```
float cuadrado(float x);          /* El prototipo de función.      */
float (*p)(float x);              /* La declaración de apuntador.  */
float cuadrado(float x)           /* La definición de función.     */
```

```
{
   return x * x;
}
```

Como la función `cuadrado()` y el apuntador `p` tienen los mismos parámetros y tipo de retorno, se puede dar valor inicial a `p` para que apunte a `cuadrado`, de la manera siguiente:

```
p = cuadrado;
```

Luego se puede llamar a la función, usando el apuntador de la siguiente manera:

```
respuesta = p(x);
```

Así de simple. Para un ejemplo real, compile y ejecute el programa del listado 15.8, que declara y asigna valor inicial a un apuntador a una función y luego llama dos veces a la función, usando la primera vez el nombre de la función y la segunda el apuntador a la función. Ambas llamadas producen el mismo resultado.

Captura

Listado 15.8. El uso de un apuntador a función para llamar la función.

```
1: /* Demostración de la declaración y uso de un apuntador a una función */
2:
3: #include <stdio.h>
4:
5: /* El prototipo de función. */
6:
7: float square(float x);
8:
9: /* La declaración de apuntador. */
10:
11: float (*p)(float x);
12:
13: main()
14: {
15:    /* Asigna valor inicial a p para que apunte a square(). */
16:
17:     p = square;
18:
19:    /* Llama a square() de dos maneras. */
20:
21:    printf("%f  %f", square(6.6), p(6.6));
22: }
23:
24: float square(float x)
25: {
26:    return x * x;
27: }
```

Salida

```
43.559999  43.559999
```



Nota: La precisión de los valores puede dar lugar a que algunos números no se desplieguen como el valor exacto tecleado. Por ejemplo, 43.56 puede aparecer como 43.559999.

Análisis La línea 7 declara a `square()`, y la línea 11 declara al apuntador, `p`, a una función, que contiene un argumento `float` y regresando un valor `float`, que concuerda con la declaración de `square()`. La línea 17 pone al apuntador, `p`, igual a `square`. Observe que no se usan paréntesis ni con square ni con `p`. La línea 21 imprime los valores de retorno de las llamadas a `square()` y `p()`.

Un nombre de función sin los paréntesis es un apuntador a la función (se parece a lo que sucede con los arreglos, ¿no es así?). ¿A qué viene el declarar y usar por separado un apuntador a función? Pues bien, el nombre de la función es en sí mismo una constante de apuntador y no puede ser cambiado (nuevamente, similar a los arreglos). Por el contrario, una variable de apuntador sí puede ser cambiada. Específicamente, puede hacerse que apunte a diferentes funciones cuando sea necesario.

El programa del listado 15.9 llama a una función, pasándole un argumento entero. Dependiendo del valor del argumento, la función asigna el valor inicial de un apuntador para que apunte alguna de tres funciones, y luego use el apuntador para que llame a la función correspondiente. Cada una de estas tres funciones despliega un mensaje específico en la pantalla.

Captura **Listado 15.9. Uso de un apuntador a función para llamar a funciones diferentes, dependiendo de las circunstancias del programa.**

```
1: /* Uso de un apuntador para llamar diferentes funciones. */
2:
3: #include <stdio.h>
4:
5: /* Los prototipos de función. */
6:
7: void func1(int x);
8: void one(void);
9:  void two(void);
10: void other(void);
11:
12:  main()
13: {
14:      int a;
15:
16:     for (;;)
17:     {
18:         puts("\nEnter an integer between 1 and 10, 0 to exit: ");
19:          scanf("%d", &a);
20:
```

```
21:          if (a == 0)
22:             break;
23:
24:          func1(a);
25:      }
26: }
27:
28: void func1(int x)
29: {
30:      /* El apuntador a función. */
31:
32:      void (*ptr)(void);
33:
34:      if (x == 1)
35:          ptr = one;
36:      else if (x == 2)
37:          ptr = two;
38:      else
39:          ptr = other;
40:
41:      ptr();
42: }
43:
44: void one(void)
45: {
46:      puts(" You entered 1.");
47: }
48:
49: void two(void)
50: {
51:      puts("You entered 2.");
52: }
53:
54: void other(void)
55: {
56:      puts("You entered something other than 1 or 2.");
57: }
```

Salida

```
Enter an integer between 1 and 10, 0 to exit:
2
You entered 2.

Enter an integer between 1 and 10, 0 to exit:
11
You entered something other than 1 or 2.
Enter an integer between 1 and 10, 0 to exit:
0
```

Análisis

Este programa emplea un ciclo infinito, en la línea 16, para hacer que continúe el programa hasta que se teclee un 0. Cuando se da un valor, si no se trata de 0 es pasado a `func1()`. Observe que la línea 32 en `func1()` contiene una declaración para un

apuntador a una función (`ptr`). Esto hace que sea local a `func1()`, lo que es adecuado debido a que ninguna otra parte del programa necesita tener acceso a ello. `func1()` luego usa este valor para poner a `ptr` igual a la función adecuada (líneas 34 a 39). Luego en la línea 41 se hace una llamada a `ptr()` que llama a la función adecuada.

Por supuesto que el programa del listado 15.9 es solamente para propósitos de ilustración. Podría haber logrado fácilmente los mismos resultados sin usar un apuntador a una función.

Ahora puede aprender otra forma de usar los apuntadores para llamar diferentes funciones: pasando el apuntador como argumento a una función. El programa del listado 15.10 es una revisión del listado 15.9.



Listado 15.10. Paso de un apuntador a función como argumento.

```
1: /* Paso de un apuntador a función como argumento. */
2:
3: #include <stdio.h>
4:
5: /* Los prototipos de función. La función func1() toma como */
6: /* su único argumento un apuntador a función */
7: /* que no toma argumentos y no tiene valor de retorno. */
8:
9: void func1(void (*p)(void));
10: void one(void);
11: void two(void);
12: void other(void);
13:
14: main()
15: {
16:     /* El apuntador a función. */
18:     void (*ptr)(void);
19:     int  a;
20:
21:     for (;;)
22:     {
23:         puts("\nEnter an integer between 1 and 10, 0 to exit: ");
24:         scanf("%d", &a);
25:
26:         if (a == 0)
27:             break;
28:         else if (a == 1)
29:             ptr = one;
30:         else if (a == 2)
31:             ptr = two;
32:         else
33:             ptr = other;
34:
35:         func1(ptr);
```

```
36:     }
37: }
38:
39: void func1(void (*p)(void))
40: {
41:     p();
42: }
43:
44: void one(void)
45: {
46:     puts(_You entered 1._);
47: }
48:
49: void two(void)
50: {
51:     puts("You entered 2.");
52: }
53:
54: void other(void)
55: {
56:     puts("You entered something other than 1 or 2.");
57:  }
```

Salida

```
Enter an integer between 1 and 10, 0 to exit:
2
You entered 2.
Enter an integer between 1 and 10, 0 to exit:
11
You entered something other than 1 or 2.
Enter an integer between 1 and 10, 0 to exit:
0
```

Análisis

Observe la diferencia entre los listados 15.9 y 15.10. La declaración del apuntador a una función ha sido movida de la línea 18 en `main()` a donde se le necesita. El código en `main()` ahora asigna el valor inicial del apuntador para que apunte a la función adecuada (líneas 26 a 33) y luego pasa el apuntador con el valor adecuado a `func1()`. Esta función, `func1()`, realmente no tiene ningún objeto en el listado 15.10, ya que todo lo que hace es llamar a la función apuntada por `ptr`. Recuerde que este programa es solamente para ilustración. Los mismos principios pueden ser usados en programas reales, tales como el ejemplo de la siguiente sección.

Una situación de programación en la cual debe usar apuntadores a funciones es cuando se necesita hacer ordenamientos. En algunas ocasiones tal vez quiera que se usen diferentes reglas de ordenamiento. Por ejemplo, tal vez quiera ordenar en una ocasión en orden alfabético y en otras en orden alfabético inverso. Usando apuntadores a funciones el programa puede llamar a la función adecuada para el ordenamiento. Dicho con más precisión, por lo general lo que es llamado es la función diferente de comparación.

Volvamos a ver el programa del listado 15.7. En la función `sort()` la forma actual de ordenamiento es determinado por el valor que regresa la función de biblioteca `strcmp()`, la que le dice al programa si una cadena dada es "menor que" otra cadena. ¿Qué tal si escribe dos funciones de comparación, una que ordene alfabéticamente (diciendo que A es menor que Z, por ejemplo) y otra que ordene en orden alfabético inverso (diciendo que Z es menor que A). El programa puede preguntarle al usuario cuál orden desea y, mediante el uso de apuntadores, la función de ordenamiento puede llamar la función de comparación adecuada. El listado 15.11 modifica el programa del listado 15.7 e incorpora esta característica.

Captura

Listado 15.11. Uso de apuntadores a función para controlar el tipo de ordenamiento.

```
1:  /* Recibe una lista de cadenas desde el teclado, las ordena */
2:  /* en forma ascendente o descendente y luego las despliega */
3:  /* en la pantalla. */
4:
5:  #include <stdio.h>
6:  #include <string.h>
7:
8:  #define MAXLINES 25
9:
10:  int get_lines(char *lines[]);
11:  void sort(char *p[], int n, int sort_type);
12:  void print_strings(char *p[], int n);
13:  int alpha(char *p1, char *p2);
14:  int reverse(char *p1, char *p2);
15:
16:  char *lines[MAXLINES];
17:
18:   main()
19:  {
20:     int number_of_lines, sort_type;
21:
22:     /* Lee las líneas desde el teclado. */
23:
24:     number_of_lines = get_lines(lines);
25:
26:     if ( number_of_lines < 0 )
27:     {
28:        puts("Memory allocation error");
29:        exit(-1);
30:     }
31:
32:     puts("Enter 0 for reverse order sort, 1 for alphabetical:
           " );
33:     scanf("%d", &sort_type);
34:
```

```
35:     sort(lines, number_of_lines, sort_type);
36:     print_strings(lines, number_of_lines);
37:
38: }
39:
40: int get_lines(char *lines[])
41: {
42:     int n = 0;
43:     char buffer[80];  /* Almacenamiento temporal para cada línéa. */
44:
45:     puts("Enter one line at time; enter a blank when done.");
46:
47:     while (n < MAXLINES && gets(buffer) != 0 && buffer[0] != '\0')
48:     {
49:         if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
50:         return -1;
51:         strcpy( lines[n++], buffer );
52:     }
53:     return n;
54:
55: } /* Fin de get_lines(). */
56:
57: void sort(char *p[], int n, int sort_type)
58: {
59:     int a, b;
60:      char *x;
61:
62:     /* El apuntador a función. */
63:
64:     int (*compare)(char *s1, char *s2);
65:
66:     /* Asigna valor inicial al apuntador para que apunte a la función */
67:     /* de comparación adecuada, dependiendo del argumento sort_type. */
68:
69:     compare = (sort_type) ? reverse : alpha;
70:
71:     for (a = 1; a < n; a++)
72:     {
73:         for (b = 0; b < n-1; b++)
74:         {
75:             if (compare(p[b], p[b+1]) > 0)
76:             {
77:                 x = p[b];
78:                 p[b] = p[b+1];
79:                 p[b+1] = x;
80:             }
81:         }
82:     }
83: }  /* fin de sort(). */
84:
```

Listado 15.11. continuación

```
85:  void print_strings(char *p[], int n)
86:  {
87:      int count;
88:
89:      for (count = 0; count < n; count++)
90:          printf("\n%s ", p[count]);
91:  }
92:
93:  int alpha(char *p1, char *p2)
94:  /* Comparación alfabética. */
95:  {
96:      return(strcmp(p2, p1));
97:  }
98:
99:  int reverse(char *p1, char *p2)
100: /* Comparación alfabética inversa. */
101: {
102:      return(strcmp(p1, p2));
103:  }
```

Salida

```
Enter one line at time; enter a blank when done.
Roses are red
Violets are blue
C has been around,
But it is new to you!
Enter 0 for reverse order sort, 1 for alphabetical:
0
Violets are blue
Roses are red
C has been around,
But it is new to you!
```

Análisis Las líneas 32 y 33 en `main()` le piden al usuario el orden deseado. El orden seleccionado es puesto en `sort_type`. Este valor es pasado a la función `sort()`, junto con la demás información que se describió para el listado 15.7. La función de ordenamiento contiene unos cuantos cambios. La línea 64 declara un apuntador a la función, llamado `compare()`, que toma dos apuntadores a carácter (cadenas) como argumentos. La línea 69 hace que `compare()` sea igual a alguna de las dos nuevas funciones añadidas al listado en base al valor de `sort_type`. Las dos nuevas funciones son `alpha()` y `reverse()`. `alpha()` usa la función de biblioteca `strcmp()` de la misma forma en que fue usada en el listado 15.7. `reverse()` no lo hace. `reverse()` cambia los parámetros pasados para que se ejecute un ordenamiento en sentido inverso.

DEBE	NO DEBE

DEBE Usar programación estructurada.

NO DEBE Olvidar el uso de paréntesis cuando declare apuntadores a funciones.

La declaración de un apuntador a una función que no lleva argumentos y que regresa un carácter es similar a ésta:

```
char (*func)();
```

La declaración de una función que regresa un apuntador a un carácter es similar a ésta:

```
char *func();
```

DEBE Asignar un valor inicial a un apuntador antes de usarlo.

NO DEBE Usar un apuntador a función que ha sido declarado con un tipo de retorno diferente o con argumentos diferentes a lo que usted necesita.

Resumen

Este capítulo ha tratado algunos de los usos avanzados de los apuntadores. Como puede darse cuenta, los apuntadores son fundamentales para el lenguaje C, y son raros los programadores que no usan apuntadores. Ya ha visto la manera de usar apuntadores a apuntadores, y cómo los arreglos de apuntadores pueden ser muy útiles cuando se trabaja con cadenas. También ha aprendido cómo el C trata a los arreglos multidimensionales, como si fueran arreglos de arreglos, y ha visto la manera de usar apuntadores con tales arreglos. Por último, ha aprendido la manera de usar apuntadores a funciones, una herramienta de programación importante y flexible.

Este ha sido un capítulo largo y pesado. Aunque algunos de sus temas son un tanto complicados, también son interesantes. Con este capítulo se ha adentrado en una de las capacidades más sofisticadas del lenguaje C. El poder y la flexibilidad son una de las razones principales del porqué el C es un lenguaje popular.

Preguntas y respuestas

1. ¿A qué tantos niveles puedo llegar con los apuntadores a apuntadores?

   Necesita revisar los manuales de su compilador para ver si hay alguna limitación. Por lo general, es poco práctico ir más allá de tres niveles de profundidad con los

apuntadores (apuntadores a apuntadores a apuntadores). La mayoría de los programas pocas veces van más allá de dos niveles.

2. ¿Hay alguna diferencia entre un apuntador a una cadena y un apuntador a un arreglo?

   No. Una cadena puede ser vista como un arreglo de caracteres.

3. ¿Es necesario usar los conceptos presentados en este capítulo para aprovecharse del C?

   Se puede usar al C sin usar nunca ningún concepto avanzado de apuntadores. Sin embargo, no aprovechará la potencia que ofrece el C. Haciendo manipulaciones de apuntadores, tales como las que se muestran en este capítulo, debe ser capaz de hacer virtualmente, cualquier tarea de programación en una manera rápida y eficiente.

4. ¿Hay otras ocasiones donde sean útiles los apuntadores a funciones?

   Sí. Los apuntadores a funciones también son usados con menús. En base a un valor regresado de un menú, se ajusta un apuntador a la función adecuada.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. Escriba código que declare una variable de tipo `float`, declare y asigne valor inicial a un apuntador a la variable y declare y asgine valor inicial a un apuntador a un apuntador.

2. Continuando con el ejemplo de la pregunta 1, digamos que quiere usar el apuntador a apuntador para asignar el valor de 100 a la variable `x`. ¿Qué, en caso de haberlo, hay de erróneo en el siguiente enunciado de asignación? En caso de no estar correcto, ¿cómo debiera escribirse?

   ```
   *ppx = 100;
   ```

3. Supongamos que ha declarado un arreglo de la manera siguiente:

   ```
   int arreglo[2][3][4];
   ```

   ¿Cuál es la estructura de este arreglo de acuerdo a como lo ve el compilador de C?

4. Continuando con el arreglo declarado en la pregunta 3, ¿qué significa la expresión arreglo[0][0]?

5. Nuevamente, usando el arreglo de la pregunta 3, ¿cuál de las siguientes comparaciones es cierta?

```
arreglo[0][0] == &arreglo[0][0][0];

arreglo[0][1] == arreglo[0][0][1];

arreglo[0][1] == &arreglo[0][1][0];
```

6. Escriba el prototipo para una función que tome como único argumento un arreglo de apuntadores a tipo `char`.

7. ¿Cómo "sabría" la función que se escribió como prototipo para la pregunta 6, qué tantos elementos hay en el arreglo de apuntadores que se le pasa?

8. ¿Qué es un apuntador a una función?

9. Escriba una declaración de un apuntador a una función que regrese un tipo `char` y tome como argumento un arreglo de apuntadores a tipo char.

10. Tal vez haya respondido la pregunta 9 con

```
char *ptr(char *x[]);
```

¿Qué tiene de erróneo esta declaración?

Ejercicios

1. ¿Qué declara lo siguiente?

   a. int *var1;

   b. int var2;

   c. int **var3;

2. ¿Qué declara lo siguiente?

   a. int a[3][12];

   b. int (*b)[12];

   c. int *c[12];

3. ¿Qué declara lo siguiente?

   a. char *z[10];

   b. char *y(int campo);

   c. char (*x)(int campo);

4. Escriba una declaración para un apuntador a una función que tome como argumento un entero y regrese una variable tipo `float`.
5. Escriba una declaración para un arreglo de apuntadores a funciones. Todas las funciones deben tomar como parámetro una cadena de caracteres y regresar un entero. ¿Para qué podría ser usado un arreglo de éstos?
6. Escriba un enunciado para declarar un arreglo de 10 apuntadores a tipo `char`.
7. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en el siguiente código?

```
int x[3][12];

int *ptr[12];

ptr = x;
```

Debido a las diversas soluciones posibles, no se proporcionan respuestas para los siguientes ejercicios.

8. Escriba un programa que declare un arreglo de caracteres de 12 por 12. Ponga X en uno de cada dos elementos. Use un apuntador al arreglo para imprimir los valores en la pantalla en una cuadrícula.
9. Escriba un programa que guarde 10 apuntadores a variables `double`. El programa debe aceptar los diez números del usuario, ordenarlos y luego imprimirlos en la pantalla. (Véase el listado 15.10.)
10. Modifique el programa del ejercicio 9 para permitirle al usuario que determine si el orden es de menor a mayor o de mayor a menor.

DIA 16

Uso de archivos de disco

Muchos de los programas que se escriben usan archivos de disco para uno u otro propósito: almacenamiento de datos, información de configuración, etc. Usted aprenderá hoy:

- ❑ A relacionar flujos con archivos de discos.
- ❑ Los dos tipos de archivo de disco del C.
- ❑ A abrir archivos.
- ❑ A escribir datos a un archivo.
- ❑ A leer datos de un archivo.
- ❑ A cerrar un archivo.
- ❑ La administración de archivos de disco.
- ❑ El uso de archivos temporales.

Flujos y archivos de disco

Tal como aprendió el Día 14, "Trabajando con la pantalla, la impresora y el teclado", el C ejecuta toda la entrada y la salida por medio de flujos. Vio la manera de usar los flujos predefinidos del C, que están conectados con dispositivos específicos tales como el teclado, la pantalla y la impresora. Los flujos de archivos de disco funcionan esencialmente de la misma forma, lo que es una ventaja de la entrada/salida de flujos. La principal diferencia con los flujos de archivo de disco es que el programa debe crear explícitamente un flujo asociado con un archivo de disco específico.

Tipos de archivos de disco

En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se vio que los flujos del C son de dos tipos: de *texto* y *binarios*. Se puede asociar cualquier tipo de flujo con un archivo, y es importante que comprenda la diferencia para usar el modo adecuado en los archivos.

Un *flujo de texto* (o archivo en modo texto) es una secuencia de líneas, donde cada línea contiene cero o más caracteres y termina con uno o más caracteres que indican el *fin-de-línea*. La longitud máxima de las líneas es de 255 caracteres. Una "línea" no es una cadena, debido a que no hay carácter o terminal \Ø. Cuando se usa un flujo de modo texto, se hace una traducción entre el carácter de nueva línea del C, `\n`, y los caracteres que use el sistema operativo para indicar el fin-de-línea en los archivos de disco. En los sistemas de la PC es una combinación de retorno de carro-avance de línea (CR-LF). Cuando los datos se escriben a un archivo de modo texto, cada `\n` es traducido a CR-LF, y cuando los datos son leídos del archivo de disco, cada CR-LF es traducido a `\n`.

Un *flujo binario* (o archivo en modo binario) es todo lo demás. Todos los datos son escritos y leídos sin cambio. Los caracteres nulo y fin-de-línea no tienen significado especial.

Algunas funciones de entrada/salida para archivos están restringidas a un solo modo de archivo, y otras funciones pueden usar cualquier modo. Este capítulo le enseña qué modo debe usar con las funciones.

Nombres de archivo

Se deben usar nombres de archivos cuando se manejen archivos de disco. Los nombres de archivos son guardados en cadenas, de manera similar a cualquier otro dato de texto. Los nombres son los mismos que se usan con el sistema operativo y deben seguir las mismas reglas. En el DOS un nombre de archivo completo consiste de un nombre de 1 a 8 caracteres, seguido opcionalmente por un punto y una extensión de 1 a 3 caracteres. Los caracteres permitidos en un nombre de archivo son las letras de la a a la z, los números del 0 al 9 y algunos otros caracteres como `_`, `!`, y `$`. Un nombre de archivo en un programa de C también puede contener información sobre la unidad y el directorio. Recuerde que el DOS usa el carácter de diagonal invertida para separar los nombres de directorio. Por ejemplo, para el DOS el nombre

```
c:\datos\lista.txt
```

se refiere a un archivo llamado LISTA.TXT que está en el directorio \DATOS de la unidad C:. Ya también sabe que el carácter de diagonal invertida tiene un significado especial para el C cuando se encuentra en una cadena. Para representar al carácter de diagonal invertida se le debe preceder por otro carácter de diagonal invertida. Por lo tanto, en un programa en C se representa al nombre de archivo de la siguiente manera:

```
char *nomarch = "c:\\datos\\lista.txt";
```

Sin embargo, si se da un nombre de archivo desde el teclado, teclee una sola diagonal invertida.

Apertura de un archivo para usarlo

El proceso de creación de un flujo asociado a un archivo de disco es llamado *apertura* del archivo. Cuando se abre un archivo, queda disponible para lectura (significando que se pueden recibir datos del archivo hacia el programa), para escritura (significando que se pueden guardar en el archivo datos del programa) o para ambas cosas. Cuando se ha terminado de usar el archivo se le debe cerrar. El cierre de un archivo se trata posteriormente, en el capítulo.

Para abrir un archivo se usa la función de biblioteca `fopen()`. El prototipo de `fopen()` está en STDIO.H y dice lo siguiente:

```
FILE *fopen(const char *nomarch, const char *modo);
```

El prototipo le dice que `fopen()` regresa un apuntador a tipo `FILE`, que es una estructura declarada en STDIO.H. Los miembros de la estructura `FILE` se usan por el programa en las diversas operaciones de acceso del archivo, pero no es necesario que uno trate con ellos. Sin embargo, para cada archivo que se quiera abrir se debe declarar un apuntador a tipo `FILE`. Cuando se llama a `fopen()` esa función crea una instancia de la estructura `FILE` y regresa un apuntador a esa estructura. Se usa este apuntador en todas las operaciones subsecuentes del archivo. Si `fopen()` falla, regresa `NULL`.

El argumento `nomarch` es el nombre del archivo a ser abierto. Tal como se dijo anteriormente, `nomarch` puede contener especificaciones de unidad de disco y directorio. El argumento `nomarch` puede ser una cadena literal encerrada entre comillas dobles, o un apuntador a una cadena guardada en cualquier lugar de memoria.

El argumento `modo` especifica el modo en el cual se quiere abrir el archivo. En este contexto, `modo` controla si el archivo es binario o de texto, y si es para lectura, escritura o ambas. Los valores posibles para modo se dan en la tabla 16.1.

Tabla 16.1. Valores de `modo`.

`modo`	Significado
`r`	Abre el archivo para lectura. Si el archivo no existe, `fopen()` regresa `NULL`.
`w`	Abre el archivo para escritura. Si el archivo del nombre especificado no existe, es creado. Si el archivo existe, son borrados los datos existentes en el archivo.
`a`	Abre el archivo para adición. Si un archivo de nombre especificado no existe, es creado. Si el archivo existe se añaden nuevos datos al final del archivo.
`r+`	Abre el archivo para lectura y escritura. Si el archivo del nombre especificado no existe, es creado. Si el archivo existe se añaden nuevos datos al inicio del archivo, sobreescribiendo los datos existentes.
`w+`	Abre el archivo para lectura y escritura. Si el archivo del nombre especificado no existe, es creado. Si el archivo existe es sobreescrito.
`a+`	Abre el archivo para lectura y adición. Si el archivo del nombre especificado no existe, es creado. Si el archivo existe, los nuevos datos son añadidos al final del archivo.

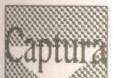
El modo predeterminado para un archivo es de texto. Para abrir un archivo en modo binario se añade una `b` al argumento del `modo`. Por lo tanto, un argumento de `modo a` abriría un archivo de modo texto para adición, y en cambio `ab` abriría un archivo de modo binario para adición.

Recuerde que `fopen()` regresa `NULL` cuando sucede un error. Las condiciones de error que pueden causar un valor de retorno de `NULL` incluyen las siguientes:

- El uso de un nombre de archivo inválido.
- El tratar de abrir un archivo en un disco que no está listo para ser usado (por ejemplo, la puerta de la unidad no está cerrada o el disco no está formateado).
- El tratar de abrir un archivo en un directorio que no existe o en una unidad de disco que no existe.
- El tratar de abrir un archivo que no existe en modo `"r"`.

Cada vez que use a `fopen()` necesita revisar si no existe algún error. No hay manera de decir exactamente el error que ha ocurrido, pero se puede desplegar un mensaje al usuario y tratar de volver a abrir el archivo, o se puede dar por terminado el programa.

El programa del listado 16.1 muestra a `fopen()`.



Listado 16.1. Uso de `fopen()` para abrir archivos de disco en varios modos.

```
1: /* Demuestra la función fopen(). */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     FILE *fp;
8:     char ch, filename[40], mode[4];
9:
10:     while (1)
11:     {
12:
13:         /* Recibe el nombre de archivo y el modo. */
14:
15:         printf("\nEnter a filename: ");
16:         gets(filename);
17:         printf("\nEnter a mode (max 3 characters): ");
18:         gets(mode);
19:
20:         /* Trata de abrir el archivo. */
21:
22:         if ( (fp = fopen( filename, mode )) != NULL )
```

Listado 16.1. continuación

```
23:         {
24:             printf("\nSuccessful opening %s in mode %s.\n",
25:                     filename, mode);
26:             fclose(fp);
27:             puts("Enter x to exit, any other to continue.");
28:             if ( (ch = getch()) == 'x')
29:                 break;
30:             else
31:                 continue;
32:         }
33:         else
34:         {
35:             fprintf(stderr, "\nError opening file %s in mode %s.\n",
36:                     filename, mode);
37:             puts("Enter x to exit, any other to try again.");
38:             if ( (ch = getch()) == 'x')
39:                 break;
40:             else
41:                 continue;
42:         }
43:     }
44: }
```

Salida

```
Enter a filename: list1601.c

Enter a mode (max 3 characters): r

Successful opening list1601.c in mode r.
Enter x to exit, any other to continue.

Enter a filename: list1601.c

Enter a mode (max 3 characters): w

Successful opening list1601.c in mode w.
Enter x to exit, any other to continue.
```

Análisis

El programa le pide el nombre del archivo y la especificación del modo en las líneas 15 a 18. Después de obtener los nombres la línea 22 trata de abrir el archivo y asigna su apuntador de archivo a `fp`. Como ejemplo de buena práctica de programación, el enunciado `if` de la línea 22 revisa si el apuntador del archivo abierto no es igual a `NULL`. Si `fp` no es igual a `NULL` se imprime un mensaje, diciendo que la apertura fue satisfactoria y que el usuario puede continuar. Si el apuntador de archivo es `NULL`, se ejecuta la condición `else` del ciclo `if`. La condición `else`, en las líneas 33 a 42, imprime un mensaje diciendo que ha habido un problema. Luego le pide al usuario que determine si el programa debe de continuar.

Se puede experimentar con diferentes nombres y modos, y ver con cuáles se obtiene un error. Si sucede un error, se da la alternativa de volver a dar la información o terminar el programa. Para forzar un error teclee un nombre de archivo inválido, tal como `[]`.

Escritura y lectura de datos de archivo

Un programa que usa un archivo de disco puede escribir datos al archivo, leer datos del archivo o ambas cosas. Se pueden escribir datos al archivo de disco en tres diferentes formas:

- Se puede usar salida formateada para guardar datos formateados en el archivo. Se debe usar salida formateada solamente con archivo de modo texto. El uso principal de la salida formateada es crear archivos que contengan datos de texto y numéricos que vayan a ser leídos por otros programas, tales como una hoja de cálculo o una base de datos. Rara vez usará salida formateada para crear un archivo que vaya a ser leído nuevamente por un programa en C.
- Se puede usar salida de carácter para guardar caracteres solos o líneas de caracteres en un archivo. Aunque es técnicamente posible usar salida de carácter con archivos de modo binario, puede ser un poco problemático. Debe restringir la salida de modo carácter al archivo de texto. El principal uso de la salida de carácter es guardar datos de texto (pero no numéricos), en forma tal que puedan ser luego leídos por un programa en C o por otros programas, tales como los procesadores de palabras.
- Se puede usar salida directa para guardar el contenido de una sección de memoria directamente en un archivo de disco. Este método es solamente para archivos binarios. La salida directa es la mejor manera para guardar datos que serán usados posteriormente por un programa en C.

Cuando se quieren leer datos de un archivo se tienen las mismas tres opciones: entrada formateada, entrada de carácter o entrada directa. El tipo de entrada que se use en un caso particular depende casi exclusivamente de la naturaleza del archivo leído.

Las descripciones anteriores de los tres tipos de entrada y salida de archivos sugieren tareas adecuadas para cada tipo de salida. Esto no es en ninguna forma un juego de reglas estrictas. El lenguaje C es muy flexible (¡lo cual es una de sus ventajas!), por lo que un programador listo puede hacer cualquier tipo de salida de archivo que mejor se ajuste a sus necesidades. Como programador novato, verá que se le facilitan las cosas si sigue por el momento estas reglas.

Entrada y salida de archivos formateados

La entrada/salida formateada de archivos maneja texto y datos numéricos que han sido formateados de una manera específica. Es muy similar a la entrada de teclado y salida a pantalla formateadas que se logran con las funciones `printf()` y `scanf()`, tales como se

dijo en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". La salida formateada se trata primero y a continuación se trata la entrada.

Salida formateada a archivos

La salida formateada a archivos se logra con la función de biblioteca `fprintf()`. El prototipo de `fprintf()` está en el archivo de encabezado STDIO.H y dice lo siguiente:

```
int fprintf(FILE *fp, char *fmt, ...);
```

El primer argumento es un apuntador a tipo `FILE`. Para escribir datos a un archivo de disco en particular, se pasa el apuntador que fue regresado cuando se abrió el archivo con `fopen()`.

El segundo argumento es la cadena de formato. Ya aprendió anteriormente sobre las cadenas de formato cuando se trató a `printf()` en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". La cadena de formato usada por `fprintf()` sigue exactamente las mismas reglas que para `printf()`. Por favor vea el Día 14, "Trabajando con la pantalla, la impresora y el teclado", para mayores detalles.

El argumento final es ... ¿Qué es lo que significa? En un prototipo de función los tres puntos representan un número variable de argumentos. En otras palabras, además del apuntador a archivo y los argumentos de cadena de formato, `fprintf()` toma cero, uno o más argumentos adicionales. En esto es similar a `printf()`. Estos argumentos son los nombres de las variables que serán enviados al flujo especificado.

Recuerde que `fprintf()` funciona casi de la misma manera que `printf()`, a excepción de que envía su salida al flujo especificado en la lista de argumentos. De hecho, si se especifica un argumento de flujo de `stdout`, `fprintf()` es idéntico a `printf()`.

El programa del listado 16.2 usa `fprintf()`.

Captura

Listado 16.2. Demostración de la equivalencia de la salida formateada de `fprintf()` para un archivo y para `stdout`.

```
1: /* Demuestra la función fprintf(). */
2:
3: #include <stdio.h>
4:
5: void clear_kb(void);
6:
7: main()
8: {
9:     FILE *fp;
10:     float data[5];
11:     int count;
12:     char filename[20];
```

```
13:
14:     puts("Enter 5 floating point numerical values.");
15:
16:     for (count = 0; count < 5; count++)
17:         scanf("%f", &data[count]);
18:
19:     /* Obtiene el nombre de archivo y abre el archivo. Primero borra */
20:     /* a stdin de cualquier carácter adicional. */
21:
22:     clear_kb();
23:
24:     puts("Enter a name for the file.");
25:     gets(filename);
26:
27:     if ( (fp = fopen(filename, "w")) == NULL)
28:     {
29:         fprintf(stderr, "Error opening file %s.", filename);
30:         exit(1);
31:     }
32:
33:     /* Escribe datos numéricos al archivo y a stdout */
34:
35:     for (count = 0; count < 5; count++)
36:     {
37:         fprintf(fp, "\ndata[%d] = %f", count, data[count]);
38:         fprintf(stdout, "\ndata[%d] = %f", count, data[count]);
39:     }
40:
41:     fclose(fp);
42: }
43:
44: void clear_kb(void)
45: /* Limpia a stdin de cualquier carácter que esté en espera. */
46: {
47:     char junk[80];
48:     gets(junk);
49: }
```

Salida

```
Enter 5 floating point numerical values.
3.14159
9.99
1.50
3.
1000.0001
Enter a name for the file.
numbers.txt

data[0] = 3.141590
data[1] = 9.990000
data[2] = 1.500000
data[3] = 3.000000
data[4] = 1000.000122
```

Análisis Este programa usa `fprintf()`, en las líneas 37 y 38, para enviar un poco de texto formateado y datos numéricos a `stdout` y a un archivo de disco. La única diferencia entre las dos líneas es el primer argumento. Después de ejecutar el programa use el editor para ver el contenido del archivo. Debe ser un duplicado exacto de la salida a pantalla.

Observe que el listado 16.2 usa la función `clear_kb()`, desarrollada en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". Esto es necesario para quitar de `stdin` cualquier carácter adicional que pudiera haber quedado de la llamada a `scanf()`. Si no se limpia a `stdin`, estos caracteres adicionales (específicamente la nueva línea) son leídos por el `gets()` que recibe el nombre de archivo, y el resultado sería un error de creación de archivo.

Entrada de archivo formateada

Para la entrada de archivo formateada use la función de biblioteca `fscanf()`, que se usa de manera similar a `scanf()` (vea el Día 14, "Trabajando con la pantalla, la impresora y el teclado"), a excepción de que la entrada viene del flujo especificado en vez de `stdin`. El prototipo para `fscanf()` es

```
int fscanf(FILE *fp, const char *fmt, ...);
```

El argumento `fp` es el apuntador a tipo `FILE` regresado por `fopen()`, y `fmt` es un apuntador a la cadena de formato que especifica la manera en que `fscanf()` debe leer la entrada. Los componentes de la cadena de formato son los mismos que para `scanf()`. Por último, los tres puntos indican uno o más argumentos adicionales, las direcciones de las variables a donde `fscanf()` asignará la entrada.

Para usar `fscanf()` tal vez quiera revisar la sección sobre `scanf()` en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". La función `fscanf()` funciona exactamente de la misma forma que `scanf()`, a excepción de que los caracteres son tomados del flujo especificado en vez de `stdin`.

Para demostrar a `fscanf()` se necesita un archivo de texto, que contenga algunos números o cadenas en un formato legible por la función. Use el editor para crear un archivo llamado `input.txt`, y teclee cinco números de punto flotante con algunos espacios entre ellos (espacios o nueva línea). Por ejemplo, el archivo podría parecerse a lo siguiente:

```
123.45   87.001
100.02
0.00456  1.0005
```

Ahora compile y ejecute el programa del listado 16.3.

Captura

Listado 16.3. Uso de `fscanf()` para leer datos formateados de un archivo de disco.

```
1: /* Lectura de datos de archivo formateado con fscanf(). */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     float f1, f2, f3, f4, f5;
8:     FILE *fp;
9:
10:     if ( (fp = fopen("INPUT.TXT", "r")) == NULL)
11:     {
12:         fprintf(stderr, "Error opening file.");
13:         exit(1);
14:     }
15:
16:     fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
17:     printf("The values are %f, %f, %f, %f, and %f.",
18:             f1, f2, f3, f4, f5);
19:
20:     fclose(fp);
21: }
```

Salida

```
The values are 123.45, 87.0001, 100.02, 0.00456, and 1.0005.
```

Nota: La precisión de los valores puede hacer que algunos números no se desplieguen exactamente como los valores dados. Por ejemplo, 100.02 puede aparecer como 100.01999.

Análisis

Este programa lee los cinco valores del archivo que se ha creado y luego los despliega en la pantalla. La llamada a `fopen()`, en la línea 10, abre el archivo para modo de lectura. También revisa si el archivo se abrió correctamente. Si el archivo no se abrió, la línea 12 despliega un mensaje y el programa termina (línea 13). La línea 16 muestra el uso de la función `fscanf()`. A excepción del primer parámetro es idéntica a `scanf()`, la cual ha estado usando a lo largo del libro. El primer parámetro apunta al archivo que se quiere que lea el programa. Se pueden hacer experimentos adicionales con `fscanf()`, creando archivos de entrada con el editor de programación y viendo la manera en que `fscanf()` lee los datos.

Entrada y salida de caracteres

Cuando se usa con archivos de disco, el término *E/S de carácter* se refiere tanto a un solo carácter como a una línea de caracteres. Recuerde que una línea es una secuencia de cero o más caracteres terminados por el carácter de nueva línea. Use E/S de carácter con archivos en modo texto. Las secciones que se encuentran a continuación describen las funciones de entrada/salida de carácter, y a continuación de ellas se encuentra un programa de demostración.

Entrada de carácter

Hay tres funciones para entrada de carácter: `getc()` y `fgetc()` para caracteres solos, y `fgets()` para líneas.

Las funciones *getc()* y *fgetc()*

Las funciones `getc()` y `fgetc()` son idénticas y pueden usarse en forma intercambiada. Ellas dan entrada a un solo carácter desde el flujo especificado. El prototipo de `getc()` está en STDIO.H.

```
int getc(FILE *fp);
```

El argumento `fp` es el apuntador regresado por `fopen()` cuando el archivo fue abierto. La función regresa el carácter recibido o `EOF` en caso de haber error.

La función *fgets()*

Para leer una línea de caracteres desde un archivo, use la función de biblioteca `fgets()`. El prototipo es

```
char *fgets(char *str, int n, FILE *fp);
```

El argumento `str` es un apuntador a un buffer donde será guardada la entrada, `n` es el número máximo de caracteres que se han de recibir y `fp` es el apuntador a tipo `FILE`, que fue regresado por `fopen()` cuando el archivo fue abierto.

Cuando se le llama, `fgets()` lee caracteres desde `fp` hacia memoria, comenzando en la posición apuntada por `str`. Los caracteres son leídos hasta que se encuentra una nueva línea o se han leído `n - 1` caracteres. Haciendo que `n` sea igual al número de bytes asignados para el buffer `str`, se previene que la entrada sobreescriba en la memoria más allá del espacio asignado. (El `n - 1` es para permitir el espacio para el `\0` terminal que `fgets()` añade). En caso satisfactorio, `fgets()` regresa `str`. Pueden suceder dos tipos de errores.

- Si sucede un error de lectura o se encuentra `EOF` antes de que cualquier carácter haya sido asignado a `str`, se regresa `NULL` y la memoria apuntada por `str` no es modificada.

- Si sucede un error de lectura o si se encuentra EOF después de que han sido asignados a str uno o más caracteres, se regresa NULL y la memoria apuntada por str contiene basura.

Puede ver que fgets() no recibe necesariamente una línea completa (esto es, hasta el siguiente carácter de nueva línea). Si han sido leídos n - 1 caracteres antes de que se encuentre una nueva línea, fgets() se detiene. La siguiente operación de lectura desde el archivo comienza donde se quedó la anterior. Para asegurarse de que fgets() lee cadenas completas, parándose solamente en las nuevas líneas, asegúrese de que el tamaño del buffer de entrada y el valor correspondiente de n pasado a fgets() son lo suficientemente grandes.

Salida de carácter

Necesita saber de dos funciones para salida de carácter, putc() y fputs().

La función *putc()*

La función de biblioteca putc() escribe un solo carácter a un flujo especificado. Su prototipo en STDIO.H dice

```
int putc(int ch, FILE *fp);
```

El argumento ch es de salida. De manera similar a otras funciones de carácter, es formalmente de tipo int, pero sólo se usa el byte de nivel inferior. El argumento fp es el apuntador asociado con el archivo (el apuntador regresado por fopen() cuando el archivo fue abierto). La función putc() regresa, en caso satisfactorio, el carácter que se acaba de escribir, o EOF en caso de que haya habido algún error. La constante simbólica EOF está definida en STDIO.H y tiene el valor - 1. Debido a que ningún carácter "real" tiene ese valor numérico, EOF puede ser usado como un indicador de error (solamente con los archivos de modo texto).

La función *fputs()*

Para escribir una línea de caracteres a un flujo, use la función de biblioteca fputs(). Esta función funciona de manera similar a puts(), tratada en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". La única diferencia es que en fputs() se puede especificar el flujo de salida. También fputs() no añade una nueva línea al final de la cadena, sino que uno debe incluirla explícitamente en caso deseado. Su prototipo en STDIO.H es

```
char fputs(char *str, FILE *fp);
```

El argumento str es un apuntador a la cadena terminada con carácter nulo que será escrita, y fp es el apuntador a tipo FILE regresado por fopen() cuando el archivo fue abierto. La cadena apuntada por str es escrita en el archivo sin el nulo terminal \0. La función fputs() regresa, en caso de ser satisfactoria, un valor no negativo, o EOF en caso de haber error.

Entrada y salida directas de archivos

Se usa E/S directa de archivo principalmente cuando se guardan datos que van a ser leídos posteriormente por el mismo u otro programa en C. La E/S directa se usa solamente con archivos de modo binario. Con salida directa se escriben bloques de datos de memoria a disco. La entrada directa es el reverso del proceso: un bloque de datos es leído de un archivo de disco hacia memoria. Por ejemplo, una sola llamada de función de salida directa puede escribir un arreglo completo de tipo `double` a disco, y una sola llamada de función de entrada directa puede leer el arreglo completo de disco hacia memoria. Las funciones de E/S directa son `fread()` y `fwrite()`.

La función *fwrite()*

La función de biblioteca `fwrite()` escribe un bloque de datos de memoria a un archivo en modo binario. Su prototipo en STDIO.H es

```
int fwrite(void *buf, int size, int count, FILE *fp);
```

El argumento `buf` es un apuntador a la región de memoria que guarda los datos que serán escritos al archivo. El tipo de apuntador es `void`, ya que puede ser un apuntador a cualquier cosa.

El argumento `size` especifica el tamaño, en bytes, de los elementos de datos individuales, y `count` especifica la cantidad de elementos que se han de escribir. Por ejemplo, si se quiere guardar un arreglo de 100 elementos enteros `size` sería `2` (debido a que cada `int` ocupa 2 bytes) y `count` sería `100` (debido a que el arreglo contiene 100 elementos). Para obtener el argumento `size` se puede usar el operador `sizeof()`.

El argumento `fp` es, por supuesto, el apuntador a tipo `FILE` regresado por `fopen()` cuando el archivo fue abierto. La función `fwrite()` regresa, en caso satisfactorio, la cantidad de elementos escritos. Si el valor regresado es menor que `count` significa que ha ocurrido algún error. Para revisar si hay errores se codifica a `fwrite()`, por lo general, de la manera siguiente:

```
if( (fwrite(buf, size, count, fp)) != count)
     fprintf(stderr, "Error al escribir al archivo.");
```

A continuación se presentan algunos ejemplos del uso de `fwrite()`.

Para escribir una sola variable `x` tipo `double` a un archivo:

```
fwrite(&x, sizeof(double), 1, fp);
```

Para escribir un arreglo `datos[]` de 50 estructuras de tipo `dirección` a un archivo, se tienen dos alternativas:

```
fwrite(datos, sizeof(dirección), 50, fp);
```

o

```
fwrite(datos, sizeof(datos), 1, fp);
```

El primer método escribe el arreglo como 50 elementos, donde cada elemento tiene el tamaño de una sola estructura tipo `dirección`. El segundo método trata al arreglo como un solo "elemento". Los dos métodos ejecutan exactamente la misma cosa.

Después de que `fread()` sea explicado en la siguiente sección, se presenta un programa que muestra ambos comandos.

La función *fread()*

La función de biblioteca `fread()` lee un bloque de datos desde un archivo de modo binario hacia memoria. Su prototipo en STDIO.H es

```
int fread(void *buf, int size, int count, FILE *fp);
```

El argumento `buf` es un apuntador a la región de memoria que recibe los datos leídos del archivo. De manera similar al caso de `fwrite()`, el tipo del apuntador es `void`.

El argumento `size` especifica el tamaño, en bytes, de los elementos de datos individuales que están siendo leídos, y `count` especifica la cantidad de elementos a leer, de manera similar a los argumentos usados por `fwrite()`. Nuevamente se usa frecuentemente el operador `sizeof()` para proporcionar el argumento `size`. El argumento `fp` es (como de costumbre) el apuntador a tipo `FILE`, que fue regresado por `fopen()` cuando el archivo fue abierto. La función `fread()` regresa la cantidad de elementos leídos. Puede ser menor que `count` si se llega al final de archivo o si sucede algún error.

El programa del listado 16.4 muestra el uso de `fwrite()` y `fread()`.

Captura

Listado 16.4. Uso de `fwrite()` y `fread()` para el acceso directo a archivos.

```
1: /* E/S directa de archivo con fwrite() y fread(). */
2:
3: #include <stdio.h>
4:
5: #define SIZE 20
6:
7: main()
8: {
9:     int count, array1[SIZE], array2[SIZE];
10:    FILE *fp;
11:
12:    /* Inicializa a array1[]. */
13:
14:    for (count = 0; count < SIZE; count++)
15:        array1[count] = 2 * count;
16:
```

Listado 16.4. continuación

```
17:     /* Abre un archivo en modo binario. */
18:
19:     if ( (fp = fopen("direct.txt", "wb")) == NULL)
20:     {
21:         fprintf(stderr, "Error opening file.");
22:         exit(1);
23:     }
24:     /* Guarda a array1[] en el archivo. */
25:
26:     if (fwrite(array1, sizeof(int), SIZE, fp) != SIZE)
27:     {
28:         fprintf(stderr, "Error writing to file.");
29:         exit(1);
30:     }
31:
32:     fclose(fp);
33:
34:     /* Ahora abre el mismo archivo para lectura en modo binario. */
35:
36:     if ( (fp = fopen("direct.txt", "rb")) == NULL)
37:     {
38:         fprintf(stderr, "Error opening file.");
39:         exit(1);
40:     }
41:
42:     /* Lee los datos en array2[]. */
43:
44:     if (fread(array2, sizeof(int), SIZE, fp) != SIZE)
45:     {
46:         fprintf(stderr, "Error reading file.");
47:         exit(1);
48:     }
49:
50:     fclose(fp);
51:
52:     /* Ahora despliega ambos arreglos para mostrar que son iguales. */
53:
54:     for (count = 0; count < SIZE; count++)
55:         printf("%d\t%d\n", array1[count], array2[count]);
56:
57: }
```

Salida

```
0       0
2       2
4       4
6       6
8       8
10      10
12      12
```

```
14      14
16      16
18      18
20      20
22      22
24      24
26      26
28      28
30      30
32      32
34      34
36      36
38      38
```

Análisis

El listado 16.4 muestra el uso de las funciones `fwrite()` y `fread()`. El programa asigna valores iniciales para un arreglo en las líneas 14 y 15. Luego usa a `fwrite()` en la línea 26 para guardar el arreglo en disco. El programa usa a `fread()`, en la línea 44, para leer los datos a un arreglo diferente. Por último, el programa despliega ambos arreglos en la pantalla, para mostrar que ahora contienen los mismos datos (líneas 54 y 55).

Cuando se guardan datos con `fwrite()`, no hay mucho que pueda fallar, aparte de algún tipo de error de disco. Sin embargo, con `fread()` se necesita tener cuidado. Por lo que toca a `fread()` los datos del disco son solamente una secuencia de bytes. La función no tiene manera de saber lo que representan. Por ejemplo, un bloque de 100 bytes pudiera ser 100 variables `char`, 50 variables `int`, 25 variables `long` o 25 variables `float`. Si se le pide a `fread()` que lea ese bloque en memoria, obedientemente lo hace. Sin embargo, si el bloque fue guardado de un arreglo de tipo `int` y se le guarda en un arreglo de tipo `float` no suceden errores, pero se obtienen resultados extraños. Cuando escriba programas asegúrese de que `fread()` se usa adecuadamente, leyendo datos en los tipos de variables y arreglos adecuados. Observe que en el listado 16.4 todas las llamadas a `fopen()`, `fwrite()` y `fread()` son revisadas para asegurarse que funcionan adecuadamente.

Buffer con archivos: cierre y vaciado de archivos

Cuando ya se ha terminado de usar un archivo, se le debe cerrar con la función `fclose()`. Ya se vio a `fclose()` en uso en los programas presentados anteriormente en el capítulo. Su prototipo es

```
int fclose(FILE *fp);
```

El argumento `fp` es el apuntador `FILE` asociado con el flujo; `fclose()` regresa `0` en caso satisfactorio o `-1` cuando hay error. Cuando se cierra un archivo, el buffer del archivo es vaciado (escrito al archivo). También se puede cerrar a todos los flujos abiertos, a excepción de los estándares (`stdin`, `stdout`, `stdprn`, `stderr` y `stdaux`). Su prototipo es

```
int fcloseall(void);
```

La función `fcloseall()` también vacía cualquier buffer de flujo y regresa la cantidad de flujos cerrados.

Cuando un programa termina (ya sea por haber llegado al final de `main()` o por ejecutar la función `exit()`), todos los flujos son vaciados y cerrados automáticamente. Sin embargo, es buena idea cerrar explícitamente a los flujos, en particular aquellos relacionados con archivos de disco, tan pronto como se ha terminado de usarlos. La razón tiene que ver con los buffers de flujo.

Cuando se crea un flujo asociado a un archivo de disco se crea y asocia automáticamente un buffer con el flujo. Un *buffer* es un bloque de memoria usado para almacenamiento temporal de los datos que están siendo escritos o leídos del archivo.

Se necesitan los buffer debido a que las unidades de disco son dispositivos orientados a bloque, lo que significa que trabajan más eficientemente cuando los datos son leídos y escritos en bloques de un tamaño determinado. El tamaño del bloque ideal difiere dependiendo del hardware específico que está siendo usado, y es típicamente del orden de unos cuantos cientos hasta unos miles de bytes. Sin embargo, no se necesita saber exactamente el tamaño del bloque.

El buffer asociado con un flujo de archivo sirve como una interfaz entre el flujo (que es orientado a caracteres) y el hardware del disco (que es orientado a bloques). Conforme el programa escribe datos al flujo los datos son guardados en el buffer hasta que se llena, y luego es escrito el contenido completo del buffer, como un bloque, al disco. Un proceso similar sucede cuando se leen datos de un archivo de disco. La creación y operación del buffer es completamente automática y no hay porqué meterse con ella. (El C también proporciona algunas funciones para el manejo de buffer, pero se encuentran más allá del alcance de este libro.)

En términos prácticos, esta operación de buffer significa que durante la ejecución del programa los datos que el programa "escribe" al disco pueden estar todavía en el buffer y no en el disco. Si el programa "se cuelga", si hay una falla de corriente o si sucede cualquier otro problema, los datos que todavía están en el buffer se pueden perder y no sabrá lo que quedó contenido en el archivo de disco.

Se puede vaciar a los buffer de un flujo sin cerrarlo, usando las funciones de biblioteca `fflush()` o `flushall()`. Use `fflush()` cuando quiera que se escriba el buffer de un archivo a disco mientras todavía se usa el archivo. Use `flushall()` para vaciar los buffer de todos los flujos abiertos. Los prototipos de estas funciones son

```
int fflush(FILE *fp);
```

y

```
int flushall(void);
```

El argumento `fp` es el apuntador `FILE`, regresado por `fopen()` cuando el archivo fue abierto. Si el archivo fue abierto para escritura `fflush()` escribe su buffer al disco. Si el archivo fue abierto para lectura el buffer es limpiado. La función `fflush()` regresa `0` en caso satisfactorio o `EOF` en caso de suceder algún error. La función `flushall()` regresa la cantidad de flujos abiertos.

DEBE	NO DEBE

DEBE Abrir un archivo antes de que trate de leerlo o escribirlo.

NO DEBE Asumir que un acceso a archivo se ejecutó correctamente. Siempre revise después de hacer una lectura, escritura o apertura para asegurarse de que la función operó correctamente.

DEBE Usar el operador `sizeof()` con las funciones `fwrite()` y `fread()`.

DEBE Cerrar todos los archivos que haya abierto.

NO DEBE Usar `fcloseall()` a menos de que tenga una razón para cerrar todos los flujos.

Acceso de archivos secuencial contra aleatorio

Cada archivo abierto tiene un indicador de posición de archivo asociado con él. El indicador de posición especifica dónde suceden las operaciones de lectura y escritura del archivo. La posición siempre es dada en términos de bytes a partir del inicio del archivo. Cuando un archivo nuevo es abierto el indicador de posición siempre se encuentra al principio del archivo, en la posición `0`. (Debido a que el archivo es nuevo con una longitud de `0`, no existe otra posición a la que se pueda hacer referencia.) Cuando se abre un archivo existente, el indicador de posición se encuentra al final del archivo en caso de que el archivo haya sido abierto en modo de adición, o al inicio del archivo si fue abierto en cualquier otro modo.

Las funciones de entrada/salida de archivo tratadas anteriormente en este capítulo hacen uso del indicador de posición. Las operaciones de escritura y lectura suceden en la posición del indicador de posición y también actualizan al indicador de posición. Por ejemplo, si se abrió un archivo para lectura y se leen 10 bytes, se reciben los primeros 10 bytes del archivo (los bytes que se encuentran en las posiciones del 0 al 9). Después de la operación de lectura el indicador de posición se encuentra en la posición 10, y la siguiente operación de lectura comienza ahí. Por lo tanto, si se quieren leer todos los datos de un archivo en forma

secuencial, o si se quieren escribir en forma secuencial los datos a un archivo, no se necesita tratar con el indicador de posición, debido a que las funciones de E/S de flujo se ocupan de él automáticamente.

Cuando se necesita más control se usan las funciones de biblioteca del C que le permiten determinar y cambiar el valor del indicador de posición. Mediante el control del indicador de posición se pueden ejecutar accesos *aleatorios* del archivo. Aquí "aleatorio" significa que se pueden leer datos de o escribir datos en cualquier posición de un archivo, sin tener que leer o escribir todos los datos anteriores.

Las funciones *ftell()* y *rewind()*

Para ajustar el indicador de posición al inicio del archivo use la función de biblioteca `rewind()`. Su prototipo en STDIO.H es

```
void rewind(FILE *fp);
```

El argumento `fp` es el apuntador `FILE` asociado con el flujo. Después de llamar a `rewind()`, el indicador de posición para el archivo es puesto al inicio del archivo (byte 0). Use `rewind()` si ha leído algunos datos de un archivo y quiere comenzar a leer desde el principio del archivo nuevamente, sin cerrar y volver a abrir el archivo.

Para determinar el valor del indicador de posición del archivo use a `ftell()`. El prototipo de esta función, que se encuentra en STDIO.H, dice

```
long ftell(FILE *fp);
```

El argumento `fp` es el apuntador `FILE` regresado por `fopen()` cuando el archivo fue abierto. La función `ftell()` regresa un `long`, que da la posición actual del archivo en bytes a partir del inicio del archivo (el primer byte está en la posición `0`). Si sucede un error `ftell()` regresa `-1L` (un -1 tipo `long`).

Para ambientarse con las operaciones de `rewind()` y `ftell()` vea el programa del listado 16.5.

Captura **Listado 16.5. Uso de `ftell()` y `rewind()`.**

```
1: /* Demuestra a ftell() y a rewind(). */
2:
3: #include <stdio.h>
4:
5: #define BUFLEN 6
6:
7: char msg[] = "abcdefghijklmnopqrstuvwxyz";
8:
9: main()
10: {
```

```
11:     FILE *fp;
12:     char buf[BUFLEN];
13:
14:     if ( (fp = fopen("TEXT.TXT", "w")) == NULL)
15:     {
16:         fprintf(stderr, "Error opening file.");
17:         exit(1);
18:     }
19:
20:     if (fputs(msg, fp) == EOF)
21:     {
22:         fprintf(stderr, "Error writing to file.");
23:         exit(1);
24:     }
25:
26:     fclose(fp);
27:
28:     /* Ahora abre el archivo para lectura. */
29:
30:     if ( (fp = fopen("TEXT.TXT", "r")) == NULL)
31:     {
32:         fprintf(stderr, "Error opening file.");
33:         exit(1);
34:     }
35:     printf("\nImmediately after opening, position = %ld", ftell(fp));
36:
37:     /* Lee cinco caracteres. */
38:
39:     fgets(buf, BUFLEN, fp);
40:     printf("\nAfter reading in %s, position = %ld", buf, ftell(fp));
41:
42:     /* Lee los siguientes cinco caracteres. */
43:
44:     fgets(buf, BUFLEN, fp);
45:     printf("\n\nThe next 5 characters are %s, and position now = %ld",
46:             buf, ftell(fp));
47:
48:     /* Regresa al inicio del flujo. */
49:
50:     rewind(fp);
51:
52:     printf("\n\nAfter rewinding, the position is back at %ld",
53:             ftell(fp));
54:
55:     /* Lee cinco caracteres. */
56:
57:     fgets(buf, BUFLEN, fp);
58:     printf("\nand reading starts at the beginning again: %s", buf);
59:     fclose(fp);
60: }
```

Salida

```
Immediately after opening, position = 0
After reading in abcde, position = 5

The next 5 characters are fghij, and position now = 10

After rewinding, the position is back at 0
and reading starts at the beginning again: abcde
```

Análisis Este programa escribe una cadena, `msg`, a un archivo llamado TEXT.TXT. Las líneas 14 a 18 abren a TEXT.TXT para escritura, y revisan para asegurarse que la apertura fue satisfactoria. Las líneas 20 a 24 escriben a `msg` al archivo usando `fputs()`, y nuevamente revisan para asegurarse de que la escritura fue satisfactoria. La línea 26 cierra el archivo con `fclose()`, completando el proceso de creación de un archivo para que lo use el resto del programa.

Las líneas 30 a 34 vuelven a abrir el archivo, pero en esta ocasión para lectura. La línea 35 imprime el valor de retorno de `ftell()`. Observe que esta posición está al principio del archivo. La línea 39 ejecuta un `gets()` para leer cinco caracteres. Los cinco caracteres y la nueva posición del archivo se imprimen en la línea 40. Observe que `ftell()` regresa el desplazamiento adecuado. La línea 50 llama a `rewind()` para poner el apuntador de regreso al principio del archivo, antes de que la línea 52 vuelva a imprimir la posición del archivo. Esto debe confirmarle que `rewind()` reajusta la posición. Una lectura adicional en la línea 57 confirma que el programa está nuevamente de regreso al principio del archivo. La línea 59 cierra el archivo antes de terminar el programa.

La función *fseek()*

Se puede tener un control más preciso sobre el indicador de posición de un flujo con la función de biblioteca `fseek()`. Usando a `fseek()` se puede poner el indicador de posición en cualquier lugar del archivo. El prototipo de función en STDIO.H es

```
int fseek(FILE *fp, long desplazamiento, int origen);
```

El argumento `fp` es el apuntador `FILE` asociado con el archivo. La distancia que es movido el indicador de posición es dada por `offset` en bytes. El argumento `origen` especifica el punto inicial del movimiento relativo. Puede haber tres valores para `origen`, con las constantes simbólicas definidas en IO.H, tales como se muestra en la tabla 16.2.

Tabla 16.2. Posibles valores de origen para `fseek()`.

Constante	Valor	Significado
`SEEK_SET`	0	Mueve el indicador un número `offset` de bytes desde el inicio del archivo.
`SEEK_CUR`	1	Mueve el indicador un número `offset` de bytes desde su posición actual.

Constante	Valor	Significado
SEEK_END	2	Mueve el indicador un número `offset` de bytes desde el final del archivo.

La función `fseek()` regresa `0` si el indicador fue movido satisfactoriamente, o diferente de cero si sucedió algún error. El programa del listado 16.6 usa a `fseek()` para acceso aleatorio de un archivo.

Captura

Listado 16.6. Acceso aleatorio de archivo con `fseek()`.

```
1: /* Acceso al azar con fseek(). */
2:
3: #include <stdio.h>
4: #include <io.h>
5:
6: #define MAX 50
7:
8: main()
9: {
10:     FILE *fp;
11:     int data, count, array[MAX];
12:     long offset;
13:
14:     /* Inicializa el arreglo. */
15:
16:     for (count = 0; count < MAX; count++)
17:         array[count] = count * 10;
18:
19:     /* Abre un archivo binario para escritura. */
20:
21:     if ( (fp = fopen("RANDOM.DAT", "wb")) == NULL)
22:     {
23:         fprintf(stderr, "\nError opening file.");
24:         exit(1);
25:     }
26:
27:     /* Escribe el arreglo al archivo y luego lo cierra */
28:
29:     if ( (fwrite(array, sizeof(int), MAX, fp)) != MAX)
30:     {
31:         fprintf(stderr, "\nError writing data to file.");
32:         exit(1);
33:     }
34:
35:     fclose(fp);
36:
37:     /* Abre el archivo para lectura */
38:
```

Listado 16.6. continuación

```
39:     if ( (fp = fopen("RANDOM.DAT", "rb")) == NULL)
40:     {
41:         fprintf(stderr, "\nError opening file.");
42:         exit(1);
43:     }
44:
45:     /* Le pide al usuario qué elemento quiere leer. Recibe el */
46:     /* elemento y lo despliega, terminando cuando se teclea -1. */
47:
48:     while (1)
49:     {
50:         printf("\nEnter element to read, 0-%d, -1 to quit: ",MAX-1);
51:         scanf("%ld", &offset);
52:
53:         if (offset < 0)
54:             break;
55:         else if (offset > MAX-1)
56:             continue;
57:
58:         /* Mueve el indicador de posición al elemento especificado. */
59:
60:         if ( (fseek(fp, (offset*sizeof(int)), SEEK_SET)) != NULL)
61:         {
62:             fprintf(stderr, "\nError using fseek().");
63:             exit(1);
64:         }
65:
66:         /* Lee un solo entero. */
67:
68:         fread(&data, sizeof(int), 1, fp);
69:
70:         printf("\nElement %ld has value %d.", offset, data);
71:     }
72:
73:     fclose(fp);
74: }
```

Salida

```
Enter element to read, 0-49, -1 to quit: 5

Element 5 has value 50.
Enter element to read, 0-49, -1 to quit: 6

Element 6 has value 60.
Enter element to read, 0-49, -1 to quit: 49

Element 49 has value 490.
Enter element to read, 0-49, -1 to quit: 1
```

```
Element 1 has value 10.
Enter element to read, 0-49, -1 to quit: 0

Element 0 has value 0.
Enter element to read, 0-49, -1 to quit: -1
```

Análisis

Las líneas 14 a 35 son similares al programa anterior. Las líneas 16 y 17 inicializan un arreglo, llamado data, con 50 valores tipo `int`. Luego el arreglo es escrito a un archivo binario, llamado RANDOM.DAT. Se sabe que es binario debido a que el archivo fue abierto con modo `"wb"` en la línea 21.

La línea 39 vuelve a abrir el archivo en modo de lectura binaria antes de continuar con un ciclo `while` infinito. El ciclo `while` le pide al usuario que dé el número de un elemento de arreglo que quiera leer. Observe que las líneas 53 a 56 revisan para ver si el elemento pedido se encuentra dentro del rango del archivo. ¿Le permite el C que lea un elemento que esté más allá del fin de archivo? Sí. De forma similar al ir más allá del fin de un arreglo con valores, el C también le permite leer más allá del fin de archivo. Si se lee más allá del final (o antes del principio) los resultados son impredecibles. Siempre es mejor revisar lo que se está haciendo (tal como lo hacen las líneas 53 a 56 en este listado).

Después de recibir el elemento que se ha de buscar, la línea 60 brinca al desplazamiento adecuado con una llamada a `fseek()`. Debido a que se usa `SEEK_SET`, la búsqueda se realiza a partir del principio del archivo. Observe que la distancia en el archivo no es solamente `offset`, sino `offset` multiplicado por el tamaño de los elementos que están siendo leídos. Luego la línea 68 lee el valor y la línea 70 la imprime.

Detección del fin de archivo

Unas veces se sabe exactamente qué tan largo es un archivo, por lo que no hay necesidad de detectar el fin de archivo. Por ejemplo, si se usó `fwrite()` para guardar un arreglo entero de 100 elementos, se sabe que el archivo es de 200 bytes de largo. Sin embargo, en otras ocasiones no se sabe qué tan largo es el archivo, pero todavía se quieren leer datos del archivo comenzando en el principio y avanzando hasta el fin. Hay dos maneras para detectar el fin de archivo.

Cuando se lee un archivo de modo texto, carácter por carácter, se puede revisar buscando el carácter EOF. La constante simbólica `EOF` está definida en STDIO.H como -1, un valor que nunca se usa por un carácter "real". Cuando una función de entrada de carácter lee `EOF` de un flujo en modo texto, se puede estar seguro de que se ha llegado al final del archivo. Por ejemplo, se podría escribir

```
while ( (c = fgetc( fp)) != EOF )
```

Con un flujo en modo binario no se puede detectar el fin de archivo revisando por -1, debido a que un byte de datos de un flujo binario puede tener ese valor, lo que daría como resultado un final de la entrada prematuro. En vez de ello, se puede usar la función de biblioteca `feof()` (que puede ser usada tanto en archivos de modo texto como binarios)

```
int feof(FILE *fp);
```

El argumento `fp` es el apuntador `FILE`, regresado por `fopen()` cuando el archivo fue abierto. La función `feof()` regresa `0` si no se ha llegado al fin de archivo de `fp`, o diferente de 0 cuando se ha llegado al fin de archivo. Si una llamada a `feof()` detecta el fin de archivo, no se permiten más operaciones de lectura mientras no se haya dado una llamada a `rewind()`, o a `fseek()` o se haya cerrado y vuelto a abrir el archivo.

El programa del listado 16.7 muestra el uso de `feof()`. Cuando se le pida el nombre del archivo, dé el nombre de cualquier archivo de texto, por ejemplo, alguno de sus archivos fuente de C o un archivo de encabezado, tales como STDIO.H. El programa lee el archivo de línea en línea, desplegando cada línea en `stdout` hasta que `feof()` detecta el fin de archivo.

Captura

Listado 16.7. Uso de `feof()` para detectar el fin de archivo.

```
1: /* Detección del fin de archivo. */
2:
3: #include <stdio.h>
4:
5: #define BUFSIZE 100
6:
7: main()
8: {
9:      char buf[BUFSIZE];
10:     char filename[20];
11:     FILE *fp;
12:
13:     puts("Enter name of text file to display: ");
14:     gets(filename);
15:
16:     /* Abre el archivo para lectura. */
17:     if ( (fp = fopen(filename, "r")) == NULL)
18:     {
19:         fprintf(stderr, "Error opening file.");
20:         exit(1);
21:     }
22:
23:     /* Si no ha llegado al fin de archivo lee una línea y la despliega. */
24:
25:     while ( !feof(fp) )
26:     {
```

```
27:         fgets(buf, BUFSIZE, fp);
28:         printf("%s",buf);
29:     }
30:
31:     fclose(fp);
32: }
```

Salida

```
Enter name of text file to display:
list1607.c
/* Detecting end-of-file. */

#include <stdio.h>

#define BUFSIZE 100

main()
{
    char buf[BUFSIZE];
    char filename[20];
    FILE *fp;

    puts("Enter name of text file to display: ");
    gets(filename);

    /* Open the file for reading. */
    if ( (fp = fopen(filename, "r")) == NULL)
    {
        fprintf(stderr, "Error opening file.");
        exit(1);
    }

/* If end of file not reached, read a line and display it. */

    while ( !feof(fp) )
    {
        fgets(buf, BUFSIZE, fp);
        printf("%s",buf);
    }

    fclose(fp);
}
```

Análisis

El ciclo `while` en este programa (líneas 25 a 29) es típico de un `while` usado en programas más complejos que hacen procesamiento secuencial. Mientras no se llegue al final de archivo se ejecutan las líneas dentro del enunciado `while` (líneas 27 y 28). Cuando `feof()` regresa un valor diferente de 0 el ciclo termina, el archivo es cerrado y el programa termina.

DEBE	NO DEBE
DEBE Revisar la posición dentro del archivo para que no lea más allá del final o antes del inicio del archivo.	
DEBE Usar `rewind()` o `fseek( fp, SEEK_SET, 0 )` para reajustar el indicador de posición del archivo al principio del archivo.	
DEBE Usar `feof()` para revisar el fin de archivo cuando trabaje con archivos binarios.	
	NO DEBE Usar `EOF` con archivos binarios.

Funciones para manejo de archivos

El término *manejo de archivos* se refiere al trato con archivos existentes, no la lectura ni la escritura a ellos, sino el borrado, renombrado y copiado de ellos. La biblioteca estándar del C contiene funciones para borrar y renombrar archivos, y también usted puede escribir sus propias funciones de copiado de archivos.

Borrado de un archivo

Para borrar un archivo se usa la función de biblioteca `remove()`. Su prototipo está en STDIO.H, de la manera siguiente:

```
int remove( const char *filename );
```

La variable `*filename` es un apuntador al nombre del archivo a ser borrado. (Vea la sección sobre nombres de archivo, anteriormente en este capítulo.) El archivo especificado no debe estar abierto. Si el archivo existe es borrado (de manera similar a como sucede con el uso del comando `DEL` desde la línea de comandos del DOS) y `remove()` regresa `0`. Si el archivo no existe, es de sólo lectura o sucede cualquier otro error, `remove()` regresa `-1`.

El programa corto del listado 16.8 muestra el uso de `remove()`. Tenga cuidado, ya que si se "borra" un archivo se ha ido para siempre.

Captura

Listado 16.8. Uso de la función `remove()` para borrar un archivo de disco.

```
1: /* Demuestra la función remove(). */
2:
3: #include <stdio.h>
```

```
4:
5: main()
6: {
7:     char filename[80];
8:
9:     printf("Enter the filename to delete: ");
10:     gets(filename);
11:
12:     if ( remove(filename) == 0)
13:         printf("The file %s has been deleted.", filename);
14:     else
15:         fprintf(stderr, "Error deleting the file %s.", filename);
16: }
```

Salida

```
>list1608
Enter the filename to delete: *.bak
Error deleting the file *.bak.

>list1608
Enter the filename to delete: list1414.bak
The file list1414.bak has been deleted.
```

Análisis

Este programa le pide al usuario, en la línea 9, el nombre del archivo que ha de ser borrado. Luego la línea 12 llama a `remove()` para borrar el archivo completo. Si el valor de retorno es 0, el archivo fue borrado y se despliega un mensaje que lo indica. Si el valor de retorno no es 0, es que ha sucedido algún error y el archivo no fue borrado.

Renombrado de un archivo

La función `rename()` cambia el nombre de un archivo de disco existente. El prototipo de función está en STDIO.H, de la manera siguiente:

```
int rename( const char *nomant, const char *nomnuevo );
```

Los nombres de archivos que son apuntados por `nomant` y `nomnuevo` siguen las reglas dadas anteriormente en este capítulo. La única restricción es que ambos nombres deben hacer referencia a la misma unidad de disco, ya que no se puede "renombrar" un archivo a una unidad de disco diferente. La función `rename()` regresa `0` en caso satisfactorio o `-1` cuando sucede algún error. Los errores pueden ser causados por las siguientes condiciones (entre otras):

- El archivo con `nomant` no existe.
- Ya existe un archivo con el nombre `nomnuevo`.
- Se trata de renombrar a otro disco.

El programa del listado 16.9 muestra el uso de `rename()`.



Listado 16.9. Uso de `rename()` para cambiar un archivo de disco.

```
1: /* Uso de rename() para cambiar un nombre de archivo. /*
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:      char oldname[80], newname[80];
8:
9:      printf("Enter current filename: ");
10:     gets(oldname);
11:     printf("Enter new name for file: ");
12:     gets(newname);
13:
14:     if ( rename( oldname, newname ) == 0 )
15:         printf("%s has been renamed %s.", oldname, newname);
16:     else
17:         fprintf(stderr, "An error has occurred renaming %s.",
                    oldname);
18: }
```



```
Enter current filename: list1609.exe
Enter new name for file: rname.exe
list1609.exe has been renamed rname.exe.
```



El listado 16.9 muestra qué tan poderoso puede ser el C. Con solamente 18 líneas de código este programa reemplaza un comando del DOS y es una función mucho más amigable. La línea 9 le pide el nombre del archivo a ser renombrado. La línea 11 le pide el nuevo nombre de archivo. La llamada a la función `rename()` está envuelta en un enunciado `if` en la línea 14. El enunciado `if` revisa para asegurarse que sucede adecuadamente el renombrado del archivo. Si es así, la línea 15 imprime un mensaje afirmativo y, en caso contrario, la línea 17 imprime un menssaje indicando que sucedió un error.

Copiado de un archivo

Frecuentemente es necesario hacer una copia de un archivo, un duplicado exacto con un nombre diferente (o con el mismo nombre pero en un directorio o unidad diferente). En el DOS esto se hace con el comando `COPY`. ¿Cómo se copia un archivo en C? No hay función de biblioteca disponible, por lo que necesita escribir su propia función.

Esto puede parecer algo complicado, pero es realmente bastante simple, gracias al uso de flujos del C para entrada y salida. Este es el método a emplear:

1. Abra el archivo fuente para lectura en modo binario (usando modo binario se asegura que la función pueda copiar todo tipo de archivos y no solamente los archivos de texto).
2. Abra el archivo de destino para escritura en modo binario.
3. Lea un carácter del archivo fuente. Recuerde que cuando un archivo se abre por primera vez el apuntador está al inicio del archivo, por lo que no hay necesidad de posicionar explícitamente al apuntador de archivo.
4. ¿Indica la función `feof()` que se ha llegado al final del archivo fuente? Si es así se ha terminado, y se pueden cerrar ambos archivos y regresar al programa que llama.
5. Si no se ha llegado al fin de archivo escriba el carácter al archivo de destino y regrese al paso 3.

El programa del listado 16.10 contiene una función `copy_file()` a la que le son pasados los nombres de los archivos fuente y destino, y luego ejecuta la operación de copia de acuerdo con el esquema mencionado anteriormente. Si hay algún error en la apertura de cualquier archivo, la función no trata de copiar y regresa `-1` al programa que la llama. Una vez que se completa la operación de copia el programa cierra ambos archivos y regresa `0`.

Captura

Listado 16.10. Una función que copia un archivo.

```
1: /* Copia de un archivo. */
2:
3: #include <stdio.h>
4:
5: int file_copy( char *oldname, char *newname );
6:
7: main()
8: {
9:     char source[80], destination[80];
10:
11:     /* Obtiene los nombres de origen y destino. */
12:
13:     printf("\nEnter source file: ");
14:     gets(source);
15:     printf("\nEnter destination file: ");
16:     gets(destination);
17:
18:     if ( file_copy( source, destination ) == 0 )
19:         puts("Copy operation successful");
20:     else
21:         fprintf(stderr, "Error during copy operation");
22: }
23:
```

Listado 16.10. continuación

```
24: int file_copy( char *oldname, char *newname )
25: {
26:     FILE *fold, *fnew;
27:     int c;
28:
29:     /* Abre el archivo fuente para lectura en modo binario. */
30:
31:     if ( ( fold = fopen( oldname, "rb" ) ) == NULL )
32:         return -1;
33:
34:     /* Abre el archivo de destino para escritura en modo binario. */
35:
36:     if ( ( fnew = fopen( newname, "wb" ) ) == NULL  )
37:     {
38:         fclose ( fold );
39:         return -1;
40:     }
41:
42:     /* Lee un byte a la vez de la fuente. */
43:     /* Si no ha llegado al fin de archivo, */
44:     /* escribe el byte al destino. */
45:
46:     while (1)
47:     {
48:         c = fgetc( fold );
49:
50:         if ( !feof( fold ) )
51:             fputc( c, fnew );
52:         else
53:             break;
54:     }
55:
56:     fclose ( fnew );
57:     fclose ( fold );
58:
59:     return 0;
60: }
```

Salida

```
Enter source file: list1610.c

Enter destination file: tmpfile.c
       Copy operation successful
```

Análisis La función `copy_file()` funciona perfectamente bien, permitiéndole copiar cualquier cosa, ya sea un pequeño archivo de texto o un archivo de programa enorme. Sin embargo, no tiene limitaciones. Si el archivo de destino ya existe la función lo borra sin preguntar. Un buen ejercicio de programación sería modificar a `copy_file()` para que

revise si ya existe el archivo de destino, y luego preguntarle al usuario si quiere que el archivo antiguo sea sobreescrito.

`main()`, en el listado 16.10, debe serle familiar. Es casi idéntica al listado 16.9 a excepción de la línea 18. En vez de `rename()` esta función usa copy(). Debido a que el C no tiene una función de copia, las líneas 24 a 60 crean una función de copia. Las líneas 31 a 32 abren el archivo fuente, `fold`, en modo de lectura binario. Las líneas 36 a 40 abren el archivo de destino, `fnew`, en modo de escritura binario. Observe que la línea 38 cierra el archivo fuente si hay algún error en la apertura del archivo de destino. El ciclo `while` de las líneas 45 a 54 hace el copiado actual del archivo. La línea 48 obtiene un carácter del archivo fuente, `fold`. La línea 50 revisa para ver si se ha leído el indicador de fin de archivo. Si se ha llegado al fin de archivo se ejecuta un enunciado `break` para salir del ciclo `while`. Si no se ha llegado al fin de archivo, el carácter es escrito al archivo de destino, `fnew`. Las líneas 56 a 57 cierran a los dos archivos antes de regresar a `main()`.

Uso de archivos temporales

Algunos programas utilizan uno o más archivos temporales durante la ejecución. Un *archivo temporal* es un archivo que es abierto por el programa, usado para algún objeto durante la ejecución del programa y luego borrando antes de que el programa termine. Cuando se crea un archivo temporal no importa cuál sea su nombre, debido a que va a ser borrado. Todo lo que se necesita es que se use un nombre que no esté siendo usado. La biblioteca estándar del C incluye una función, `tmpnam()`, que crea un nombre de archivo válido que no entre en conflicto con ningún archivo existente. Su prototipo en STDIO.H dice lo siguiente:

```
char *tmpnam(char *s);
```

El argumento `s` debe ser un apuntador a un buffer lo suficientemente grande para que quepa el nombre de archivo. También se le puede pasar un apuntador nulo (`NULL`), en cuyo caso el nombre temporal es guardado en un buffer interno a `tmpnam()` y la función regresa un apuntador a ese buffer. El programa del listado 16.11 muestra ambos métodos de uso de `tmpnam()` para crear nombres de archivos temporales.



Listado 16.11. Uso de `tmpnam()` para crear nombres de archivo temporales.

```
1: /* Demostración de nombres de archivo temporales. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     char buffer[10], *c;
```

Listado 16.11. continuación

```
8:
9:      /* Obtiene un nombre temporal en el buffer definido */
10:
11:     tmpnam(buffer);
12:
13:     /* Obtiene otro nombre, pero esta vez en el buffer */
14:     /* interno de la función. */
15:
16:     c = tmpnam(NULL);
17:
18:     /* Despliega los nombres. */
19:
20:     printf("Temporary name 1: %s", buffer);
21:     printf("\nTemporary name 2: %s", c);
22: }
```

Salida

```
Temporary name 1: TMP1.$$$
Temporary name 2: TMP2.$$$
```

Análisis Este programa sólo genera e imprime los nombres temporales. La línea 11 guarda un nombre temporal en el arreglo de caracteres, `buffer`. La línea 16 asigna a `c` el apuntador de carácter al nombre regresado por `tmpnam()`. El programa podría haber usado el nombre generado para abrir el archivo temporal, y luego borrar el archivo antes de que termine la ejecución del programa.

```
char tempname[80];
FILE *tmpfile;
tmpnam(tempname);
tmpfile = fopen(tempname, "w");  /* Use appropriate mode */

fclose(tmpfile);
remove(tempname);
```

DEBE NO DEBE

NO DEBE Borrar un archivo que vaya a necesitar nuevamente.

NO DEBE Tratar de renombrar archivos con diferentes unidades de disco.

NO DEBE Olvidar el borrar los archivos temporales que cree. No son borrados automáticamente.

Resumen

En este capítulo aprendió la manera en que los programas del C pueden usar archivos de disco. El C trata a los archivos de disco como un flujo, una secuencia de caracteres similar a los flujos predefinidos sobre los que aprendió en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". Un flujo asociado con un archivo de disco debe ser abierto antes de que pueda ser usado, y debe ser cerrado después de su uso. Un flujo de archivos de disco puede ser abierto en modo texto o binario.

Una vez que ha sido abierto un archivo de disco se pueden leer datos del archivo hacia el programa, escribir datos del programa hacia el archivo o ambas cosas. Hay tres tipos generales de E/S de archivo: formateada, de carácter y directa. Cada tipo de E/S es la mejor indicada para determinados tipos de tareas de almacenamiento y recuperación de datos.

Cada archivo de disco abierto tiene un indicador de posición de archivo asociado con él. Este indicador especifica la posición en el archivo, medida como la cantidad de bytes a partir del inicio del archivo, donde suceden las siguientes operaciones de lectura y escritura. Con algunos tipos de acceso a archivo el indicador de posición es actualizado automáticamente y no tiene uno porqué preocuparse de ello. Para el acceso aleatorio de archivos, la biblioteca estándar del C proporciona funciones para manejar el indicador de posición.

Por último, el C proporciona algunas funciones rudimentarias para el manejo de archivos, que le permiten borrar y renombrar archivos de disco. En este capítulo, usted desarrolló su propia función para el copiado de archivos.

Preguntas y respuestas

1. ¿Puedo usar unidades y rutas con nombres de archivos cuando use `erase()`, `rename()`, `fopen()` y las otras funciones de archivo?

   Sí. Puede usar nombres completos de archivo con rutas y unidades o solamente el nombre de archivo. Si usa solamente el nombre de archivo, la función busca el archivo en el directorio actual. Recuerde que cuando use diagonales invertidas, necesita usar secuencias de escape.

2. ¿Puedo leer más allá del fin de archivo?

   Sí. También puede leer antes del inicio del archivo. Los resultados de estas lecturas pueden ser desastrosos. La lectura de archivos es similar al manejo de arreglos. Se están viendo desplazamientos dentro de memoria. Si se está usando `fseek()` se debe revisar para asegurarse que no se va más allá del fin de archivo.

3. ¿Qué pasa si no cierro un archivo?

   Es una buena práctica de programación el cerrar cualquier archivo que se abra. Por omisión, el archivo debe ser cerrado cuando el programa termine, más sin embargo, no debe confiarse en ello. Si el archivo no está cerrado no será capaz de accesarlo posteriormente, debido a que el sistema operativo pensará que el archivo está todavía en uso.

4. ¿Qué tantos archivos puedo tener abiertos a la vez?

   Esta pregunta no puede ser respondida con un simple número. Los límites sobre la cantidad de archivos que pueden ser abiertos se basan en variables puestas en el sistema operativo. En los sistemas de DOS una variable de ambiente, llamada FILES, determina la cantidad de archivos que pueden ser abiertos (esta variable también incluye los programas que están ejecutando). Consulte los manuales de su sistema operativo para mayor información.

5. ¿Puedo leer un archivo secuencialmente con funciones de acceso aleatorio?

   Cuando se lee un archivo secuencialmente no hay necesidad de usar funciones tales como `fseek()`. Debido a que el apuntador de archivo es dejado en la última posición que ha ocupado, siempre se encuentra donde se desea para lectura secuencial. Se puede usar `fseek()` para leer un archivo secuencialmente, más, sin embargo, no se gana nada.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Cuál es la diferencia entre un flujo de modo texto y un flujo binario?
2. ¿Qué debe hacer un programa antes de que pueda accesar un archivo de disco?
3. Cuando se abre un archivo con `fopen()`, ¿qué información se debe especificar y qué es lo que regresa la función?
4. ¿Cuáles son los tres métodos generales de acceso a archivos?
5. ¿Cuáles son los dos métodos generales para la lectura de la información de un archivo?
6. ¿Cuál es el valor de `EOF`?

7. ¿Cuándo se usa EOF?
8. ¿Cómo se detecta el fin de archivo en modos de texto y binario?
9. ¿Qué es el indicador de posición de archivo y cómo puede ser modificado?
10. Cuando se abre un archivo por primera vez, ¿dónde está apuntado el indicador de posición de archivo? (Si no está seguro, vea el listado 16.5.)

Ejercicios

1. Escriba el código para cerrar todos los flujos de archivo.
2. Muestre dos maneras diferentes para reajustar el apuntador de posición de archivo al inicio del archivo.
3. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
FILE *fp;
int c;

if ( ( fp = fopen( oldname, "rb" ) ) == NULL )
    return -1;

while (( c = fgetc( fp)) != EOF )
    fprintf( stdout, "%c", c );

fclose ( fp );
```

Debido a que hay muchas soluciones posibles, no se proporcionan respuestas para los siguientes ejercicios.

4. Escriba un programa que despliegue un archivo en la pantalla.
5. Escriba un programa que abra un archivo y lo imprima en la impresora (stdprn). El programa debe imprimir solamente 55 líneas por página.
6. Modifique el programa del ejercicio cinco para imprimir encabezados en cada página. El encabezado debe contener el nombre de archivo y el número de página.
7. Escriba un programa que abra un archivo y cuente la cantidad de caracteres. Cuando acabe, el programa debe imprimir la cantidad de caracteres.
8. Escriba un programa que abra un archivo de texto existente y lo copie a un archivo de texto nuevo, cambiando todas las letras minúsculas a mayúsculas y dejando sin modificación a los demás caracteres.

9. Escriba un programa que abra cualquier archivo de disco, lo lea en bloques de 128 bytes y despliegue el contenido de cada bloque en la pantalla en formato hexadecimal y ASCII.

10. Escriba una función que abra un archivo temporal nuevo con un modo especificado. Todos los archivos temporales creados por esta función deben ser cerrados y borrados automáticamente cuando termine el programa. (Consejo: Use la función de biblioteca `atexit()`.)

DIA 17

Manipulación de cadenas

Los datos de texto, que el C guarda en cadenas, son parte importante de muchos programas. Hasta ahora usted ha aprendido la manera de dar entrada y salida a las cadenas y la manera en que el programa las guarda. El C ofrece también una variedad de funciones para otros tipos de manejos de cadenas. Hoy aprenderá

- Cómo determinar la longitud de una cadena.
- Cómo copiar y unir cadenas.
- Funciones que comparan cadenas.
- Cómo buscar cadenas.
- Cómo convertir cadenas.
- Cómo hacer preguntas sobre caracteres.

Longitud y almacenamiento de cadenas

Tal vez recuerde de los capítulos anteriores que en los programas en C una cadena es una secuencia de caracteres, que tiene su inicio indicado por un apuntador y su final indicado por el carácter nulo, `\0`. A veces se necesita saber la longitud de una cadena (la cantidad de caracteres entre el inicio y el final de la cadena). Esto se logra con la función de biblioteca `strlen()`. Su prototipo en STRING.H es

```
size_t strlen(char *str);
```

Tal vez se pregunte qué significa el tipo de retorno `size_t`. Este está definido en STRING.H como `unsigned`, por lo que la función `strlen()` regresa un entero sin signo. El tipo `size_t` se usa con muchas funciones de cadenas. Recuerde que sencillamente significa `unsigned`.

El argumento pasado a `strlen` es un apuntador a la cadena de la cual se quiere saber su longitud. La función `strlen()` regresa la cantidad de caracteres entre `str` y el siguiente carácter nulo, sin contar el carácter nulo. El programa del listado 17.1 muestra a `strlen()`.

Listado 17.1. Uso de la función `strlen()` para determinar la longitud de una cadena.

```
1: /* Uso de la función strlen(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
```

```
6: main()
7: {
8:     size_t length;
9:     char buf[80];
10:
11:     while (1)
12:     {
13:         puts("\nEnter a line of text; a blank line / terminates.");
14:         gets(buf);
15:
16:         length = strlen(buf);
17:
18:         if (length != 0)
19:             printf("\nThat line is %u characters long.",
                    length);
20:         else
21:             break;
22:     }
23: }
```


Salida

```
Enter a line of text; a blank line terminates.
Just do it!

That line is 11 characters long.
Enter a line of text; a blank line terminates.
```

Análisis

Este programa hace más que mostrar el uso de `strlen()`. Las líneas 13 y 14 despliegan un mensaje y obtienen una cadena llamada `buf`. La línea 16 usa `strlen()` para asignar la longitud de `buf` a la variable `lenght`. La línea 18 revisa para ver si la cadena está en blanco, buscando una longitud de cero. Si la cadena no está en blanco, la línea 19 imprime el tamaño de la cadena.

Copia de cadenas

La biblioteca del C tiene tres funciones para la copia de cadenas. Debido a la forma en que el C maneja a las cadenas no se puede asignar una cadena a otra, como se puede hacer en otros lenguajes de computadora. Se debe copiar la cadena de origen, de su posición en memoria a la posición de memoria de la cadena de destino. Las funciones para la copia de cadenas son `strcpy()`, `strncpy()` y `strdup()`. Todas las funciones de copiado de cadenas requieren el archivo de encabezado STRING.H.

La función *strcpy()*

La función de biblioteca `strcpy()` copia una cadena completa a otra posición de memoria. Su prototipo es

```
char *strcpy( char *destino, char *origen );
```

La función `strcpy()` copia la cadena (incluido el carácter nulo terminal `\0`) apuntada por `origen` a la ubicación apuntada por `destino`. El valor de retorno es un apuntador a la nueva cadena, `destino`.

Cuando use `strcpy()` primero debe asignar espacio de almacenamiento para la cadena de destino. La función no tiene manera de saber si `destino` apunta a algún espacio asignado. Si el espacio no ha sido asignado, la función sobreescribe la cantidad de bytes de memoria que serían determinados por `strlen(origen)`, comenzando en `destino`. El uso de `strcpy()` es ilustrado en el listado 17.2.

Captura

Listado 17.2. Antes de usar `strcpy()` se debe asignar espacio de almacenamiento para la cadena de destino.

```
1: /* Demuestra a strcpy(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: char source[] = "The source string.";
7:
8: main()
9: {
10:     char dest1[80];
11:     char *dest2, *dest3;
12:
13:     printf("\nsource: %s", source );
14:
15:     /* La copia a dest1 es correcta, debido a que dest1 apunta a */
16:     /* 80 bytes de espacio asignado. */
17:
18:     strcpy(dest1, source);
19:     printf("\ndest1:  %s", dest1);
20:
21:     /* Para copiar a dest2 se debe asignar espacio. */
22:
23:     dest2 = (char *)malloc(strlen(source) +1);
24:     strcpy(dest2, source);
25:     printf("\ndest2:  %s", dest2);
26:
27:     /* La copia sin haber asignado el destino está prohibida. */
28:     /* Lo siguiente puede causar serios problemas. */
29:
30:     /* strcpy(dest3, source); */
31: }
```

Salida

```
source: The source string.
dest1:  The source string.
dest2:  The source string.
```

Análisis

Este programa muestra la copia de cadenas, tanto a arreglos de carácter, como `dest1` (declarado en la línea 10), como a apuntadores a carácter, como `dest2` (declarado junto con `dest3` en la línea 11). La línea 13 imprime la cadena de origen original. Esta cadena es copiada luego a `dest1` con `strcpy()` en la línea 18. La línea 24 copia la cadena de origen a `dest2`. Tanto `dest1` como `dest2` son impresas para mostrar que la función ejecutó satisfactoriamente. Observe que la línea 23 asigna la cantidad adecuada de espacio para `dest2` con la función `malloc()`. Si se copia una cadena a un apuntador a carácter al que no le ha sido asignada memoria, se obtienen resultados impredecibles.

La función *strncpy()*

La función `strncpy()` es similar a `strcpy()`, a excepción de que `strncpy()` le permite especificar qué tantos caracteres han de copiarse. Su prototipo es

```
char *strncpy(char *destino, char *origen, size_t n);
```

Los argumentos `destino` y `origen` son apuntadores a las cadenas de destino y origen. La función copia, a lo máximo, los primeros `n` caracteres del origen hacia el destino. Si el `origen` es más corto que `n` caracteres, se añaden los suficientes caracteres nulos al final del `origen`, para completar un total de `n` caracteres copiados a `destino`. Si `origen` es más grande que `n` caracteres no se añade `\0` al destino. El valor de retorno de la función es la dirección de `destino`.

El programa en el listado 17.3 muestra el uso de `strncpy()`.

Captura

Listado 17.3. La función `strncpy`.

```
1: / Uso de la función strncpy(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: char dest[] = ".........................";
7: char source[] = "abcdefghijklmnopqrstuvwxyz";
8:
9: main()
10: {
11:     size_t n;
12:
13:     while (1)
14:     {
```

Listado 17.3. continuación

```
15:         puts("Enter the number of characters to copy (1-26)");
16:         scanf("%d", &n);
17:
18:         if (n > 0 && n < 27)
19:             break;
20:     }
21:
22:     printf("\nBefore strncpy destination = %s", dest);
23:
24:     strncpy(dest, source, n);
25:
26:     printf("\nAfter strncpy destination = %s", dest);
27: }
```

Salida

```
Enter the number of characters to copy (1-26)
15

Before strncpy destination = ........................
After strncpy destination = abcdefghijklmno...........
```

Análisis

Este programa muestra la función de `strncpy()` y una manera efectiva para asegurarse de que sólo se da la información correcta. Las líneas 13 a 20 contienen un ciclo `while`, que le pide al usuario un número del 1 al 26. El ciclo continúa hasta que es dado un valor válido. Una vez que se da un número del 1 al 26 la línea 22 imprime el valor de `dest`, la línea 24 copia la cantidad de caracteres indicados por el usuario y la línea 26 imprime el resultado.

La función *strdup()*

La función de biblioteca `strdup()` es similar a `strcpy()`, a excepción de que `strdup()` ejecuta su propia asignación de memoria para la cadena de destino con una llamada a `malloc()`. De hecho, hace lo que usted hizo en el listado 17.2, la asignación de espacio con `malloc()` y luego la llamada a `strcpy()`. El prototipo para `strdup()` es

```
char *strdup( char *origen );
```

El argumento origen es un apuntador a la cadena de origen. La función regresa un apuntador a la cadena de destino, el espacio asignado por `malloc()`, o `NULL` si la memoria necesaria no pudo ser asignada. El listado 17.4 muestra el uso de `strdup()`. Observe que `strdup()` no es una función estándar ANSI. Está incluida en las bibliotecas del C de Microsoft, Borland y Zortech, pero tal vez no esté presente (o sea diferente) en otros compiladores C.

Listado 17.4. Uso de `strdup()` para copiar una cadena con asignación automática de memoria.

```
1: /* La función strdup(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: char source[] = "The source string.";
7:
8: main()
9: {
10:     char *dest;
11:
12:     if ( (dest = strdup(source)) == NULL)
13:     {
14:         fprintf(stderr, "Error allocating memory.");
15:         exit(1);
16:     }
17:
18:     printf("The destination = %s", dest);
19: }
```

Salida

```
The destination = The source string.
```

Análisis

En este listado `strdup()` asigna la cantidad adecuada de memoria para `dest`. Luego hace una copia de la cadena que se le pasa, `origen`. La línea 18 imprime la cadena duplicada.

Concatenación de cadenas

Si no le es familiar el término *concatenación,* tal vez se pregunte "¿qué es?" y "¿es válido"? Bueno, significa la unión de dos cadenas, pegar una cadena a la cola de la otra, y sí es válido. La biblioteca estándar del C contiene dos funciones para la concatenación de cadenas, `strcat()` y `strncat()`, requiriendo ambas el archivo de encabezado STRING.H.

La función *strcat()*

El prototipo de `strcat()` es

```
char *strcat(char *str1, char *str2);
```

La función añade una copia de `str2` al final de `str1`, moviendo el carácter nulo terminal al final de la nueva cadena. Se debe asignar suficiente espacio para `str1` a fin de que quepa la cadena resultante. El valor de retorno de `strcat()` es un apuntador a `str1`. El programa del listado 17.5 muestra a `strcat()`.



Listado 17.5. Uso de `strcat()` para concatenar cadenas.

```
1: /* La función strcat(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: char str1[27] = "a";
7: char str2[2];
8:
9: main()
10: {
11:     int n;
12:
13:     /* Pone un carácter nulo al final de str2[]. */
14:
15:     str2[1] = '\0';
16:
17:     for (n = 98; n < 123; n++)
18:     {
19:         str2[0] = n;
20:         strcat(str1, str2);
21:         puts(str1);
22:     }
23: }
```




```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
```

```
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

Análisis Los códigos ASCII para las letras de la *b* a la *z* son del 98 al 122. Este programa usa estos códigos ASCII en su demostración de `strcat()`. El ciclo `for` de las líneas 17 a 22 asigna estos valores a su vez a `str2[0]`. Debido a que `str2[1]` ya es el carácter nulo (línea 15), el efecto es asignar las cadenas `"b"`, `"c"`, etc. a `str2`. Cada una de estas cadenas es concatenada con `str1` (línea 20), y `str1` es desplegado en la pantalla (línea 21).

La función *strncat()*

La función de biblioteca `strncat()` también ejecuta la concatenación de cadenas, pero le permite especificar qué tantos caracteres de la cadena de origen son añadidos al final de la cadena de destino. El prototipo es

```
char *strncat(char *str1, char *str2, size_t n);
```

Si `str2` contiene más de `n` caracteres, son añadidos al final de `str1` los primeros `n` caracteres de `str2`. Si `str2` contiene menos de `n` caracteres, `str2` es añadido completamente al final de `str1`. En ambos casos se añade un carácter nulo terminal. Se debe asignar suficiente espacio para `str1` a fin de que quepa la cadena resultante. La función regresa un apuntador a `str1`. El programa del listado 17.6 usa `strncat()` para producir la misma salida que el listado 17.5.

Captura **Listado 17.6. Uso de la función `strncat()` para concatenar cadenas.**

```
1: /* La función strncat(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: char str2[] = "abcdefghijklmnopqrstuvwxyz";
7:
8: main()
9: {
10:     char str1[27];
11:     int n;
12:
13:     for (n=1; n < 27; n++)
```

Listado 17.6. continuación

```
14:     {
15:         strcpy(str1, "");
16:         strncat(str1, str2, n);
17:         puts(str1);
18:     }
19: }
```

Salida

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

Análisis

Tal vez se pregunte para qué sirve la línea 15, `strcpy(str1, "");`. Esta línea copia a `str1` una cadena vacía, que consiste solamente en el carácter nulo. El resultado es que el primer carácter en `str1, str1[0]`, es 0 (el carácter nulo). Lo mismo podría lograrse con los enunciados `str1[0] = 0;` o `str1[0] = '\0';`.

Comparación de cadenas

Las cadenas son comparadas para determinar si son iguales o diferentes. Si son diferentes, una cadena es "mayor que" o "menor que" otra. La determinación de "mayor" y "menor"

se hace con los códigos ASCII de los caracteres. En el caso de las letras es equivalente al orden alfabético. La biblioteca del C contiene funciones para tres tipos de comparaciones de cadenas: comparación de dos cadenas completas, comparación de dos cadenas sin tomar en cuenta mayúsculas y minúsculas y comparación de una determinada cantidad de caracteres en ambas cadenas. Debido a que estas tres funciones no están dentro del estándar ANSI, no todos los compiladores las tratan de la misma manera.

Comparación de dos cadenas

La función `strcmp()` compara dos cadenas, carácter por carácter. Su prototipo es

```
int strcmp(char *str1, char *str2);
```

Los argumentos `str1` y `str2` son apuntadores a las cadenas a comparar. Los valores de retorno de la función se dan en la tabla 17.1. El programa del listado 17.7 muestra a `strcmp()`.

Tabla 17.1. Los valores regresados por `strcmp()`.

Valor de retorno	Significado
< 0	`str1` es menor que `str2`.
0	`str1` es igual a `str2`.
> 0	`str1` es mayor que `str2`.

Captura

Listado 17.7. Uso de `strcmp()` para comparar cadenas.

```
1: /* La función strncmp(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8:     char str1[80], str2[80];
9:     int x;
10:
11:     while (1)
12:     {
13:
14:        /* Recibe dos cadenas. */
15:
```

Listado 17.7. continuación

```
16:         printf("\n\nInput the first string, a blank to exit: ");
17:         gets(str1);
18:
19:         if ( strlen(str1) == 0 )
20:             break;
21:
22:         printf("\nInput the second string: ");
23:         gets(str2);
24:
25:         /* Las compara y despliega el resultado. */
26:
27:         x = strcmp(str1, str2);
28:
29:         printf("\nstrcmp(%s,%s) returns %d", str1, str2, x);
30:     }
31:}
```

Salida

```
Input the first string, a blank to exit: First string
Input the second string: Second string
strcmp(First string,Second string) returns -13
Input the first string, a blank to exit: test string
Input the second string: test string
strcmp(test string,test string) returns 0
Input the first string, a blank to exit: zebra
Input the second string: aardvark
strcmp(zebra,aardvark) returns 25
Input the first string, a blank to exit:
```

Análisis

El programa del listado 17.7 muestra a `strcmp()`, pidiéndole al usuario dos cadenas (líneas 16, 17, 22 y 23) y desplegando el resultado regresado por `strcmp()` en la línea 29. Experimente con este programa para ambientarse en la manera en que `strcmp()` compara las cadenas. Trate de dar dos cadenas que son idénticas a excepción de mayúsculas y minúsculas, como "MORENO" y "moreno". Verá que `strcmp()` toma en cuenta a las mayúsculas, lo que significa que el programa considera diferentes a las letras mayúsculas y minúsculas.

Comparación de dos cadenas: ignorando mayúsculas y minúsculas

Las funciones de comparación de cadenas que no toman en cuenta a las mayúsculas y las minúsculas funcionan en la misma forma a la función `strcmp()` que sí las toma. La mayoría de los compiladores tienen su propia función de comparación que toma en cuenta a las mayúsculas y minúsculas. Zortech usa la función de biblioteca `strcmpl()`. Microsoft usa una función llamada `_stricmp()`. Borland tiene dos funciones, `strcmpi()` y `stricmp()`. Necesita revisar el manual de consulta de la biblioteca para determinar cuál función es la adecuada para su compilador. Cuando use una función que no tome en cuenta mayúsculas y minúsculas las cadenas `"moreno"` y `"MORENO"` resultan iguales en la comparación. Modifique la línea 27 del listado 17.7 para usar la función de comparación adecuada para su compilador.

Comparación parcial de cadenas

La función de biblioteca `strncmp()` compara una cantidad especificada de caracteres de una cadena con otra cadena. Su prototipo es:

```
int strncmp(char *str1, char *str2, size_t n);
```

La función `strncmp()` compara `n` caracteres de `str2` contra `str1`. La comparación se realiza hasta que han sido comparados `n` caracteres o se ha llegado al final de `str1`. El método de comparación y los valores de retorno son los mismos que para `strcmp()`. La comparación sí toma en cuenta las mayúsculas y las minúsculas. El listado 17.8 muestra a `strncmp()`.

Captura

Listado 17.8. Comparación de partes de cadenas con `strncmp()`.

```
1: /* La función strncmp(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: char str1[] = "The first string.";
7: char str2[] = "The second string.";
8:
9: main()
10: {
11:     size_t n, x;
12:
13:     puts(str1);
14:     puts(str2);
15:
16:     while (1)
17:     {
18:         puts("\n\nEnter number of characters to compare, 0 to \
            exit.");
```

Listado 17.8. continuación

```
19:         scanf("%d", &n);
20:
21:         if (n <= 0)
22:             break;
23:
24:         x = strncmp(str1, str2, n);
25:
26:         printf("\nComparing %d characters, strncmp() returns \
                %d.",
                n, x);
27:
28:     }
29: }
```

Salida

```
The first string.
The second string.
Enter number of characters to compare, 0 to exit.
3
Comparing 3 characters, strncmp() returns 0.
Enter number of characters to compare, 0 to exit.
6
Comparing 6 characters, strncmp() returns -13.
Enter number of characters to compare, 0 to exit.
0
```

Análisis

Este programa compara dos cadenas, definidas en las líneas 6 y 7. Las líneas 13 y 14 imprimen las cadenas en la pantalla, para que el usuario pueda ver cuáles son. El programa ejecuta un ciclo `while`, en las líneas 16 a 28, para que se puedan realizar varias comparaciones. En las líneas 18 y 19 se pide la cantidad de caracteres que se han de comparar, y si el usuario pide que se comparen cero caracteres el programa termina el ciclo en la línea 22. En caso contrario, se ejecuta `strncmp()` en la línea 24 y el resultado es impreso con la línea 26.

Búsqueda en cadenas

La biblioteca del C contiene varias funciones para la búsqueda de cadenas. La búsqueda determina si una cadena se encuentra dentro de otra cadena y, en caso de estar, la posición que ocupa. Hay seis funciones de búsqueda de cadenas, requiriendo todas el archivo de encabezado STRING.H.

La función *strchr()*

La función `strchr()` busca la primera aparición de un carácter específico en una cadena. El prototipo es:

```
char *strchr(char *str, int ch);
```

La función `strchr()` busca en `str` hasta que encuentra el carácter `ch` o el carácter nulo terminal. Si encuentra a `ch` regresa un apuntador a él. En caso contrario, regresa `NULL`.

Cuando `strchr()` encuentra el carácter, regresa un apuntador a ese carácter. Sabiendo que `str` es un apuntador al primer carácter de la cadena, se puede obtener la posición del carácter encontrado restando `str` del valor del apuntador regresado por `strchr()`. El programa del listado 17.9 ilustra esto. Recuerde que el primer carácter de una cadena está en la posición 0.



Listado 17.9. Uso de `strchr()` para buscar en una cadena un carácter solo.

```
1: /* Búsqueda de un solo carácter con strchr(). /*
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8:     char *loc, buf[80];
9:     int ch;
10:
11:     /* Recibe la cadena y el carácter. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf);
15:     printf("Enter the character to search for: ");
16:     ch = getchar();
17:
18:     /* Ejecuta la búsqueda. */
19:
20:     loc = strchr(buf, ch);
21:
22:     if ( loc == NULL )
23:         printf("The character %c was not found.", ch);
24:     else
25:         printf("The character %c was found at position %d.",
26:                 ch, loc-buf);
27: }
```




```
Enter the string to be searched: How now Brown Cow?
Enter the character to search for: C
The character C was found at position 14.
```



Este programa usa `strchr()`, en la línea 20, para buscar un carácter dentro de una cadena. `strchr()` regresa un apuntador a la posición donde se encuentra por primera vez el carácter, o `NULL` si el carácter no se encuentra. La línea 22 revisa para ver si el valor de loc es `NULL` e imprime un mensaje adecuado.

La función *strrchr()*

La función de biblioteca `strrchr()` es idéntica a `strchr()`, a excepción de que busca en una cadena la última aparición de un carácter especificado. Su prototipo es

```
char *strrchr(char *str, int ch);
```

La función `strrchr()` regresa un apuntador a la última aparición de `ch` en `str` y NULL si no existe. Para ver la manera en que funciona la función modifique la línea 20 del listado 17.9, para usar `strrchr()` en vez de `strchr()`.

La función *strcspn()*

La función de biblioteca `strcspn()` busca en una cadena la primera aparición de cualquiera de los caracteres que se encuentren en una segunda cadena. Su prototipo es

```
size_t strcspn(char *str1, char *str2);
```

La función `strcspn()` comienza la búsqueda en el primer carácter de `str1`, revisando para ver si encuentra alguno de los caracteres contenidos en `str2`. Si lo encuentra regresa el desplazamiento, a partir del inicio de `str1`, donde se halla el carácter encontrado. Si no encuentra nada, `strcspn()` regresa el valor de la longitud de `str1`, obtenido mediante `strlen(str1)`. Esto indica que lo primero que encontró fue el carácter nulo terminal de la cadena. El programa del listado 17.10 muestra la manera de usar `strcspn()`.

Captura

Listado 17.10. Búsqueda de un juego de caracteres con `strcspn()`.

```
1: /* Búsqueda con strcspn(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8:     char  buf1[80], buf2[80];
9:     size_t loc;
10:
11:     /* Recibe las cadenas. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf1);
15:     printf("Enter the string containing target characters: ");
16:     gets(buf2);
17:
18:     /* Ejecuta la búsqueda. */
19:
```

```
20:     loc = strcspn(buf1, buf2);
21:
22:     if ( loc ==  strlen(buf1) )
23:         printf("No match was found.");
24:     else
25:         printf("The first match was found at position %d.",
                loc);
26: }
```

Salida

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: Cow
The first match was found at position 14.
```

Análisis

Este listado es similar al listado 17.10. En vez de buscar la primera aparición de un solo carácter busca la primera aparición de cualquiera de los caracteres de la segunda cadena. El programa llama a `strcspn()`, en la línea 20, con `buf1` y `buf2`. Si alguno de los caracteres de `buf2` está en `buf1`, `strcspn()` regresa el desplazamiento, a partir del inicio de `buf1`, de la posición de la primera aparición. La línea 22 revisa el valor de retorno para determinar si es cero. Si el valor es cero significa que no se encontró ningún carácter y la línea 23 despliega un mensaje adecuado. Si se encuentra un valor, se despliega un mensaje indicando la posición del carácter en la cadena.

La función *strspn()*

Esta función está relacionada con la anterior, `strcspn()`, como lo explica el siguiente párrafo. Su prototipo es

```
size_t strspn(char *str1, char *str2);
```

La función `strspn()` revisa a `str1`, comparándolo carácter por carácter con los caracteres que están contenidos en `str2`. Regresa la posición del primer carácter de `str1` que no concuerde con un carácter de `str2`. En otras palabras, `strspn()` regresa la longitud del segmento inicial de `str1` que consiste por completo en los caracteres que se encuentran en `str2`. El programa del listado 17.11 muestra a `strspn()`.

Captura

Listado 17.11. Búsqueda con `strspn()` del primer carácter que no concuerda.

```
1: /* Búsqueda con strspn(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
```

Listado 17.11. continuación

```
8:     char  buf1[80], buf2[80];
9:     size_t loc;
10:
11:     /* Recibe las cadenas. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf1);
15:     printf("Enter the string containing target characters: ");
16:     gets(buf2);
17:
18:     /* Ejecuta la búsqueda. */
19:
20:     loc = strspn(buf1, buf2);
21:
22:     if ( loc ==  NULL )
23:         printf("No match was found.");
24:     else
25:         printf("Characters match up to position %d.", loc-1);
26:
27: }
```

Salida

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: How now what?
Characters match up to position 7.
```

Análisis

Este programa es similar al ejemplo anterior, a excepción de que llama a `strspn()`, en vez de `strcspn()`, en la línea 20. La función regresa el desplazamiento en `buf1` donde se encuentra el primer carácter que no está en `buf2`. Las líneas 22 a 25 evalúan el valor de retorno e imprimen un mensaje adecuado.

La función *strpbrk()*

La función de biblioteca `strpbrk()` es similar a `strcspn()`, buscando en una cadena la primera aparición de cualquier carácter contenido en otra cadena. Su diferencia es que no incluye en la búsqueda al carácter nulo terminal. El prototipo de función es:

```
char *strpbrk(char *str1, char *str2);
```

La función `strpbrk()` regresa un apuntador al primer carácter de `str1` que concuerda con cualquiera de los caracteres de `str2`. Si no encuentra concordancia la función regresa `NULL`. Como se explicó anteriormente para la función `strchr()`, se puede obtener el desplazamiento de la primera concordancia en `str1` restando el apuntador `str1` del apuntador regresado por `strpbrk()` (por supuesto, cuando no es `NULL`). Por ejemplo, reemplace a `strcspn()` en la línea 20 del listado 17.10 con `strpbrk()`.

La función *strstr()*

La última, y tal vez más útil de las funciones de búsqueda de cadenas del C, es `strstr()`. Esta función busca la primera aparición de una cadena dentro de otra. Su prototipo es

```
char *strstr(char *str1, char *str2);
```

La función `strstr()` regresa un apuntador a la primera aparición de `str2` dentro de `str1`. Si no la encuentra, la función regresa `NULL`. Si la longitud de `str2` es cero la función regresa `str1`. Cuando `strstr()` encuentra una concordancia, se puede obtener el desplazamiento de `str2` dentro de `str1` mediante la resta de apuntadores, como se explicó anteriormente para `strchr()`. El procedimiento de búsqueda que utiliza `strstr()` sí toma en cuenta las mayúsculas y las minúsculas. El programa del listado 17.12 muestra la manera de usar `strstr()`.

Captura

Listado 17.12. Uso de `strstr()` para buscar una cadena dentro de otra.

```
1: /* Búsqueda con strstr(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8:     char *loc, buf1[80], buf2[80];
9:
10:     /* Recibe las cadenas. */
11:
12:     printf("Enter the string to be searched: ");
13:     gets(buf1);
14:     printf("Enter the target string: ");
15:     gets(buf2);
16:
17:     /* Ejecuta la búsqueda. */
18:
19:     loc = strstr(buf1, buf2);
20:
21:     if ( loc ==  NULL )
22:         printf("No match was found.");
23:     else
24:         printf("%s was found at position %d.", buf2, loc-buf1);
25: }
```

Salida

```
Enter the string to be searched: How now brown Cow?
Enter the target string:
Cow was found at position 14.
```

Análisis Aquí se le presenta una manera alterna de buscar una cadena. Esta vez puede buscar una cadena dentro de otra cadena. Las líneas 12 a 15 le piden dos cadenas. La línea 19 usa `strstr()` para buscar la segunda cadena, `buf2` dentro de la primera, `buf1`. Se regresa un apuntador a la primera aparición, o `NULL` en caso de que la cadena no se encuentre. Las líneas 21 a 24 evalúan el valor regresado, `loc`, e imprimen un mensaje adecuado.

DEBE	NO DEBE
DEBE Recordar que para muchas de las funciones de cadenas hay funciones equivalentes que le permiten especificar la cantidad de caracteres que se han de manejar. Las funciones que le permiten la especificación de la cantidad de caracteres son denominadas, por lo general, `strnxxx()`, donde `xxx` es específico para la función.	**NO DEBE** Olvidar que el C toma en cuenta las mayúsculas y las minúsculas. `"A"` y `"a"` son diferentes.

Conversión de cadenas

Muchas de las bibliotecas del C contienen dos funciones que cambian las mayúsculas y las minúsculas de los caracteres dentro de una cadena. Estas funciones no están dentro del estándar ANSI y, por lo tanto, pueden ser diferentes o ni siquiera existir en su compilador. Sus prototipos, en STRING.H, deben ser similares a

```
char *strlwr(char *str);
char *strupr(char *str);
```

La función `strlwr()` convierte todos los caracteres de letra de `str` de mayúscula a minúscula, y `strupr()` hace lo contrario, convirtiendo todos los caracteres de `str` a mayúsculas. Los caracteres que no corresponden a letras no son afectados. Ambas funciones regresan `str`. Observe que ninguna función crea, de hecho, una nueva cadena, sino que modifica en su lugar a la existente. El programa del listado 17.13 muestra estas funciones.

Análisis

Listado 17.13. Conversión de mayúsculas y minúsculas en una cadena con `strlwr()` y `strupr()`.

```
1: /* Las funciones para conversión de caracteres strlwr() y strupr(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
```

```
8:      char buf[80];
9:
10:     while (1)
11:     {
12:         puts("Enter a line of text, a blank to exit.");
13:         gets(buf);
14:
15:         if ( strlen(buf) == 0 )
16:             break;
17:
18:         puts(strlwr(buf));
19:         puts(strupr(buf));
20:     }
21: }
```


Salida

```
Enter a line of text, a blank to exit.
Bradley L. Jones
bradley l. jones
BRADLEY L. JONES
Enter a line of text, a blank to exit.
```

Análisis

Este listado le pide, en la línea 12, una cadena. Luego revisa si la cadena no está en blanco (línea 15). La línea 18 imprime la cadena después de convertirla a minúsculas. La línea 19 imprime la cadena en puras mayúsculas.

Estas funciones son parte de las bibliotecas del C de Zortech y Borland. En el C de Microsoft las funciones están precedidas por un carácter de subrayado (`_strlwr()` y `_strupr()`). Necesita revisar el manual de consulta de la biblioteca de su compilador antes de usar estas funciones. Si necesita que su programa sea portable debe evitar el uso de funciones no ANSI como éstas.

Funciones diversas para cadenas

Esta sección trata unas cuantas funciones de cadena que no corresponden a ninguna otra categoría. Todas requieren el archivo de encabezado STRING.H.

La función *strrev()*

La función `strrev()` invierte el orden de todos los caracteres en una cadena. Su prototipo es

```
char *strrev(char *str);
```

El orden de todos los caracteres de `str` se invierte, quedando al final el carácter nulo terminal. La función regresa `str`. Después de definir `strset()` y `strnset()` en la siguiente sección, se muestra a `strrev()` en el listado 17.14.

Las funciones *strset()* y *strnset()*

De manera similar a las funciones anteriores, `strrev()`, `strset()` y `strnset()` no son parte de la biblioteca estándar del C ANSI. Estas funciones cambian todos los caracteres (`strset()`) o una cantidad específica de caracteres (`strnset()`) en una cadena a un carácter especificado. Los prototipos son

```
char *strset(char *str, int ch);
char *strnset(char *str, int ch, size_t n);
```

La función `strset()` cambia todos los caracteres en `str` a `ch`, a excepción del carácter nulo terminal. La función `strnset()` cambia los primeros n caracteres de `str` a `ch`. Si `n >= strlen(str)`, `strnset()` cambia todos los caracteres en `str`. El listado 17.14 muestra ambas funciones.

Captura

Listado 17.14. Una demostración de `strrev()`, `strnset()` y `strset()`.

```
1: /* strrev(), strset() y strnset(). */
2: #include <stdio.h>
3: #include <string.h>
4:
5: char str[] = "This is the test string.";
6:
7: main()
8: {
9:     printf("\nThe original string: %s", str);
10:     printf("\nCalling strrev(): %s", strrev(str));
11:     printf("\nCalling strrev() again: %s", strrev(str));
12:     printf("\nCalling strnset(): %s", strnset(str, '!', 5));
13:     printf("\nCalling strset(): %s", strset(str, '!'));
14: }
```

Salida

```
The original string: This is the test string.
Calling strrev(): .gnirts tset eht si sihT
Calling strrev() again: This is the test string.
Calling strnset(): !!!!!is the test string.
Calling strset(): !!!!!!!!!!!!!!!!!!!!!!!!
```

Análisis

Este programa muestra las tres diferentes funciones de cadena. La demostración se hace imprimiendo el valor de una cadena, `str`. La línea 9 imprime la cadena normal. La línea 10 imprime la cadena después de que ha sido invertida con `strrev()`. La línea 11 la vuelve a invertir a su estado original. La línea 12 usa la función `strnset()` para cambiar los primeros cinco caracteres de `str` a signos de admiración. Para terminar el programa, la línea 13 cambia la cadena completa a signos de admiración.

Aunque estas funciones no son del estándar ANSI, son parte de los compiladores C de Zortech y Borland. Microsoft soporta estas funciones precediéndolas con un carácter de

subrayado (`_strrev()`, `_strnset()` y `_strnset()`). Debe revisar el manual de consulta de la biblioteca para determinar si su compilador acepta estas funciones.

Conversión de cadenas a números

Hay veces que se necesita convertir la representación en cadena de un número a la variable numérica actual. Por ejemplo, la cadena `"123"` puede ser convertida a una variable tipo `int` con el valor `123`. Hay tres funciones que convierten una cadena a un número, explicadas en los siguientes párrafos, y sus prototipos están en STDLIB.H.

La función *atoi()*

La función de biblioteca `atoi()` convierte una cadena a un entero. El prototipo es

```
int atoi(char *ptr);
```

La función `atoi()` convierte la cadena apuntada por `ptr` a un entero. Además de los dígitos, la cadena puede contener espacios en blanco al inicio y un signo de más + o de menos —. La conversión se inicia al principio de la cadena y continúa hasta que encuentra un carácter no convertible (por ejemplo, una letra o signo de puntuación). El entero resultante es regresado al programa que llama. Si no encuentra caracteres convertibles, `atoi()` regresa cero. La tabla 17.2 presenta algunos ejemplos.

Tabla 17.2. Conversión de cadenas a números con `atoi()`.

Cadena	Valor regresado por `atoi()`
`"157"`	157
`"-1.6"`	-1
`"+50x"`	50
`"doce"`	0
`"x506"`	0

El primer ejemplo es muy claro. En el segundo ejemplo tal vez le confunda el porqué no se traduce el `".6"`. Recuerde que es una conversión de cadena a entero. El tercer ejemplo también es claro, ya que la función "comprende" el signo de más y lo considera como parte del número. El cuarto ejemplo usa `"doce"`. La función `atoi()` no puede traducir palabras,

ya que todo lo que "ve" son caracteres. Debido a que la cadena no comienza con un número, `atoi()` regresa un cero. Esto también es cierto para el último ejemplo.

La función *atol()*

La función de biblioteca `atol()` funciona exactamente igual que `atoi()`, a excepción de que regresa un número tipo `long`. El prototipo de función es

```
long atol(char *ptr);
```

Los valores regresados por `atol()` podrían ser los mismos que los mostrados para `atoi()` en la tabla 17.2, a excepción de que cada valor de retorno sería de tipo `long` en vez de tipo `int`.

La función *atof()*

La función `atof()` convierte una cadena a tipo `double`. El prototipo es

```
double atof(char *str);
```

El argumento `str` apunta a la cadena a ser convertida. Esta cadena puede contener espacios en blanco iniciales y un carácter de signo de más (+) o de menos (-). El número puede contener los dígitos del 0 al 9, el punto decimal y un indicador de exponente, E o e. Si no hay caracteres convertibles, `atof()` regresa cero. La tabla 17.3 lista algunos ejemplos de `atof()`. El programa del listado 17.15 le permite teclear sus propias cadenas para convertirlas.

Tabla 17.3. Conversión de cadenas a números con `atof()`.

Cadena	Valor regresado por `atof()`
`"12"`	12.000000
`"-0.123"`	-0.123000
`"123E+3"`	123000.000000
`"123.1e-5"`	0.001231



Listado 17.15. Uso de `atof()` para convertir cadenas a variables numéricas tipo `double`.

```
1: /* Demostración de atof(). */
2:
3:
4: #include <stdio.h>
```

```
5: #include <stdlib.h>
6:
7: main()
8: {
9:     char buf[80];
10:    double d;
11:
12:    while (1)
13:    {
14:        printf("\nEnter the string to convert /
               (blank to exit):     ");
15:        gets(buf);
16:
17:        if ( strlen(buf) == 0 )
18:            break;
19:
20:        d = atof( buf );
21:
22:        printf("The converted value is %f.", d);
23:    }
24:}
```


Salida

```
Enter the string to convert (blank to exit): 1009.12
The converted value is 1009.120000.
Enter the string to convert (blank to exit): abc
The converted value is 0.000000.
Enter the string to convert (blank to exit): 3
The converted value is 3.000000.
Enter the string to convert (blank to exit):
```

Análisis

Este listado usa un ciclo `while`, en las líneas 12 a 23, para permitir que se mantenga ejecutando el programa hasta que se dé una línea en blanco. Las líneas 14 y 15 le piden el valor. La línea 17 revisa para ver si de dio una línea en blanco. Si fue así, el programa sale del ciclo `while` y termina. La línea 20 llama a `atof()`, convirtiendo el valor dado, `buf`, a un tipo double, `d`. La línea 22 imprime el resultado final.

Funciones de prueba de caracteres

El archivo de encabezado CTYPE.H contiene los prototipos para varias funciones que prueban caracteres, regresando `TRUE` o `FALSE` en caso de que el carácter satisfaga determinada condición. Por ejemplo, ¿será una letra o un número? Las funciones `isxxxx()` son, de hecho, macros definidas en CTYPE.H. Aprenderá sobre las macros en el Día 21, "Cómo aprovechar las directivas del preprocesador y más", y en ese momento tal vez querrá ver las definiciones en CTYPE.H para ver la manera en que funcionan. Por ahora, solamente necesita saber la manera en que se usan.

Todas las macros `isxxxx()` tienen el mismo prototipo.

```
int isxxxx(int ch);
```

donde `ch` es el carácter a revisión. El valor de retorno es `TRUE` (diferente de cero) si se satisface la condición, o `FALSE` (cero) en caso contrario. La tabla 17.4 lista el juego completo de macros `isxxxx()`.

Tabla 17.4. Las macros `isxxxx()`.

Macro	Acción
`isalnum()`	Regresa `TRUE` si `ch` es una letra o un dígito.
`isalpha()`	Regresa `TRUE` si `ch` es una letra.
`isascii()`	Regresa `TRUE` si `ch` es un carácter ASCII estándar (entre 0 y 127).
`iscntrl()`	Regresa `TRUE` si `ch` es un carácter de control.
`isdigit()`	Regresa `TRUE` si `ch` es un dígito.
`isgraph()`	Regresa `TRUE` si `ch` es un carácter imprimible (diferente de espacio).
`islower()`	Regresa `TRUE` si `ch` es una letra minúscula.
`isprint()`	Regresa `TRUE` si `ch` es un carácter imprimible (incluyendo el espacio).
`ispunct()`	Regresa `TRUE` si `ch` es un carácter de puntuación.
`isspace()`	Regresa `TRUE` si `ch` es un carácter de espacio en blanco (espacio, tabulador, tabulador vertical, avance de línea, avance de forma o retorno).
`isupper()`	Regresa `TRUE` si `ch` es una letra mayúscula.
`isxdigit()`	Regresa `TRUE` si `ch` es un dígito hexadecimal (0 - 9, a - f, A - F).

Se pueden hacer muchas cosas interesantes con las macros de prueba de carácter. Un ejemplo es la función `get_int()` del listado 17.16. Esta función recibe un entero de `stdin` y lo regresa como una variable tipo `int`. La función pasa por alto los espacios en blanco iniciales, y regresa 0 en caso de que el primer carácter no blanco no sea un carácter numérico.

Listado 17.16. Uso de las macros `isxxxx()` para implementar una función que reciba un entero.

```
1: /* Uso de las macros para revisión de caracteres para crear */
2: /* una función para recibir enteros. */
3:
4: #include <stdio.h>
5: #include <ctype.h>
6:
7: int get_int(void);
8:
9: main()
10: {
11:     int x;
12:     x =  get_int();
13:
14:     printf("You entered %d.", x);
15: }
16:
17: int get_int(void)
18: {
19:     int ch, i, sign = 1;
20:
21:     /* Se salta cualquier espacio en blanco inicial. */
22:
23:     while ( isspace(ch = getchar()) )
24:         ;
25:
26:     /* Si el primer carácter es no numérico, */
27:     /* lo desobtiene y regresa 0. */
28:
29:     if (ch != '-' && ch != '+' && !isdigit(ch) && ch != EOF)
30:     {
31:         ungetc(ch, stdin);
32:         return 0;
33:     }
34:
35:     /* Si el primer carácter es un signo de menos, */
36:     /* ajusta el signo adecuadamente. */
37:
38:     if (ch == '-')
39:         sign = -1;
40:
41:     /* Si el primer carácter fue un signo de más o de menos, */
42:     /* obtiene el siguiente carácter. */
43:
44:     if (ch == '+' || ch == '-')
45:         ch = getchar();
46:
47:     /* Lee caracteres hasta que se recibe uno que no es dígito. */
```

Listado 17.16. continuación

```
48:     /* Asigna valores a i, multiplicados por la potencia adecuada de 10. */
49:
50:     for (i = 0; isdigit(ch); ch = getchar() )
51:         i = 10 * i + (ch - '0');
52:
53:     /* Hace que el resultado sea negativo si sign es negativo. */
54:
55:     i *= sign;
56:
57:     /* Si no se ha encontrado EOF, se debe haber leído un carácter */
58:     /* que no es dígito y, por lo tanto, hay que desobtenerlo. */
59:
60:     if (ch != EOF)
61:         ungetc(ch, stdin);
62:
63:     /* Regresa el valor recibido. */
64:
65:     return i;
66: }
```

Salida

```
>list1716
-100
You entered -100.
>list1716
abc3.145
You entered 0.
>list1716
9 9 9
You entered 9.
>list1716
  2.5
You entered 2.
```

Análisis Este programa usa, en las líneas 31 y 61, la función de biblioteca `ungetc()`, que se vio en el Día 14, "Trabajando con la pantalla, la impresora y el teclado". Recuerde que esta función "desobtiene" o regresa un carácter al flujo especificado. Este carácter "regresado" es el primero que se recibe la siguiente vez que el programa lee un carácter de ese flujo. Esto es necesario para el caso en que la función `get_int()` lea un carácter no numérico de `stdin`, debido a que se quiere que regrese ese carácter por si el programa lo necesita posteriormente.

En este programa la función `main()` es simple. Una variable entera, `x`, es declarada (línea 11), se le asigna el valor de la función `get_int()` (línea 12) y se imprime en la pantalla (línea 14). La función `get_int()` forma el resto del programa.

La función `get_int()` no es tan simple como `main()`. Para quitar los espacios en blanco iniciales que puedan ser dados, la línea 23 hace ciclo con un comando `while`. La macro `isspace()` revisa un carácter, `ch`, que ha sido obtenido con la función `getchar()`. Si `ch` es un espacio se obtiene otro carácter, y así sucesivamente hasta que se reciba un carácter que no sea espacio en blanco. La línea 29 revisa si el carácter es alguno que pueda ser usado. La línea 29 podría leerse "si el carácter obtenido no es un signo negativo, un signo de más, un dígito o el fin del archivo". Si esto es cierto se usa `ungetc()`, en la línea 31, para regresar el carácter y la función regresa a `main()`. Si el carácter es utilizable continúa la ejecución.

Las líneas 38 a 45 se encargan del signo del número. La línea 38 revisa para ver si el carácter dado fue un signo negativo. En caso de haberlo sido, la variable `sign` es puesta a -1. `sign` es utilizada para hacer positivo o negativo al número final (línea 55). Debido a que por omisión esperamos números positivos, una vez que se ha ocupado del signo negativo podemos continuar. Si se ha dado un signo el programa necesita obtener otro carácter. Las líneas 44 y 45 se encargan de esto.

La parte medular de la función es el ciclo `for` en las líneas 50 y 51, que continúa obteniendo caracteres en tanto que los que obtenga sean dígitos. La línea 51 puede parecer un poco confusa a primera vista. Esta línea toma el carácter individual dado y lo convierte a un número. Al restar del número al carácter `'0'` se cambia un carácter de número a un número real. (Recuerde los valores ASCII.) Una vez que se obtiene el valor numérico correcto los números son multiplicados por la potencia de 10 adecuada. El ciclo `for` continúa hasta que se recibe un carácter no numérico. En ese momento la línea 55 aplica el signo al número, dándolo por terminado.

Antes de regresar el programa necesita hacer algo de limpieza. Si el último número no fue el fin de archivo necesita regresarlo, por si se necesita en otro lugar. La línea 61 hace esto antes de que la línea 65 haga el regreso.

DEBE	NO DEBE
NO DEBE Usar funciones no ANSI si tiene planes de transportar su aplicación a otras plataformas.	
DEBE Aprovechar las funciones de cadena que tiene disponibles.	
	NO DEBE Confundir los caracteres con los números. Es fácil olvidar que `'1'` no es lo mismo que `1`.

Resumen

Este capítulo mostró diversas formas en que se puede manejar a las cadenas. Usando las funciones de biblioteca estándar del C (y posiblemente también las funciones específicas de su compilador) puede copiar, concatenar, comparar y buscar cadenas. Todas éstas son tareas necesarias en la mayoría de proyectos de programación. La biblioteca estándar también contiene funciones para convertir entre mayúsculas y minúsculas dentro de cadenas, y para convertir cadenas a números. Por último, el C proporciona una variedad de funciones para la revisión de carácter o, siendo más precisos, macros que ejecutan una variedad de pruebas sobre caracteres individuales. Usando estas macros para revisar caracteres puede crear sus propias funciones de entrada.

Preguntas y respuestas

1. ¿Cómo sé si la función es compatible con ANSI?

   La mayoría de los compiladores tienen un manual, o sección, de consulta de la biblioteca de funciones. Este manual, o sección del manual, lista todas las funciones de la biblioteca del compilador y la manera de usarlas. Por lo general, el manual incluye información sobre la compatibilidad de la función. Algunas veces las descripciones indican no solamente si la función es compatible con ANSI, sino también si es compatible con el DOS, UNIX, Windows, C++ u OS/2. (La mayoría de los compiladores solamente dicen lo que es relevante a ese compilador.)

2. ¿Han sido presentadas en este capítulo todas las funciones de cadenas disponibles?

   No. Sin embargo, las funciones de cadena presentadas en este capítulo deben cubrir virtualmente todas sus necesidades. Revise el manual de consulta de la biblioteca del compilador para ver qué otras funciones están disponibles.

3. ¿Ignora `strcat()` los espacios al final de la cadena cuando hace una concatenación?

   No. `strcat()` considera a los espacios como cualquier otro carácter.

4. ¿Puedo convertir números a cadenas?

   Sí. Se puede escribir una función similar a la que se encuentra en el listado 17.16, o se puede revisar el manual de consulta de la biblioteca para ver las funciones disponibles. Algunas funciones disponibles incluyen a `itoa()`, `ltoa()` y `ultoa()`.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Qué es la longitud de una cadena y cómo puede ser determinada?
2. ¿De qué cosa debe asegurarse antes de copiar una cadena?
3. ¿Qué significa el término *concatenación?*
4. Cuando se comparan cadenas, ¿qué quiere decir "una cadena es mayor que otra"?
5. ¿Cuál es la diferencia entre `strcmp()` y `strncmp()`?
6. ¿Cuál es la diferencia entre `strcmp()` y `strcmpl()`?
7. ¿Qué valores revisa `isascii()`?
8. Usando la tabla 17.4, ¿cuáles macros regresarían TRUE para `var`?

```
int var = 1;
```

9. Usando la tabla 17.4, ¿cuáles macros regresarían TRUE para `x`?

```
char x = 65;
```

10. ¿Para qué se usan las funciones de revisión de caracteres?

Ejercicios

1. ¿Qué valores regresan las funciones de revisión?
2. ¿Cuál sería el valor de retorno de la función `atoi()` si se le pasaran los siguientes valores?

    a. "65"

    b. "81.23"

    c. "-34.2"

    d. "diez"

    e. "+12mil"

    f. "100negativo"

3. ¿Cuál sería el valor de retorno de la función `atof()` si se le pasaran los siguientes valores?

   a. "65"

   b. "81.23"

   c. "-34.2"

   d. "diez"

   e. "+12mil"

   f. "1e+3"

4. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
char *cadena1, cadena2;

cadena 1 = "Hello World";

strcpy( cadena2, cadena1);

printf( "%s, %s", cadena1, cadena2 );
```

   Debido a que hay muchas soluciones posibles, no se proporcionan respuestas para los siguientes ejercicios.

5. Escriba un programa que le pida al usuario su nombre y apellidos en forma individual. Luego, guarde el nombre en una nueva cadena como inicial del nombre, punto, espacio, apellido paterno, espacio, inicial del segundo apellido, punto. Por ejemplo, si se diera "Juan", "Pérez", "López", se guardaría "J. Pérez L.". Despliegue el nuevo nombre en la pantalla.

6. Escriba un programa que compruebe sus respuestas a las preguntas 8 y 9 del cuestionario.

7. La función `strstr()` encuentra la primera aparición de una cadena dentro de otra y toma en cuenta a las mayúsculas y minúsculas. Escriba una función que ejecute la misma tarea sin tomar en cuenta mayúsculas y minúsculas.

8. Escriba una función que determine la cantidad de veces que una cadena se encuentra dentro de otra.

9. Escriba un programa que busque en un archivo de texto las apariciones de una cadena especificada por el usuario, y luego informe los números de línea donde encuentre la cadena. Por ejemplo, si se busca en un archivo de código fuente del C la cadena `"printf()"`, el programa debe listar todos los renglones donde la función `printf()` sea llamada por el programa.

10. El listado 17.16 muestra una función que recibe un entero de `stdin`. Escriba una función, `get_float()`, que reciba un valor de punto flotante de `stdin`.

DIA 18

Cómo obtener más de las funciones

Como usted ya debe saber, las funciones son básicas en la programación en C. Hoy aprenderá

- ❑ Apuntadores como argumentos a funciones.
- ❑ Apuntadores tipo void como argumentos.
- ❑ Cómo usar funciones con un número variable de argumentos.
- ❑ Cómo regresar un apuntador desde una función.

Algunos de estos temas han sido tratados anteriormente en el libro, pero este capítulo le da información más minuciosa.

Paso de apuntadores a funciones

El método predeterminado para pasar un argumento a una función es *por valor.* El paso por valor significa que a la función se le pasa una copia del valor del argumento. Este método tiene tres pasos:

1. Se evalúa la expresión del argumento.
2. El resultado es copiado a *la pila,* que es un área en memoria para almacenamiento temporal.
3. La función recupera el valor del argumento desde la pila.

El procedimiento de pasar un argumento por valor es ilustrado en la figura 18.1. En este caso el argumento es una simple variable tipo `int`, pero el principio es el mismo para otro tipo de variables y para expresiones más complejas.

Cuando una variable es pasada por valor a una función, la función tiene acceso al valor de la variable pero no a la copia original de la variable. Por consecuencia, el código de la función no puede modificar a la variable original. Esta es la razón principal de que el método predeterminado para pasar argumentos sea *por valor.* Los datos externos a una función están protegidos de modificaciones inadvertidas.

El paso por valor es posible con los tipos de datos básicos (`char, int, long, float` y `double`) y con las estructuras. Sin embargo, hay otra manera de pasar argumentos a una función, pasando un apuntador a la variable del argumento en vez del valor de la variable misma. A este método de pasar un argumento se le llama *paso por referencia.*

Como aprendió en el Día 9, "Apuntadores", el paso por referencia es la única forma de pasar un arreglo a una función, ya que no es posible pasar los arreglos por valor. Sin embargo, con otros tipos de datos se puede usar cualquiera de ambos métodos. Si el programa usa estructuras grandes, el pasarlas por valor puede hacer que el programa se quede sin espacio de pila. Aparte de esta consideración, el pasar un argumento por referencia, en vez de por valor, ofrece ventajas y desventajas:

- La ventaja de pasar por referencia es que la función puede modificar el valor de la variable del argumento.
- La desventaja de pasar por referencia es que la función puede modificar el valor de la variable del argumento.

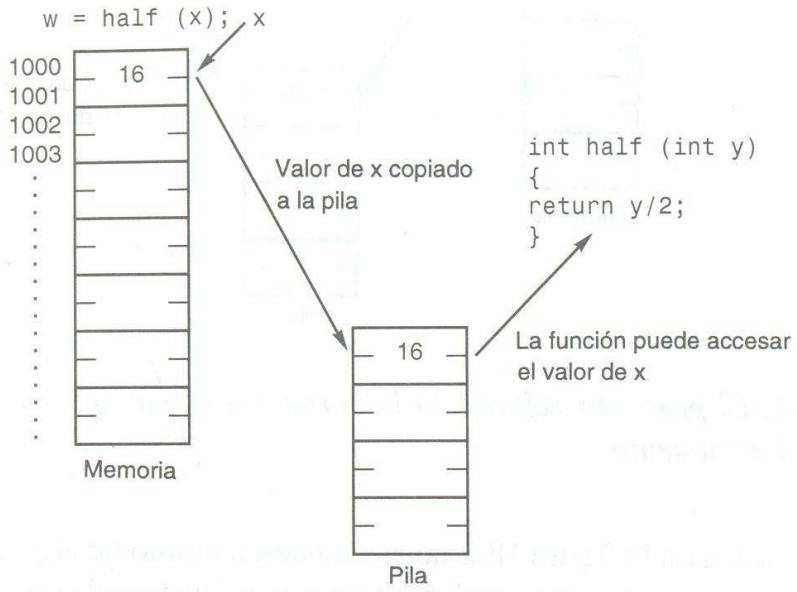


Figura 18.1. *Paso de un argumento por valor. La función no puede modificar el original de la variable de argumento.*

Tal vez diga usted "¡Caramba!, ¿una ventaja que también es una desventaja?". Sí. Todo depende de la situación específica. Si el programa requiere una función para modificar una variable del argumento, el pasarla por referencia es una ventaja. Si no se necesita es una desventaja, debido a la posibilidad de modificaciones inadvertidas.

Tal vez se pregunte por qué no se usa el valor de retorno de la función para modificar la variable del argumento. Por supuesto que lo puede hacer, como se muestra en el siguiente ejemplo:

```
x = half(x);
int half(int y)
{
   return y/2;
}
```

Recuerde, sin embargo, que una función puede regresar solamente un solo valor. Al pasar uno o más argumentos por referencia se permite que la función "regrese" más de un valor al programa que la llama. La figura 18.2 ilustra el paso por referencia para un solo argumento.

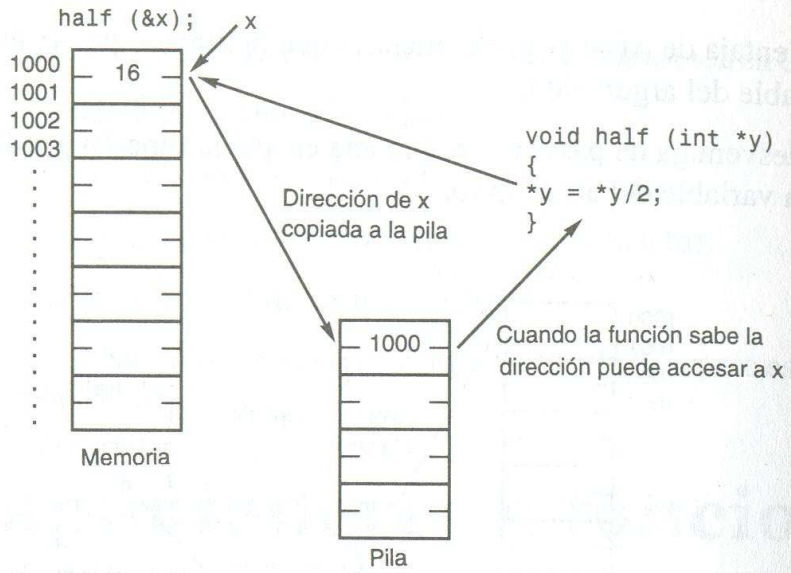


Figura 18.2. *El paso por referencia le permite a la función modificar el original de la variable del argumento.*

La función usada en la figura 18.2 no es un buen ejemplo de algo que podría usar *pasando por referencia* en un programa real, pero sirve para ilustrar el concepto. Cuando se pasa por referencia, debe asegurarse que la definición de la función y el prototipo reflejen el hecho de que el argumento pasado a la función es un apuntador. Dentro del cuerpo de la función también debe usar el operador de *indirección* para accesar la o las variables pasadas por referencia.

El programa del listado 18.1 muestra el paso por referencia y el paso por valor predeterminado. Su salida muestra claramente que una variable pasada por valor no puede ser cambiada por la función, y en cambio una variable pasada por referencia sí puede ser cambiada. Por supuesto, una función no necesita modificar una variable pasada por referencia, pero si la función no lo necesita, no hay razón para pasarla por referencia.

Captura

Listado 18.1. Paso por valor y paso por referencia.

```
1: /* Paso de argumentos por valor y por referencia.
2:
3: */ #include <stdio.h>
4:
5: void by_value(int a, int b, int c);
6: void by_ref(int *a, int *b, int *c);
7:
8: main()
9: {
10:     int x = 2, y = 4, z = 6;
11:
```

```
12:     printf("\nBefore calling by_value(), x = %d, y = %d, z =%d.",
13:         x, y, z);
14:
15:     by_value(x, y, z);
16:
17:     printf("\nAfter calling by_value(), x = %d, y = %d, z = %d.",
18:           x, y, z);
19:
20:     by_ref(&x, &y, &z);
21:
22:     printf("\nAfter calling by_ref(), x = %d, y = %d, z = %d.",
23:           x, y, z);
24: }
25:
26: void by_value(int a, int b, int c)
27: {
28:     a = 0;
29:     b = 0;
30:     c = 0;
31: }
32:
33: void by_ref(int *a, int *b, int *c)
34: {
35:     *a = 0;
36:     *b = 0;
37:     *c = 0;
38: }
```


Salida

```
Before calling by_value(), x = 2, y = 4, z = 6.
After calling by_value(), x = 2, y = 4, z = 6.
After calling by_ref(), x = 0, y = 0, z = 0.
```

Análisis

Este programa muestra la diferencia entre el pasar variables por valor y pasarlas por referencia. Las líneas 5 y 6 contienen prototipos para las dos funciones llamadas por el programa. Observe que la línea 5 describe tres argumentos de tipo `int` para la función `by_value()`, pero `by_ref()` difiere en la línea 6, debido a que requiere como argumentos tres apuntadores a variables tipo `int`. Los encabezados de función para estas dos funciones, en las líneas 26 y 33, siguen el mismo formato que los prototipos. Los cuerpos de las dos funciones son parecidos pero no son el mismo. Ambas funciones asignan 0 a las tres variables que se les pasan. En la función `by_value()` se asigna 0 directamente a las variables. En la función `by_ref()` se usan apuntadores, por lo que las variables deben ser desreferenciadas.

Cada función es llamada una vez por `main()`. Primero, se les asignan valores diferentes de cero, en la línea 10, a las tres variables que han de pasarse. La línea 12 imprime estos valores en la pantalla. La línea 15 llama a la primera de las dos funciones, `by_value()`. La línea 17 imprime nuevamente las tres variables. Observe que no han cambiado. La función `by_value()` recibe las variables por valor y, por lo tanto, no puede cambiar su contenido

original. La línea 20 llama a `by_ref()` y la línea 22 vuelve a imprimir los valores. En esta ocasión todos los valores han cambiado a cero. Al pasar las variables por referencia se le permite a `by_ref()` el acceso al contenido actual de las variables.

Se puede escribir una función que reciba algunos argumentos por referencia y otros por valor. Simplemente recuerde manejarlos adecuadamente dentro de la función, usando el operador de indirección (*) para desreferenciar los argumentos pasados por referencia.

DEBE	NO DEBE
NO DEBE Pasar gran cantidad de datos por valor si no es necesario. Se puede quedar sin espacio de pila.	
DEBE Pasar variables por valor si no quiere alterar el valor original.	
NO DEBE Olvidar que una variable pasada por referencia debe ser un apuntador. También use el operador de indirección para desreferenciar la variable en la función.	

Apuntadores tipo *void*

Ha visto que se usa la palabra clave `void` para especificar que una función ni toma argumentos ni regresa un valor. La palabra clave `void` también puede ser usada para crear un apuntador "genérico", un apuntador que puede apuntar a cualquier tipo de objeto de datos. Por ejemplo, el enunciado

```
void *x;
```

declara a `x` como un apuntador genérico. `x` apunta a algo, pero simplemente no se ha especificado *a qué*.

El uso más común para los apuntadores tipo `void` es en la declaración de parámetros de función. Tal vez quiera crear una función que pueda manejar diferentes tipos de argumentos: en una ocasión le puede pasar un tipo `int`, en otra ocasión un tipo `float` y así sucesivamente. Al declarar que la función toma un apuntador tipo `void` como argumento, no se le restringe a que acepte únicamente un solo tipo de dato. Si declara una función para que tome como argumento un apuntador `void`, le puede pasar a la función un apuntador a cualquier cosa.

Aquí tiene un ejemplo simple: se quiere una función que acepte como argumento a una variable numérica y la divida entre 2, regresando la respuesta en la variable del argumento. Por lo tanto, si la variable `x` tiene el valor de 4, después de la llamada a `half(x)` la variable

x es igual a 2. Debido a que se quiere modificar el argumento, se le pasa por referencia. Debido a que se quiere usar la función con cualquiera de los tipos de dato numéricos del C se declara la función para que reciba un apuntador `void`:

```
void half(void *x);
```

Ahora puede llamar a la función pasándole como argumento cualquier apuntador. Sin embargo, hay una cosa adicional que es necesaria. Aunque se puede pasar un apuntador `void` sin saber a qué tipo de dato apunta, no se podrá desreferenciar el apuntador. Antes de que el código de la función pueda hacer cualquier cosa con el apuntador, se debe saber el tipo de dato. Esto se logra con un *especificador de tipo,* que es simplemente una manera de decirle al programa "trata a este apuntador `void` como un apuntador a tipo". Si `x` es un apuntador `void`, se puede especificar el tipo de la manera siguiente:

```
(tipo *)x
```

donde `tipo` es el tipo de dato adecuado. Para decirle al programa que `x` es un apuntador a tipo `int`, escriba:

```
(int *)x
```

Para desreferenciar el apuntador, esto es, para accesar el `int` al que apunta `x`, escriba:

```
*(int *)x
```

Regresando al tema original (el pasar un apuntador `void` a una función) puede ver que para usar el apuntador la función debe saber el tipo de dato al cual apunta. En el caso de la función que está escribiendo para obtener la mitad del argumento, hay cuatro posibilidades para tipo: `int`, `long`, `float` y `double`. Además del apuntador `void` a la variable que se va a dividir entre dos, se le debe decir a la función el tipo de variable a la que apunta el apuntador `void`. Se puede modificar la definición de la función de la manera siguiente:

```
void half(void *x, char type);
```

En base al argumento `type`, la función ajusta el apuntador a `x`, tipo `void`, al tipo apropiado. Luego el apuntador puede ser desreferenciado y puede ser usado el valor de la variable apuntada. La versión final de la función `half()` se muestra en el listado 18.2.

Captura

Listado 18.2. Uso de un apuntador a `void` para pasar tipos de dato diferentes a una función.

```
1: /* Uso de apuntadores a tipo void. */
2:
3: #include <stdio.h>
4:
5: void half(void *x, char type);
6:
7: main()
```

Listado 18.2. continuación

```
8: {
9:     /* Inicializa una variable de cada tipo. */
10:
11:     int i = 20;
12:     long l = 100000;
13:     float f = 12.456;
14:     double d = 123.044444;
15:
16:     /* Despliega los valores iniciales. */
17:
18:     printf("\n%d", i);
19:     printf("\n%ld", l);
20:     printf("\n%f", f);
21:     printf("\n%lf\n\n", d);
22:
23:     /* Llama a half() para cada variable. */
24:
25:     half(&i, 'i');
26:     half(&l, 'l');
27:     half(&d, 'd');
28:     half(&f, 'f');
29:
30:     /* Despliega los nuevos valores. */
31:
32:     printf("\n%d", i);
33:     printf("\n%ld", l);
34:     printf("\n%f", f);
35:     printf("\n%lf", d);
36: }
37:
38: void half(void *x, char type)
39: {
40:     /* Dependiendo del valor de type asigna el */
41:     /* apuntador a x adecuado y divide entre dos. */
42:
43:     switch (type)
44:     {
45:         case 'i':
46:             {
47:             *((int *)x) /= 2;
48:             break;
49:             }
50:         case 'l':
51:             {
52:             *((long *)x) /= 2;
53:             break;
54:             }
55:         case 'f':
```

```
56:             {
57:             *((float *)x) /= 2;
58:             break;
59:             }
60:         case 'd':
61:             {
62:             *((double *)x) /= 2;
63:             break;
64:             }
65:     }
66: }
```

Salida

```
20
100000
12.456000
123.044444

10
50000
6.228000
61.522222
```

Análisis

Tal como está instrumentada, la función `half()`, en las líneas 38 a 66, no incluye revisión de errores (como, por ejemplo, el que le sea pasado un argumento de tipo inválido). Sin embargo, en un programa real no se utilizaría una función para ejecutar una tarea tan simple como es el dividir un valor entre dos. Este es solamente un ejemplo ilustrativo.

Tal vez piense que la necesidad de pasar el tipo de la variable apuntada hace menos flexible a la función. La función podría ser más general si no necesitara saber el tipo de objeto de dato apuntado, pero ésta no es la forma en que trabaja el C. Siempre se debe especificar el tipo de un apuntador `void` a un tipo específico antes de que se le desreferencie. Usando el método anterior se puede escribir una sola función. Si no se utiliza un apuntador `void`, se necesitaría escribir cuatro funciones separadas, una para cada tipo de dato.

Cuando se necesita una función que pueda manejar diferentes tipos de datos, por lo general se escribe una *macro* en vez de la función. El ejemplo que se acaba de presentar, en el cual la tarea ejecutada por la función es bastante simple, sería un buen candidato para una macro. (En el Día 21, "Cómo aprovechar las directivas del preprocesador y más", se trata a las macros.)

Tome en cuenta que los apuntadores tipo `void` no pueden ser incrementados ni decrementados.

DEBE	NO DEBE
DEBE Especificar el tipo de un apuntador `void` cuando use el valor al que apunta.	
NO DEBE Tratar de incrementar ni decrementar un apuntador `void`.	

Funciones con número variable de argumentos

Ya ha usado varias funciones de biblioteca, como `printf()` y `scanf()`, que toman un número variable de argumentos. Usted puede escribir sus propias funciones que tomen una lista variable de argumentos. Los programas que tengan funciones con listas variables de argumentos deben de incluir el archivo de encabezado STDARG.H.

Cuando se declara una función que toma una lista variable de argumentos primero se listan los parámetros fijos, es decir, aquellos que siempre están presentes (debe haber por lo menos un parámetro fijo). Luego se incluyen *puntos suspensivos* (...) al final de las lista de parámetros, para indicar que se pasan cero o más argumentos adicionales a la función. En esta explicación no olvide la distinción entre un parámetro y un argumento, como se explicó en el Día 5, "Funciones: lo básico".

¿Cómo sabe la función qué tantos argumentos se le han pasado en una llamada específica? Uno se lo dice. Uno de los parámetros fijos le dice a la función la cantidad total de argumentos. Por ejemplo, cuando se usa la función `printf()` la cantidad de especificadores de conversión, en la cadena de formato, le dice a la función qué tantos argumentos adicionales debe esperar. En forma más directa, un argumento fijo puede ser la cantidad de argumentos adicionales. El ejemplo que se presenta a continuación usa este método, pero primero se necesita dar una mirada a las herramientas que proporciona el C para manejar una lista variable de argumentos.

La función también debe saber el tipo de cada argumento en la lista de variables. En el caso de `printf()`, los especificadores de conversión indican el tipo de cada argumento. En otros casos, como el del siguiente ejemplo, todos los argumentos de la lista variable son del mismo tipo, por lo que no hay problema. Para crear una función que acepte diferentes tipos en la lista variable de argumentos se debe inventar un método para pasar la información acerca de los tipos de argumentos.

Las herramientas para usar una lista variable de argumentos están definidas en STDARG.H. Estas herramientas se usan dentro de la función para recuperar los argumentos en la lista variable. Son las siguientes:

- `va_list` un tipo de dato apuntador.
- `va_start()` una macro usada para inicializar la lista de argumentos.
- `va_arg()` una macro usada para recuperar cada argumento, en turno, de la lista de variables.
- `va_end()`, una macro usada para limpieza cuando han sido recuperados todos los argumentos.

La manera en que se usan las macros es explicada aquí, seguida de un ejemplo. Cuando la función es llamada accesa sus argumentos de la manera siguiente:

1. Declara una variable de apuntador de tipo `va_list`. Este apuntador se usa para accesar los argumentos individuales. Por lo general, aunque no es requerido, se le denomina a esta variable `arg_ptr`.
2. Llama a la macro `va_start()`, pasándole el apuntador `arg_ptr` así como el nombre del último argumento fijo. La macro `va_start()` no tiene valor de retorno, sino simplemente inicializa al apuntador `arg_ptr` para que apunte al primer argumento de la lista variable.
3. Para recuperar cada argumento llama a `va_arg()`, pasándole el apuntador `arg_ptr` y el tipo de dato del siguiente argumento. El valor de retorno de `va_arg()` es el valor del siguiente argumento. Si la función ha recibido *n* argumentos en la lista variable, llama a `va_arg()` *n* veces para recuperar los argumentos en el orden listado en la llamada de función.
4. Cuando han sido recuperados todos los argumentos de la lista variable llama a `va_end()`, pasándole el apuntador `arg_ptr`. En algunas instrumentaciones esta macro no ejecuta ninguna acción, pero en otras ejecuta las acciones de limpieza necesarias. Se debe tener la costumbre de llamar a `va_end()` en caso de que use una instrumentación de C que la requiera.

Ahora veamos el ejemplo. La función `average()`, del listado 18.3, calcula la media aritmética de una lista de enteros.



Listado 18.3. Uso de una lista variable de argumentos.

```
1: /* Funciones con lista variable de argumentos. */
2:
3: #include <stdio.h>
4: #include <stdarg.h>
5:
6: float average(int num, ...);
7:
```

Listado 18.3. continuación

```
8: main()
9: {
10:     float x;
11:
12:     x = average(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
13:     printf("\nThe first average is %f.", x);
14:
15:     x = average(5, 121, 206, 76, 31, 5);
16:     printf("\nThe second average is %f.", x);
17: }
18:
19: float average(int num, ...)
20: {
21:     /* Declara una variable de tipo va_list. */
22:
23:     va_list arg_ptr;
24:     int count, total = 0;
25:
26:     /* Inicializa el apuntador a argumentos. */
27:
28:     va_start(arg_ptr, num);
29:
30:     /* Recupera cada argumento de la lista variable. */
31:
32:     for (count = 0; count < num; count++)
33:         total += va_arg( arg_ptr, int );
34:
35:     /* Ejecuta la limpieza.*/
36:
37:     va_end(arg_ptr);
38:
39:     /* Divide el total entre la cantidad de valores */
40:     /* para obtener la media aritmética. Asigna el total a tipo */
41:     /* float para que el valor regresado sea tipo float. */
42:
43:     return ((float)total/num);
44: }
```

Salida

```
The first average is 5.500000.
The second average is 87.800003.
```

Análisis

La función `average()` es llamada por primera vez en la línea 19. El primer argumento pasado, el único argumento fijo, especifica la cantidad de valores en la lista variable de argumentos. Cada vez que es recuperado un argumento de la lista variable, en las líneas 32 y 33, es sumado a la variable total. Una vez que han sido recuperados todos los argumentos, la línea 43 especifica el tipo de `total` como `float`, y luego divide a `total` entre `num` para obtener la media aritmética.

Se debe hacer notar otras dos cosas en este listado. La línea 28 llama a `va_start()` para inicializar la lista de argumentos. Esto debe hacerse antes de que sean recuperados los valores. La línea 37 llama a `va_end()` para "hacer la limpieza", debido a que la función ha terminado de utilizar los valores. Ambas funciones deben usarse en los programas.

Hablando estrictamente, una función que acepta una cantidad variable de argumentos no necesita tener un parámetro fijo que le informe de la cantidad de argumentos que le son pasados. Se podría, por ejemplo, marcar el final de la lista de argumentos con un valor especial, que no sea usado en otro lugar. Este método impone limitaciones sobre los argumentos que pueden ser pasados y, por lo tanto, se le evita en lo posible.

Funciones que regresan un apuntador

Ya ha visto varias funciones de la biblioteca estándar del C que regresan un apuntador al programa que las llama. Usted también puede escribir sus propias funciones que regresen un apuntador. Como lo debe suponer, se usa el operador de indirección (*) tanto en la declaración de la función como en la definición de la función. La forma general es

```
tipo *func(lista_de_parámetros);
```

Este enunciado declara a una función, `func()`, que regresa un apuntador a `tipo`. A continuación se dan dos ejemplos concretos:

```
double *func(lista_de_parámetros);
struct dirección *func2(lista_de_parámetros);
```

La primera línea declara una función que regresa un apuntador a tipo `double`. La segunda línea declara una función que regresa un apuntador a tipo `dirección` (que se supone que es una estructura definida por el usuario).

No confunda una *función que regresa un apuntador* y un *apuntador a función.* Si se incluye un par de paréntesis adicionales en la declaración se declara lo segundo.

```
double (*func)(...);      /* Apuntador a función que regresa
                             un double. */

double *func(...);        /* Función que regresa un apuntador
                             a un double*/
```

Ya que no hay duda sobre el formato de la declaración, ¿cómo usa una función que regresa un apuntador? No hay nada especial acerca de tales funciones, se les usa de la misma forma que cualquier otra función, asignando su valor de retorno a una variable del tipo apropiado (que en este caso es un apuntador). Debido a que la llamada de función es una expresión del C, se le puede usar en cualquier lugar que pueda ser usado un apuntador a ese tipo.

El listado 18.4 presenta un ejemplo simple, una función a la que se le pasan dos argumentos y determina cuál es más grande. En el listado dos funciones diferentes ejecutan esta tarea: una regresando un `int` y la otra regresando un apuntador a `int`.

Captura

Listado 18.4. Regreso de un apuntador desde una función.

```
1: /* Función que regresa un apuntador. */
2:
3: #include <stdio.h>
4:
5: int larger1(int x, int y);
6: int *larger2(int *x, int *y);
7:
8: main()
9: {
10:     int a, b, bigger1, *bigger2;
11:
12:     printf("Enter two integer values: ");
13:     scanf("%d %d", &a, &b);
14:
15:     bigger1 = larger1(a, b);
16:     printf("\The larger value is %d.", bigger1);
17:
18:     bigger2 = larger2(&a, &b);
19:     printf("\nThe larger value is %d.", *bigger2);
20: }
21:
22: int larger1(int x, int y)
23: {
24:     if (y > x)
25:         return y;
26:     return x;
27: }
28:
29: int *larger2(int *x, int *y)
30: {
31:     if (*y > *x)
32:         return y;
33:
34:     return x;
35: }
```

Salida

```
Enter two integer values: 1111 3000
The larger value is 3000.
The larger value is 3000.
```

Análisis

Este es un programa relativamente fácil de seguir. Las líneas 5 y 6 contienen los prototipos para las dos funciones. La primera, `larger1()` recibe dos variables `int` y regresa un `int`. La segunda, `larger2()` recibe dos apuntadores a variables `int` y

regresa un apuntador a un `int`. La función `main()`, en las líneas 8 a 20, es directa. La línea 10 declara cuatro variables. `a` y `b` guardan las dos variables a ser comparadas. `bigger1` y `bigger2` guardan los valores de retorno para las funciones `larger1()` y `larger2()` respectivamente. Observe que `bigger2` es un apuntador a un `int` y `bigger1` es simplemente un `int`.

La línea 15 llama a `larger1()` con los dos `int`, `a` y `b`. El valor regresado por la función es asignado a `bigger1`, que es impreso en la línea 16. La línea 18 llama a `larger2()` con las direcciones de los dos `int`. El valor regresado por `larger2()`, un apuntador, es asignado a `bigger2`, que también es un apuntador. Este valor es desreferenciado e impreso en la siguiente línea.

Las dos funciones de comparación son muy similares. Ambas comparan los dos valores. El valor más grande es regresado. La diferencia entre las funciones está en `larger2()`. En esta función los valores apuntados son comparados en la línea 31. Luego es regresado el apuntador a la variable del valor más grande. Observe que se utiliza el operador de desreferencia en la comparación, pero no en los enunciados `return` de las líneas 32 a 34.

En muchos casos, como en los ejemplos anteriores, es igualmente factible escribir una función para regresar un valor o un apuntador. Lo que se seleccione depende del caso específico del programa, principalmente la manera en que pretende usar el valor de retorno.

DEBE	NO DEBE
DEBE Usar todos los elementos descritos anteriormente cuando escriba funciones con argumentos variables. Esto es cierto, incluso si su compilador no requiere todos los elementos. Las partes son `va_list`, `va_start()`, `va_arg()` y `va_end()`.	**NO DEBE** Confundir los apuntadores a funciones con las funciones que regresan apuntadores.

Resumen

En este capítulo usted aprendió algunas cosas adicionales que pueden hacer los programas en C con las funciones. Aprendió la diferencia entre pasar argumentos por valor y por referencia, y la manera en que la segunda técnica le permite a una función "regresar" más de un valor al programa que la llama. También vio la manera en que puede ser usado el tipo `void`, para crear un apuntador genérico que pueda apuntar a cualquier tipo de objeto de dato del C. Los apuntadores tipo `void` se usan, por lo general, con funciones a las que pueden serles pasados argumentos que no estén restringidos a un solo tipo de dato. Recuerde que a un apuntador a tipo `void` le debe ser especificado el tipo antes de que se le pueda desreferenciar.

Este capítulo también le mostró cómo usar las macros definidas en STDARG.H para escribir una función que acepte una cantidad variable de argumentos. Estas funciones proporcionan bastante flexibilidad a la programación. Por último vio cómo escribir una función que regrese un apuntador.

Preguntas y respuestas

1. ¿Es práctica común en la programación en C pasar apuntadores?

   ¡Claro que sí! En muchos casos una función necesita cambiar varias variables y hay dos formas en que se puede lograr. La primera es declarar y usar variables globales. La segunda es pasar apuntadores para que la función pueda modificar directamente los datos. La primera opción es buena solamente si casi todas las funciones van a usar la variable. (Vea el Día 12, "Alcance de las variables".)

2. ¿Es mejor modificar un valor regresándolo o pasando un apuntador al dato actual?

   Cuando necesita modificar solamente un valor con una función, por lo general es mejor regresar el valor de la función en vez de pasar un apuntador a la función. La lógica de esto es simple. Al no pasar un apuntador no se corre el riesgo de cambiar cualquier dato que no se pretenda cambiar, y se mantiene a la función independiente del resto del código.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. Cuando se pasan argumentos a una función, ¿cuál es la diferencia entre pasarlos por valor y pasarlos por referencia?
2. ¿Qué es un apuntador a tipo `void`?
3. ¿Cuál es la razón para usar un apuntador a tipo `void`?
4. Cuando se usa un apuntador a `void`, ¿qué significa la especificación de tipo y cuándo debe ser usada?
5. ¿Se puede escribir una función que tome una lista variable de argumentos solamente, sin argumentos fijos?

6. ¿Qué macros deben usarse cuando se escriben funciones con lista variable de argumentos?
7. ¿Qué valor se añade a un apuntador a `void` cuando es incrementado?
8. ¿Puede una función regresar un apuntador?

Ejercicios

1. Escriba el prototipo para una función que regrese un entero. Debe tomar un apuntador a un arreglo de caracteres como argumento.
2. Escriba un prototipo para una función, llamada `números`, que tome tres argumentos enteros. Los enteros deben ser pasados por referencia.
3. Muestre la manera en que podría llamar a la función `números`, del ejercicio dos, con los tres enteros `int1`, `int2` e `int3`.
4. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
void cuadrado(void *num)
{
   *num *= *num;
}
```

5. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
float total( int num, ...)
{
   int contador, total = 0;
   for ( contador = 0; contador < num; contador++ )
         total += va_arg( arg_ptr, int );
   return ( total );
}
```

Debido a que hay muchas soluciones posibles, no se proporcionan respuestas para los siguientes ejercicios.

6. Escriba una función que a) le sea pasado un número variable de cadenas como argumentos; b) concatene las cadenas en orden para hacer una cadena más larga y, c) regrese al programa que la llama un apuntador a la nueva cadena.

7. Escriba una función que a) le sea pasado un arreglo de cualquier tipo de dato numérico como argumento; b) encuentre los valores mayor y menor en el arreglo y, c) regrese al programa que llama apuntadores a estos valores. (Consejo: necesitará alguna manera de decirle a la función qué tantos elementos hay en el arreglo.)
8. Escriba una función que acepte una cadena y un carácter. La función deberá buscar la primera aparición del carácter en la cadena y regresar un apuntador a esa posición.

DIA 19

Exploración de la biblioteca de funciones

Como ha visto a lo largo de este libro, gran parte de la potencia del C le viene de las funciones de la biblioteca estándar. En este capítulo exploraremos algunas de las funciones que no caen en el tema de otros capítulos. Hoy aprenderá

- Funciones matemáticas.
- Funciones que manejan el tiempo.
- Funciones para manejo de errores.
- Funciones para búsqueda y ordenamiento de datos.

Funciones matemáticas

La biblioteca estándar del C contiene una diversidad de funciones que ejecutan operaciones matemáticas. Los prototipos para las funciones matemáticas están en el archivo de encabezado MATH.H. Todas las funciones matemáticas regresan un tipo `double`. Para las funciones trigonométricas los ángulos están expresados en *radianes*. Recuerde que un radián es igual a 57.296 grados, y un círculo completo (360 grados) contiene 2 radianes.

Funciones trigonométricas

- `double acos(double x)`

  La función `acos()` regresa el arco coseno de su argumento. El argumento debe estar en el rango `-1 <= x <= 1`, y el valor de retorno está en el rango `0 <= acos <= p`.

- `double asin(double x)`

  La función `asin()` regresa el arco seno de su argumento. El argumento debe estar en el rango `-1 <= x <= 1`, y el valor de retorno está en el rango `- /2 <= asin <= /2`.

- `double atan(double x)`

  La función `atan()` regresa el arco tangente de su argumento. El valor de retorno está en el rango `- /2 <= atan <= /2`.

- `double atan2(double x, double y)`

  La función `atan2()` regresa el arco tangente de `x/y`. El valor regresado está en el rango `- <= atan2 <=`.

- `double cos(double x)`

  La función `cos()` regresa el coseno de su argumento.

- `double sin(double x)`

  La función `sin()` regresa el seno de su argumento.

- `double tan(double x)`

  La función `tan()` regresa la tangente de su argumento.

Funciones exponenciales y logarítmicas

- `double exp(double x)`

  La función `exp()` regresa el exponente natural de su argumento, esto es, `ex`, donde `e == 2.7182818284590452354`.

- `double log(double x)`

  La función `log()` regresa el logaritmo natural de su argumento. El argumento debe ser mayor que 0.

- `double log10(double x)`

  La función `log10()` regresa el logaritmo base 10 de su argumento. El argumento debe ser mayor que 0.

- `double frexp(double x, int *y)`

  La función `frexp()` calcula la fracción normalizada que representa el valor de `x`. El valor de retorno de la función, `r`, es una fracción en el rango `0.5 <= r <= 1.0`. La función asigna a `y` un exponente entero tal que `x = r * 2y`. Si el valor pasado a la función es `0`, tanto `r` como y son `0`.

- `double ldexp(double x, int y)`

  La función `ldexp()` regresa `x * 2y`.

Funciones hiperbólicas

Las funciones hiperbólicas ejecutan cálculos trigonométricos hiperbólicos.

- `double cosh(double x)`

  La función `cosh()` regresa el coseno hiperbólico de su argumento.

- `double sinh(double x)`

  La función `sinh()` regresa el seno hiperbólico de su argumento.

- `double tanh(double x)`

  La función `tanh()` regresa la tangente hiperbólica de su argumento.

Otras funciones matemáticas

Esta sección lista algunas funciones matemáticas diversas.

- `double sqrt(double x)`

  La función `sqrt()` regresa la raíz cuadrada de su argumento. El argumento debe ser 0 o mayor.

- `double ceil(double x)`

  La función `ceil()` regresa el entero más pequeño que no es menor que su argumento. Por ejemplo, `ceil(4.5)` regresa `5.0` y `ceil(-4.5)` regresa `-4.0`. Aunque `ceil()` regresa un valor entero, es regresado como tipo `double`.

- `int abs(int x)` y `long int labs(long int x)`

  Las funciones `abs()` y `labs()` regresan el valor absoluto de sus argumentos.

- `double floor(double x)`

  La función `floor()` regresa el entero más grande que no es mayor que su argumento. Por ejemplo, `floor(4.5)` regresa `4.0` y `floor(-4.5)` regresa `-5.0`.

- `double modf(double x, double *y)`

  La función `modf()` divide a `x` en sus partes entera y fraccionaria, cada una con el mismo signo que `x`. La parte fraccionaria es regresada por la función y la parte entera es asignada a `*y`.

- `double pow(double x, double y)`

  La función `pow()` regresa x^y. Sucede un error si `x == 0` y `y <= 0` o si `x < 0` y `y` no es un entero.

- `double fmod(double x, double y)`

  La función `fmod()` regresa el resíduo en punto flotante de `x/y`, con el mismo signo que `x`. La función regresa `0` si `x == 0`.

Se podría llenar un libro entero con programas que muestren todas las funciones matemáticas. El listado 19.1 contiene un solo programa que muestra unas cuantas de las funciones.

Captura

Listado 19.1. Uso de las funciones matemáticas de la biblioteca del C.

```
1: /* Demuestra algunas funciones matemáticas. */
2:
3: #include <stdio.h>
4: #include <math.h>
5:
6: main()
7: {
8:
9:     double x;
10:
11:     printf("Enter a number: ");
12:     scanf( "%lf", &x);
13:
14:     printf("\n\nOriginal value: %lf", x);
15:
16:     printf("\nCeil: %lf", ceil(x));
17:     printf("\nFloor: %lf", floor(x));
18:     if( x >= 0 )
19:         printf("\nSquare root: %lf", sqrt(x) );
20:     else
21:        printf("\nNegative number" );
22:
23:     printf("\nCosine: %lf", cos(x));
24: }
```

Salida

```
Enter a number: 100.95
Original value: 100.950000
Ceil: 101.000000
Floor: 100.000000
Square root: 10.047388
Cosine: 0.913482
```

Análisis

Este listado usa sólo algunas funciones matemáticas. Un valor, aceptado en la línea 12, es impreso después de que es enviado a cuatro de las funciones matemáticas, `ceil()`, `floor()`, `sqrt()` y `cos()`. Observe que `sqrt()` es llamada solamente, si el número no es negativo. No se puede obtener la raíz cuadrada de un número negativo. Cualquiera de las otras funciones matemáticas podría haber sido añadida a un programa como éste para probar su funcionamiento.

Manejo del tiempo

La biblioteca del C contiene varias funciones que le permiten al programa trabajar con el tiempo. Los prototipos de función y la definición de la estructura usada por muchas de las funciones de tiempo están en el archivo de encabezado TIME.H.

Representación del tiempo

Las funciones de tiempo del C representan al tiempo en dos formas. El método más elemental es la cantidad de segundos transcurridos desde la medianoche del 1de enero de 1970. Los valores negativos se usan para representar el tiempo anterior a esa fecha.

Estos valores de tiempo son guardados como enteros tipo `long`. En TIME.H los símbolos `time_t` y `clock_t` están definidos, mediante un enunciado `typedef`, como `long`. Estos símbolos se usan en vez de `long` en los prototipos de funciones de tiempo.

El segundo método representa el tiempo dividido en sus componentes: año, mes, día, etc. Para este tipo de representación las funciones de tiempo usan una estructura, `tm`, definida en TIME.H de la manera siguiente:

```
struct tm
{ int tm_sec,       /* segundos 0..59 */
      tm_min,       /* minutos 0..59 */
      tm_hour,      /* hora del día 0..23 */
      tm_mday,      /* día del mes 1..31 */
      tm_mon,       /* mes 0..11 */
      tm_year,      /* años desde 1900 */
      tm_wday,      /* día de la semana, 0..6 (Domingo..Sábado) */
      tm_yday,      /* día del año, 0..365 */
      tm_isdst;     /* > 0 si es hora para ahorro de luz de día (DST) */
                    /* == 0 si no es DST */
                    /* < 0 si no se sabe */
};
```

Las funciones de tiempo

Esta sección describe las diversas funciones de la biblioteca del C que tienen que ver con el tiempo. El término *tiempo* se refiere tanto a las fechas como a las horas, minutos y segundos. Un programa de muestra se encuentra a continuación de esta descripción.

Obtención del tiempo actual

Para obtener el tiempo actual, como está puesto en el reloj interno del sistema, use la función `time()`. El prototipo es

```
time_t time(time_t *timeptr);
```

Recuerde que `time_t` está definido en TIME.H como un sinónimo para `long`. La función `time()` regresa la cantidad de segundos transcurridos desde la medianoche del 1 de enero de 1970. Si le es pasado un apuntador que no es `NULL`, `time()` también guarda este valor en la variable tipo `time_t` apuntada por `timeptr`. Por lo tanto, para guardar el tiempo actual en la variable ahora, tipo `time_t`, se podría escribir:

```
time_t ahora;

ahora = time(0);
```

O también:

```
time_t ahora;
time_t *ptr_ahora = &ahora;
time(ptr_ahora);
```

Conversión entre representaciones de tiempo

Ya que saber la cantidad de segundos desde el 1 de enero de 1970 no es muy útil, el tiempo representado como un valor `time_t` puede convertirse a una estructura `tm` mediante la función `localtime()`. Una estructura `tm` contiene el día, el mes y el año y otra información relativa al tiempo en un formato más adecuado para el despliegue y la impresión. El prototipo para esta función es

```
struct tm *localtime(time_t *ptr);
```

Esta función regresa un apuntador a una estructura estática tipo `tm`, por lo que no se necesita declarar una estructura tipo `tm` para usarla, sino solamente un apuntador a tipo `tm`. Esta estructura estática es reutilizada y sobreescrita cada vez que se llama a `localtime()`. Si se quiere guardar el valor regresado, el programa debe declarar una estructura tipo `tm` por separado y copiar los valores de la estructura estática.

La conversión inversa, de una estructura tipo `tm` a un valor tipo `time_t`, se ejecuta por la función `mktime()`. El prototipo es

```
time_t mktime(struct tm *ntime);
```

La función regresa la cantidad de segundos entre la medianoche del 1 de enero de 1970 y el tiempo representado por la estructura tipo `tm` que es apuntada por `ntime`.

Desplegado de tiempo

Para convertir tiempos en cadenas formateadas, adecuadas para el despliegue, use las funciones `ctime()` y `asctime()`. Ambas regresan el tiempo como una cadena con un formato específico. Su diferencia se debe a que a `ctime()` se le pasa el tiempo como un valor tipo `time_t`, y a `asctime()` se le pasa el tiempo como una estructura tipo `tm`. Sus prototipos son:

```
char *asctime(struct tm *ptr);
char *ctime(time_t *ptr);
```

Ambas funciones regresan un apuntador a una cadena estática de 26 caracteres terminada en el carácter nulo, que da el tiempo del argumento de la función en el siguiente formato:

```
Thu Jun 13 10:22:23 1991
```

El formato del tiempo es el militar, de 24 horas. Ambas funciones usan una cadena estática, sobreescribiéndola cada vez que son llamadas.

Para un mayor control sobre el formato del tiempo use la función `strftime()`. A esta función se le pasa el tiempo como una estructura tipo `tm`. Formatea al tiempo de acuerdo con una cadena de formato. El prototipo de función es:

```
size_t strftime(char *s, size_t max, char *fmt, struct tm
                *ptr);
```

La función toma el tiempo de la estructura tipo `tm` apuntada por `ptr`, lo formatee de acuerdo con la cadena de formato `fmt` y escribe el resultado como una cadena terminada en nulo en la posición de la memoria apuntada por `s`. El argumento `max` debe especificar la cantidad de espacio asignado a `s`. Si el tamaño de la cadena resultante (incluso el carácter nulo terminal) tiene más caracteres que los indicados por `max`, la función regresa `0` y la cadena `s` es inválida. En caso contrario la función regresa la cantidad de caracteres escritos, `strlen(s)`.

La cadena de formato consiste en uno o más especificadores de conversión, de acuerdo con la tabla 19.1.

Tabla 19.1. Especificadores de conversión que pueden usarse con `strftime()`.

Especificador	Es reemplazado por
`%a`	Nombre de día de la semana abreviado.
`%A`	Nombre de día de la semana completo.
`%b`	Nombre de mes abreviado.
`%B`	Nombre de mes completo.
`%c`	Representación de fecha y hora (por ejemplo, 10:41:50. 30-Jun-91).
`%d`	Día de mes en número decimal 01-31.
`%H`	La hora (en reloj de 24 horas) como número decimal 00-23.
`%I`	La hora (en reloj de 12 horas) como número decimal 00-11.

Especificador	Es reemplazado por
%j	El día del año como número decimal 001-366.
%m	El mes como número decimal 01-12.
%M	El minuto como número decimal 00-59.
%p	AM o PM.
%S	El segundo como número decimal 00-59.
%U	La semana del año como número decimal 00-53. El domingo está considerado como el primer día de la semana.
%w	El día de la semana como número decimal 0-6 (el domingo = 0).
%W	La semana del año como número decimal 00-53. El lunes es considerado como el primer día de la semana.
%x	La representación de fecha (por ejemplo, 30-Jun-91).
%X	La representación de hora (por ejemplo, 10:41:50).
%y	El año sin centuria como número decimal 00-99.
%Y	El año con centuria como número decimal.
%z	El nombre de la zona de tiempo, si se dispone de la información, o blanco en caso contrario.
%%	Un signo de porcentaje solo.

Cálculo de diferencias de tiempo

Se puede calcular la diferencia, en segundos, entre dos tiempos con la macro `difftime()`, que resta dos valores `time_t` y regresa la diferencia. El prototipo es

```
double difftime(time_t despues, time_t antes);
```

La función resta `antes` de `después` y regresa la diferencia, la cantidad de segundos transcurridos entre los dos tiempos. Un uso común para `difftime()` es el cálculo del tiempo transcurrido, como se muestra, junto con otras operaciones de tiempo, en el listado 19.2.

Se puede determinar la duración en una forma diferente con la función `clock()`, que regresa la cantidad de tiempo transcurrido desde que el programa inició la ejecución, en unidades de 1/100 de segundo. El prototipo es:

```
clock_t clock(void);
```

Para determinar la duración de alguna parte del programa llame a `clock()` dos veces, antes y después de que suceda el proceso, y reste los dos valores de retorno.

Uso de las funciones de tiempo

El programa del listado 19.2 muestra la manera de usar las funciones de tiempo de la biblioteca del C.

Captura

Listado 19.2. Uso de las funciones de tiempo de la biblioteca del C.

```
1: /* Demuestra las funciones de tiempo. */
2:
3: #include <stdio.h>
4: #include <time.h>
5:
6: main()
7: {
8:     time_t start, finish, now;
9:     struct tm *ptr;
10:     char *c, buf1[80];
11:     double duration;
12:
13:     /* Registra el tiempo en el que el programa inicia la ejecución. */
14:
15:     start = time(0);
16:
17:     /* Registra el tiempo actual con el método alterno de */
18:     /* llamar a time(). */
19:
20:     time(&now);
21:
22:     /* Convierte el valor time_t en una estructura tipo tm. */
23:
24:     ptr = localtime(&now);
25:
26:     /* Crea y despliega una cadena formateada con */
27:     /* el tiempo actual. */
28:
29:     c = asctime(ptr);
30:     puts(c);
31:     getch();
32:
33:     /* Ahora usa la función strftime() para crear varias */
34:     /* versiones formateadas diferentes del tiempo. */
35:
36:     strftime(buf1, 80, "This is week %U of the year %Y", ptr);
```

```
37:     puts(buf1);
38:     getch();
39:
40:     strftime(buf1, 80, "Today is %A, %x", ptr);
41:     puts(buf1);
42:     getch();
43:
44:     strftime(buf1, 80, "It is %M minutes past hour %I.", ptr);
45:     puts(buf1);
46:     getch();
47:
48:     /* Ahora obtiene el tiempo actual y calcula la duración
               del programa. */
49:
50:     finish = time(0);
51:     duration = difftime(finish, start);
52:     printf("\nProgram execution time = %f seconds.", duration);
53:
54:     /* También despliega la duración del programa en */
55:     /* centésimas de segundo usando clock(). */
56:
57:     printf("\nProgram execution time = %ld hundredths of sec.",
58:         clock());
59: }
```

Salida

```
Sun Aug 23 19:54:25 1992
This is week 34 of the year 1992
Today is Sunday, Sun Aug 23, 1992
It is 54 minutes past hour 07.
Program execution time = 22.000000 seconds.
Program execution time = 401 hundredths of sec.
```

Análisis

Este programa tiene muchas líneas de comentario, por lo que debe ser fácil seguirlo. Debido a que se usan las funciones de tiempo, en la línea 4 se incluye al archivo de encabezado TIME.H. La línea 8 declara tres variables de tipo `time_t`, llamadas `start`, `finish` y `now`. Estas variables pueden guardar el tiempo, en segundos, como un desplazamiento desde el 1 de enero de 1970. La línea 9 declara un apuntador a una estructura `tm`. La estructura `tm` ya se describió anteriormente. El resto de las variables tienen tipos que deben serle familiares.

El programa registra su tiempo de inicio en la línea 15. Esto lo logra con una llamada a `time()`. Luego el programa hace virtualmente la misma cosa en forma diferente. En vez de usar el valor regresado por la función `time()`, la línea 20 pasa a `time()` un apuntador a la variable `now`. La línea 24 hace exactamente lo que dice el comentario de la línea 22. Convierte el valor tipo `time_t` de `now` a una estructura `tm`. Las siguientes secciones del programa imprimen en la pantalla el valor del tiempo actual en diversos formatos. La línea 29 usa la función `asctime()` para asignar la información a un apuntador a carácter, `c`. La línea 30 imprime la información formateada. Luego, el programa espera a que se oprima un carácter.

Las líneas 36 a 46 usan la función `strftime()` para imprimir la fecha en tres diferentes formatos. Con la tabla 19.1 usted debe ser capaz de determinar lo que imprimen estas líneas.

Luego, el programa vuelve a determinar el tiempo en la línea 50. Este es el tiempo de finalización del programa. La línea 51 usa este tiempo de finalización, junto con el tiempo de inicio, para calcular la duración del programa. Este valor se imprime en la línea 52. El programa termina imprimiendo el tiempo de la ejecución del programa a partir de la función `clock()`.

Funciones para el manejo de errores

La biblioteca estándar del C contiene diversas funciones y macros que le ayudan a manejar errores del programa.

La función *assert()*

La macro `assert()` puede diagnosticar errores del programa. Es definida en ASSERT.H y su prototipo es:

```
void assert(int expression);
```

El argumento `expression` puede ser cualquier cosa que se quiera probar, una variable o cualquier expresión de C. Si la `expression` evalúa a `TRUE`, `assert` no hace nada. Si la `expression` evalúa a `FALSE`, `assert()` despliega un mensaje de error en `stderr` y termina la ejecución del programa.

¿Cómo se usa `assert()`? Su uso más frecuente es para descubrir errores de programa (diferentes de los errores de compilación). Por ejemplo, tal vez un programa de análisis financiero que se está escribiendo ocasionalmente dé respuestas incorrectas. Se sospecha que el problema es a causa de la variable `tasa_interés` cuando toma un valor negativo, lo cual nunca debiera suceder. Para revisar esto ponga el enunciado

```
assert(tasa_interés >=0);
```

en posiciones del programa donde se use `tasa_interés`. Si la variable llega a ser negativa en algún momento, la macro `assert()` le avisa.

Para ver el funcionamiento de `assert()` ejecute el programa del listado 19.3. Si se da un valor diferente de cero, el programa despliega el valor y termina normalmente. Si se da cero, la macro `assert()` fuerza la terminación anormal del programa. El mensaje de error que se ve desplegado es

```
Assertion failed: x, file list1903.c, line 13
```

Captura

Listado 19.3. Uso de la macro `assert()`.

```
1: /* La macro assert(). */
2:
3: #include <stdio.h>
4: #include <assert.h>
5:
6: main()
7: {
8:     int x;
9:
10:     printf("\nEnter an integer value: ");
11:     scanf("%d", &x);
12:
13:     assert(x);
14:
15:     printf("You entered %d.", x);
16: }
```

Salida

```
>list1903
Enter an integer value: 10
You entered 10.
>list1903

Enter an integer value: 0
Assertion failed: x, file list1903.c, line 13
Abnormal program termination
```

Análisis

Ejecute este programa para ver que el mensaje de error, desplegado por `assert()` en la línea 13, incluye la expresión cuya prueba falló, el nombre de archivo y el número de línea, donde se encuentra el `assert()`.

La acción de `assert()` depende de otra macro llamada NDEBUG (por "no debugging", no depuración). Si la macro NDEBUG no está definida (la opción por omisión) `assert()` está activo. Si NDEBUG está definida, `assert()` está desactivado y no tiene efecto. Si se pone a `assert()` en varias posiciones del programa para ayudarle en la depuración y luego se resuelve el problema, se puede definir a NDEBUG para desactivar a `assert()`. Esto es mucho más fácil que revisar todo el programa para quitar los enunciados `assert()` (y sólo para descubrir posteriormente que los vuelve a necesitar). Para definir la macro NDEBUG use la directiva `#define`. Se puede demostrar esto añadiendo el renglón

```
#define NDEBUG
```

al listado 19.3 en la línea 2. Ahora el programa imprime el valor tecleado y termina normalmente, incluso si se teclea `0`.

Observe que NDEBUG no necesita definirse como algo en particular, siempre y cuando sea incluido en una directiva #define. Aprenderá más acerca de la directiva #define en el Día 21, "Cómo aprovechar las directivas del preprocesador y más".

El archivo de encabezado ERRNO.H

El archivo de encabezado ERRNO.H define varias macros usadas para definir y documentar errores del momento de ejecución. Estas macros se usan junto con la función perror(), que se describe en los siguiente párrafos.

Las definiciones de ERRNO.H incluyen un entero externo llamado errno. Muchas de las funciones de biblioteca del C asignan un valor a esta variable cuando sucede un error durante la ejecución de la función. El archivo ERRNO.H también define un grupo de constantes simbólicas para estos errores, que se listan en la tabla 19.2.

Tabla 19.2. Las constantes simbólicas de error definidas en ERRNO.H.

Nombre	Valor	Mensaje y significado
E2BIG	1000	Lista de argumentos demasiado larga (la longitud de la lista excede 128 bytes).
EACCES	5	Permiso negado (por ejemplo, el tratar de escribir a un archivo abierto para sólo lectura).
EBADF	6	Mal descriptor de archivo.
EDOM	1002	Argumento matemático fuera de dominio (un argumento pasado a una función matemática estaba fuera del rango permitido).
EEXIST	80	El archivo existe.
EMFILE	4	Demasiados archivos abiertos.
ENOENT	2	No hay tal archivo o directorio.
ENOEXEC	1001	Error de formato de Exec.
ENOMEM	8	No hay suficiente memoria (por ejemplo, no hay suficiente memoria para ejecutar la función exec()).
ENOPATH	3	No se encontró la ruta.
ERANGE	1003	Resultado fuera de rango (por ejemplo, el resultado regresado por una función matemática es demasiado grande o pequeño para el tipo de dato de retorno).

Se puede usar a errno de dos maneras. Algunas funciones señalan, por medio de su valor de retorno, que ha ocurrido un error. Si esto sucede, se puede revisar el valor de errno para determinar la naturaleza del error y tomar la acción adecuada. En otras ocasiones, cuando no se tiene indicación específica de que ha sucedido algún error, se puede revisar a errno. Si es diferente de cero, ha sucedido algún error, y el valor específico de errno indica su naturaleza. Asegúrese de restaurar a errno a cero después de haber manejado el error. Después de explicar perror() se ilustra el uso de errno en el listado 19.4.

La función *perror()*

La función perror() es otra de las herramientas para el manejo de errores del C. Cuando es llamada, perror() despliega un mensaje en stderr, describiendo el error más reciente que ha ocurrido durante una llamada a una función de biblioteca o a una llamada de sistema. El prototipo en STDIO.H es:

```
void perror(char *msg);
```

El argumento msg apunta a un mensaje opcional definido por el usuario. Este mensaje se imprime primero, seguido de dos puntos y el mensaje, definido en la instrumentación, que describe el error más reciente. Si se llama a perror() cuando no ha sucedido ningún error, el mensaje desplegado es no error.

Una llamada a perror() no hace nada para manejar la condición de error. Le toca al programa ejecutar alguna acción, que puede consistir en pedirle al usuario que haga algo, como por ejemplo terminar el programa. La acción que ejecuta el programa puede ser determinada revisando el valor de errno y la naturaleza del error. Tome en cuenta que un programa no necesita incluir al archivo de encabezado ERRNO.H para usar la variable externa errno. Ese archivo de encabezado se requiere solamente si el programa constantes simbólicas de error listadas en la tabla 19.2.



Listado 19.4. Uso de perror() y errno para manejar errores del momento de ejecución.

```
1: /* Demostración del manejo de errores con perror() y errno. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <errno.h>
6:
7: main()
8: {
9:     FILE *fp;
10:     char filename[80];
11:
12:     printf("Enter filename: ");
13:     gets(filename);
14:
```

Listado 19.4. continuación

```
15:     if (( fp = fopen(filename, "r")) == NULL)
16:     {
17:         perror("You goofed!");
18:         printf("errno = %d.", errno);
19:         exit(1);
20:     }
21:     else
22:     {
23:         puts("File opened for reading.");
24:         fclose(fp);
25:     }
26: }
```

Salida

```
>list1904
Enter file name: list1904.c
File opened for reading.

>list1904
Enter file name: notafile.xxx
You goofed!: No such file or directory
errno = 2.
```

Análisis

Este programa imprime uno de dos mensajes, basándose en si un archivo puede ser abierto para lectura. La línea 15 abre el archivo. Si el archivo se abre, ejecuta la parte de `else` del ciclo `if`, imprimiendo el siguiente mensaje:

```
File opened for reading.
```

Si sucede algún error en la apertura del archivo, como, por ejemplo, que el archivo no exista, se ejecutan las líneas 17 a 19 del ciclo `if`. La línea 17 llama a la función `perror()` con el texto `"You goofed!"`. Esto es seguido por la impresión del número de error. El resultado de teclear un archivo que no existe es

```
You goofed!: No such file or directory.
errno = 2
```

DEBE	NO DEBE
DEBE Incluir el archivo de encabezado ERRNO.H, si va a hacer uso de los errores simbólicos de la tabla 19.2.	
NO DEBE Incluir el archivo de encabezado ERRNO.H si no va a usar las constantes simbólicas de error descritas en la tabla 19.2.	

DEBE Verificar los errores posibles en el programa. Nunca suponga que todo funciona bien.

Búsqueda y ordenamiento

Entre las tareas más comunes que deben ejecutar los programas están buscar y ordenar datos. La biblioteca estándar del C contiene funciones con fines generales que se pueden usar para cada tarea.

Búsqueda con *bsearch()*

La función de biblioteca `bsearch()` ejecuta una búsqueda binaria en los datos de un arreglo, buscando un elemento de arreglo que concuerde con una clave. Para usar `bsearch()`, el arreglo debe estar ordenado en forma ascendente. También el programa debe proporcionar la función de comparación, que será usada por `bsearch()` para determinar si un elemento de dato es mayor, menor o igual a otro elemento. El prototipo de `bsearch()` está en STDLIB.H.

```
void *bsearch(void *key, void *base, size_t num, size_t width,
              int (*cmp)(void *elemento1, void *elemento2));
```

Este es un prototipo bastante complejo, por lo que habrá que verlo cuidadosamente. El argumento `key` es un apuntador al elemento de datos que es buscado y `base` es un apuntador al primer elemento del arreglo que es revisado. Ambos son declarados como apuntadores tipo `void`, para que puedan apuntar a cualquiera de los objetos de datos del C.

El argumento `num` es la cantidad de elementos del arreglo y `width` es el tamaño (en bytes) de cada elemento. El especificador de tipo, `size_t`, se refiere al tipo de dato regresado por el operador `sizeof()`, que es `unsigned`. El operador `sizeof()` se usa, por lo general, para obtener los valores de `num` y `width`.

El argumento final, `cmp`, es un apuntador a la función de comparación. Esta puede ser una función escrita por el usuario o, cuando se buscan datos de cadenas, la función de biblioteca `strcmp()`. La función de comparación debe satisfacer los siguientes criterios: 1) se le pasan apuntadores a dos elementos de datos y, 2) regresa un tipo `int` de la manera siguiente:

```
< 0   elemento1 es menor que elemento2.
0     elemento1 = elemento2.
> 0   elemento1 es mayor que elemento2.
```

El valor de retorno de `bsearch()` es un apuntador tipo `void`. La función regresa un apuntador al primer elemento del arreglo que concuerda con la clave, o `NULL` si no encuentra concordancia. Se debe hacer una especificación de tipo sobre el apuntador regresado, para que apunte al tipo adecuado antes de usarlo.

El operador `sizeof()` puede proporcionar los argumentos `num` y `width` de la siguiente manera. Si `arreglo[]` es el arreglo que ha de ser revisado, el enunciado

```
sizeof(arreglo[0]);
```

regresa el valor para `width`, el tamaño (en bytes) de un elemento de arreglo. Debido a que la expresión `sizeof(arreglo)` regresa el tamaño, en bytes, del arreglo completo, el enunciado

```
sizeof(arreglo)/sizeof(arreglo[0])
```

obtiene el valor de `num`, la cantidad de elementos en el arreglo.

El algoritmo de búsqueda binaria es muy eficiente y puede buscar rápidamente en un arreglo grande. Su operación depende de que el arreglo esté en orden ascendente. Esta es la manera en que funciona el algoritmo:

1. La clave es comparada con el elemento que está a la mitad del arreglo. Si concuerda, se termina la búsqueda. En caso contrario, la clave debe ser mayor o menor que el elemento de arreglo.
2. Si la clave es menor que el elemento de arreglo, el elemento concordante, en caso de haberlo, debe estar localizado en la primera mitad del mismo. De manera similar, si la clave es mayor que el elemento de arreglo, el elemento concordante debe estar localizado en la segunda mitad del arreglo.
3. Se restringe la búsqueda a la mitad adecuada del arreglo y se regresa al paso uno.

Se puede ver que cada comparación ejecutada por una búsqueda binaria elimina la mitad del arreglo que se está revisando. Por ejemplo, un arreglo de 1,000 elementos puede revisarse con sólo 10 comparaciones, y un arreglo de 16,000 elementos con sólo 14 comparaciones. Por lo general, una búsqueda binaria requiere n comparaciones para buscar en un arreglo de 2^n elementos.

Ordenamiento con *qsort()*

La función de biblioteca `qsort()` es una instrumentación del *algoritmo quicksort*, inventado por C.A.R. Hoare. La función pone un arreglo en orden y, por lo general, el resultado es orden ascendente, pero `qsort()` también puede ser usada para orden descendente. El prototipo de función, definido en STDLIB.H es

```
void qsort(void *base, size_t num, size_t size,
           int (*cmp)(void *elemento1, void *elemento2));
```

El argumento `base` apunta al primer elemento del arreglo, `num` es la cantidad de elementos en el arreglo y `size` es el tamaño (en bytes) de un elemento del arreglo. El argumento `cmp` es un apuntador a una función de comparación. Las reglas para la función de comparación son las mismas que para la función de comparación usada por `bsearch()`, descritas en la sección anterior, y frecuentemente se usa la misma función de comparación, tanto en `bsearch()` como en `qsort()`. La función `qsort()` no tiene valor de retorno.

Dos demostraciones de búsqueda y ordenamiento

El programa del listado 19.5 muestra el uso de `qsort()` y `bsearch()`. El programa ordena y busca en un arreglo de valores.



Listado 19.5. Uso de las funciones `qsort()` y `bsearch()` con valores.

```
1: /* Uso de qsort() y bsearch() con valores. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: #define MAX 20
7:
8: int intcmp(const void *v1, const void *v2);
9:
10: main()
11: {
12:     int arr[MAX], count, key, *ptr;
13:
14:     /* El usuario teclea algunos enteros. */
15:
16:     printf("Enter %d integer values; press Enter after \
            each.\n", MAX);
17:
18:     for (count = 0; count < MAX; count++)
19:         scanf("%d", &arr[count]);
20:
21:     puts("Press a key to sort the values.");
22:     getch();
23:
24:     /* Ordena el arreglo en forma ascendente. */
25:
26:     qsort(arr, MAX, sizeof(arr[0]), intcmp);
27:
28:     /* Despliega el arreglo ordenado. */
29:
```

Listado 19.5. continuación

```
30:     for (count = 0; count < MAX; count++)
31:         printf("\narr[%d] = %d.", count, arr[count]);
32:
33:     puts("\nPress a key to continue.");
34:     getch();
35:
36:     /* Se teclea una clave por buscar. */
37:
38:     printf("Enter a value to search for: ");
39:     scanf("%d", &key);
40:
41:     /* Ejecuta la búsqueda. */
42:
43:     ptr = (int *)bsearch(&key, arr, MAX,
                             sizeof(arr[0]),intcmp);
44:
45:     if ( ptr != NULL )
46:         printf("%d found at arr[%d].", key, (ptr - arr));
47:     else
48:         printf("%d not found.", key);
49: }
50:
51: int intcmp(const void *v1, const void *v2)
52: {
53:     return (*(int *)v1 - *(int *)v2);
54: }
```

Salida

```
Enter 20 integer values; press Enter after each.
45
12
999
1000
321
123
2300
954
1968
12
2
1999
1776
1812
1456
1
9999
3
76
200
Press a key to sort the values.
```

```
arr[0] = 1.
arr[1] = 2.
arr[2] = 3.
arr[3] = 12.
arr[4] = 12.
arr[5] = 45.
arr[6] = 76.
arr[7] = 123.
arr[8] = 200.
arr[9] = 321.
arr[10] = 954.
arr[11] = 999.
arr[12] = 1000.
arr[13] = 1456.
arr[14] = 1776.
arr[15] = 1812.
arr[16] = 1968.
arr[17] = 1999.
arr[18] = 2300.
arr[19] = 9999.
Press a key to continue.
Enter a value to search for: 1776
1776 found at arr[14]
```

Análisis El listado 19.5 incorpora todo lo que se ha dicho anteriormente acerca del ordenamiento y la búsqueda. El programa le permite dar hasta `MAX` valores (20 en este caso). Ordena los valores y los imprime en orden. Luego, le permite teclear un valor que será buscado en el arreglo. Un mensaje impreso indica el estado de la búsqueda.

La obtención de los valores para el arreglo, en las líneas 18 y 19, es código con el que usted ya está familiarizado. La línea 26 contiene la llamada a `qsort()` para lograr que se ordene el arreglo. El primer argumento es un apuntador al primer elemento del arreglo. A continuación está `MAX`, la cantidad de elementos del arreglo. Luego, se proporciona el tamaño del primer elemento, para que `qsort()` sepa la anchura de cada elemento. La llamada termina con el argumento para la función necesaria para el ordenamiento, `intcmp`.

La función `intcmp()` está definida en las líneas 51 a 54. Regresa la diferencia de los dos valores que se le pasan. A primera vista esto parece demasiado simple, pero recuerde los valores que se supone que debe regresar la función de comparación. Si los elementos son iguales, se debe regresar 0, si el elemento uno es mayor que el elemento 2, se debe regresar un número positivo y, si el elemento 1 es menor que el elemento 2, se debe regresar un número negativo. Esto es exactamente lo que hace `intcmp()`.

La búsqueda se realiza con `bsearch()`. Observe que sus argumentos son virtualmente los mismos que los de `qsort()`. La diferencia es que el primer argumento de `bsearch()` es la clave que se busca. `bsearch()` regresa un apuntador a la posición de la clave encontrada, o `NULL` si no encuentra la clave. En la línea 43 se le asigna a `ptr` el valor de retorno de `bsearch()`. `ptr` se usa en el ciclo `if` de las líneas 45 a 48 para imprimir el estado de la búsqueda.

El listado 19.6 tiene la misma funcionalidad que el listado 19.5, mas, sin embargo, el listado 19.6 ordena y busca cadenas.

Captura

Listado 19.6. Uso de `qsort()` y `bsearch()` con cadenas.

```
1: /* Uso de qsort() y bsearch() con cadenas. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: #define MAX 20
8:
9: int comp(const void *s1, const void *s2);
10:
11: main()
12: {
13:     char *data[MAX], buf[80], *ptr, *key, **key1;
14:     int count;
15:
16:     /* Recibe una lista de palabras. */
17:
18:     printf("Enter %d words, pressing Enter after each.\n",
            MAX);
19:
20:     for (count = 0; count < MAX; count++)
21:     {
22:         printf("Word %d: ", count+1);
23:         gets(buf);
24:         data[count] = malloc(strlen(buf)+1);
25:         strcpy(data[count], buf);
26:     }
27:
28:    /* Ordena las palabras (de hecho, ordena los apuntadores). */
29:
30:     qsort(data, MAX, sizeof(data[0]), comp);
31:
32:     /* Despliega las palabras ordenadas. */
33:
34:     for (count = 0; count < MAX; count++)
35:         printf("\n%d: %s", count+1, data[count]);
36:
37:     /* Recibe una clave que se ha de buscar. */
38:
39:     printf("\n\nEnter a search key: ");
40:     gets(buf);
41:
42:     /* Ejecuta la búsqueda. Primero hace que key1 sea un apuntador */
43:     /* al apuntador de la clave de búsqueda. */
44:
```

```
45:     key = buf;
46:     key1 = &key;
47:     ptr = bsearch(key1, data, MAX, sizeof(data[0]), comp);
48:
49:     if (ptr != NULL)
50:         printf("%s found.", buf);
51:     else
52:         printf("%s not found", buf);
53: }
54:
55: int comp(const void *s1, const void *s2)
56: {
57:     return (strcmp(*(char **)s1, *(char **)s2));
58: }
```

Salida

```
Enter 20 words, pressing Enter after each
Word 1: apple
Word 2: orange
Word 3: grapefruit
Word 4: peach
Word 5: plum
Word 6: pear
Word 7: cherries
Word 8: banana
Word 9: lime
Word 10: lemon
Word 11: tangerine
Word 12: star
Word 13: watermelon
Word 14: cantaloupe
Word 15: musk melon
Word 16: strawberry
Word 17: blackberry
Word 18: blueberry
Word 19: grape
Word 20: cranberry

1: apple
2: banana
3: blackberry
4: blueberry
5: cantaloupe
6: cherries
7: cranberry
8: grape
9: grapefruit
10: lemon
11: lime
12: musk melon
13: orange
```

```
14: peach
15: pear
16: plum
17: star
18: strawberry
19: tangerine
20: watermelon

Enter a search key: orange
orange found.
```

Análisis Vale la pena mencionar unos cuantos puntos acerca del listado 19.6. Este programa utiliza un arreglo de apuntadores a cadenas, una técnica que fue presentada en el Día 15, "Más sobre apuntadores". Como se vio en ese capítulo, se pueden "ordenar" las cadenas ordenando el arreglo de apuntadores. Sin embargo, este método requiere una modificación en la función de comparación. A esta función se le pasan apuntadores a los dos conceptos del arreglo que son comparados. Sin embargo, no se quiere que el arreglo de apuntadores quede ordenado con base en el valor de los apuntadores, sino basándose en los valores de las cadenas a las que apuntan.

Debido a esto, se debe usar una función de comparación a la que se le pasen apuntadores a apuntadores. Cada argumento para `comp()` es un apuntador a un elemento del arreglo y, como cada elemento es en sí mismo un apuntador (a una cadena), el argumento es, por lo tanto, un apuntador a apuntador. Dentro de la función misma se desreferencia a los apuntadores, para que el valor de retorno de `comp()` dependa de los valores de las cadenas apuntadas.

El hecho de que los argumentos pasados a `comp()` sean apuntadores a apuntadores crea otro problema. Se guarda la clave que se ha de buscar en `buf[]`, y también se sabe que el nombre de un arreglo (en este caso, `buf`) es un apuntador al arreglo. Sin embargo, se necesita pasar no a `buf` misma, sino un apuntador a `buf`. El problema es que `buf` es una constante de apuntador y no una variable de apuntador. En sí misma, `buf` no tiene dirección en memoria, sino que es un símbolo que evalúa a la dirección del arreglo. Debido a esto no se puede crear un apuntador que apunte a `buf`, con el operador de dirección de al inicio de `buf`, como en `&buf`.

¿Qué hacer? Primero, cree una variable de apuntador y asígnele el valor de `buf`. En el programa esta variable de apuntador tiene el nombre `key`. Como `key` es una variable de apuntador tiene una dirección, y se puede crear un apuntador que contenga la dirección (en este caso, `key1`). Cuando, por último, se llama a `bsearch()` el primer argumento es `key1`, un apuntador a un apuntador a la cadena de clave. La función `bsearch()` pasa ese argumento a `comp()` y todo trabaja adecuadamente.

DEBE	NO DEBE

NO DEBE Olvidar el poner el arreglo a buscar en orden ascendente antes de usar `bsearch()`.

Resumen

Este capítulo exploró más funciones útiles que se proporcionan en la biblioteca de funciones del C. Estas son funciones que ejecutan cálculos matemáticos, manejan el tiempo y le ayudan al programa con el manejo de errores. Las funciones para ordenamiento y búsqueda de datos son particularmente útiles, ya que pueden ahorrarle bastante tiempo cuando esté escribiendo sus programas.

Preguntas y respuestas

1. ¿Por qué casi todas las funciones matemáticas regresan `double`?

   La respuesta a esta pregunta es por la precisión y no por consistencia. Un `double` es más preciso que los otros tipos de variables y, por lo tanto, las respuestas son más precisas. En el Día 20, "Otras funciones", aprenderá puntos específicos sobre la especificación de tipo de variables y la promoción de variables. Estos temas también son aplicables a la precisión obtenida.

2. ¿Son `bsearch()` y `qsort()` las únicas maneras de ordenar y buscar en C?

   Se proporcionan estas dos funciones, mas, sin embargo, no tiene por qué usarlas. Muchos libros de texto de programación de computadora le enseñan la manera de escribir sus propios programas de búsqueda y ordenamiento. El C contiene todos los comandos que necesita para escribir los propios. Se pueden comprar rutinas escritas especialmente para búsqueda y ordenamiento. Los mayores beneficios de `bsearch()` y `qsort()` son que ya están escritas y que se ofrecen con cualquier compilador compatible con ANSI.

3. ¿Las funciones matemáticas validan los datos incorrectos?

   Nunca suponga que los datos tecleados son correctos. Siempre valide los datos tecleados por un usuario. Por ejemplo, si se le pasa un valor negativo a `sqrt()`, la función despliega un error. Si se está formateando la salida, probablemente no querrá que este error sea desplegado tal como es. Quite el enunciado `if` del listado 19.1 y teclee un número negativo.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado, así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Cuál es el tipo de dato que regresan todas las funciones matemáticas del C?
2. ¿A qué tipo de variable del C es equivalente `time_t`?
3. ¿Cuáles son las diferencias entre las funciones `time()` y `clock()`?
4. Cuando se llama a la función `perror()`, ¿qué hace para corregir una condición de error existente?
5. Antes de buscar en un arreglo con `bsearch()`, ¿qué se debe hacer?
6. Al usar `bsearch()`, ¿qué tantas comparaciones se requerirán para encontrar un elemento si el arreglo tiene 16,000 elementos? (Esta respuesta se da en el capítulo.)
7. Al usar `bsearch()`, ¿qué tantas comparaciones se requerirán para encontrar un elemento si el arreglo tiene 10 conceptos?
8. Al usar `bsearch()`, ¿qué tantas comparaciones se requerirán para encontrar un elemento si el arreglo tiene 2,000,000 elementos?
9. ¿Qué valores debe regresar una función de comparación para `bsearch()` y `qsort()`?
10. ¿Qué regresa `bsearch()` si no puede encontrar un elemento en un arreglo?

Ejercicios

1. Escriba una llamada a `bsearch()`. El arreglo que se ha de revisar es llamado `nombres` y los valores son caracteres. La función de comparación es llamada `comp_nom()`. Asuma que todos los nombres son del mismo tamaño.
2. BUSQUEDA DE ERRORES: ¿Qué hay de erróneo en el siguiente programa?

```
#include <stdio.h>

#include <stdlib.h>

main
```

```
{
    int values[10], count, key, *ptr;

    printf("Enter values");
    for( ctr = 0; ctr < 10; ctr++ )
        scanf( "%d", &values[ctr] );

    qsort(values, 10, función_de_comparación());
}
```

3. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en la siguiente función de comparación?

```
int intcmp( int elemento1, int elemento2)
{
    if ( elemento1 > elemento2 )
        return -1;
    else if (elemento1 < elemento2 )
        return 1;
    else
        return 0;
}
```

No se proporcionan respuestas para los siguientes ejercicios:

4. Modifique el listado 19.1 para que la función `sqrt()` trabaje con números negativos. Haga esto tomando el valor absoluto de `x`.
5. Escriba un programa que consista en un menú que ejecute varias funciones matemáticas. Use tantas funciones matemáticas como pueda.
6. Escriba una función que logre que el programa haga una pausa de aproximadamente cinco segundos, usando las funciones de tiempo aprendidas en este capítulo.
7. Añada la función `assert()` al programa del ejercicio cuatro. El programa debe imprimir un mensaje si se da un valor negativo.
8. Escriba un programa que acepte 30 nombres, y los ordene con `qsort()`. El programa debe imprimir los nombres ordenados.

9. Modifique el programa del ejercicio ocho, para que, si el usuario teclea `"FIN"`, el programa termine la aceptación de datos y ordene los valores dados.
10. Vea en el Día 15, "Más sobre apuntadores", un método de "fuerza bruta" para ordenar un arreglo de apuntadores a cadenas con base en los valores de las cadenas. Escriba un programa que mida el tiempo necesario para ordenar un arreglo grande de apuntadores con ese método, y luego compare ese tiempo con el que requiere la ejecución del mismo ordenamiento con la función de biblioteca `qsort()`.

DIA 20

Otras funciones

Este capítulo trata algunos cabos sueltos acerca de algunos temas de la programación en C que no han sido tratados en los capítulos anteriores. Usted aprenderá hoy:

- Conversiones de tipo.
- Asignación y liberación de almacenamiento en memoria.
- Argumentos de la línea de comandos.
- Operaciones sobre bits.
- Campos de bits en estructuras.

Conversiones de tipo

Todos los objetos de datos del C tienen un tipo específico. Una variable numérica puede ser un `int` o un `float`, un apuntador puede apuntar a un `double` o a un `char`, y así sucesivamente. Frecuentemente se mezclan diferentes tipos en expresiones y enunciados. ¿Qué pasa entonces? Algunas veces el C maneja automáticamente los tipos diferentes y no debe preocuparle a uno. Otras veces se deben hacer conversiones específicas de un tipo de dato a otro. Usted ya vio esto en capítulos anteriores, cuando tuvo que convertir o *especificar* una conversión explícita de tipo `void` hacia un tipo específico. La siguiente sección trata las conversiones automáticas y explícitas del C.

Conversiones automáticas de tipo

Como su nombre lo dice, las conversiones automáticas de tipo se ejecutan automáticamente por el compilador C, sin que se necesite que el programador haga algo. Sin embargo, usted debe estar consciente de lo que sucede para que pueda entender la manera en que el C evalúa expresiones.

Promoción de tipo en expresiones

Cuando es evaluada una expresión de C, el valor resultante tiene un tipo de dato. Si todos los componentes de la expresión tienen el mismo tipo, el tipo resultante es el mismo. Por ejemplo, si `x` y `y` son ambas tipo `int`, la expresión

```
x + y
```

también es de tipo `int`. ¿Qué pasa si los componentes de una expresión tienen diferente tipo? La expresión tiene el mismo tipo que el componente más complejo. De menos a más complejo, los tipos de datos numéricos son `char`, `int`, `long`, `float` y `double`. Por lo tanto, una expresión que contiene un `int` y un `char` es de tipo `int`, una expresión que contiene un `long` y un `float` es de tipo `float`, y así sucesivamente.

Dentro de las expresiones los operandos individuales son *promovidos*, según sea necesario, para que concuerden con los operandos asociados en la expresión. Los operandos son

promovidos en pares para cada operador binario de la expresión. Por supuesto que no es necesaria la promoción si ambos operandos son del mismo tipo. Si no son del mismo tipo, la promoción sigue estas reglas:

- Si cualquier operando es un `double`, el otro operando es promovido a tipo `double`.
- Si cualquier operando es un `float`, el otro operando es promovido a `float`.
- Si cualquier operando es un `long`, el otro operando es convertido a `long`.

Por ejemplo, si `x` es un `int` y `y` es un `float`, la evaluación de la expresión `x/y` hace que `x` sea promovido a tipo `float`. Esto no significa que el tipo de la variable `x` sea cambiado. Significa que se crea una copia tipo `float` de `x` y se usa en la evaluación de la expresión. El valor de la expresión es, como ya se dijo, tipo `float`.

Conversión por asignación

La promoción también sucede con el operador de asignación. Una expresión del lado derecho de un enunciado de asignación siempre es promovida al tipo del objeto de dato del lado izquierdo del operador de asignación. Tome en cuenta que esto puede causar una "degradación" en vez de una promoción. Si `f` es de tipo `float` e `i` es de tipo `int`, en el enunciado de asignación

```
f = i;
```

`i` es promovida a tipo `float`. Por el contrario, el enunciado de asignación

```
i = f;
```

hace que `f` sea degradada a tipo `int`. Su parte fraccional se pierde en la asignación a `i`. Recuerde que `f` en sí misma no es cambiada, ya que la promoción solamente afecta a una copia del valor.

Cuando un número de punto flotante es convertido a un tipo entero, se pierde la parte fraccionaria. Cuando un tipo entero es convertido a tipo de punto flotante, tal vez el valor de punto flotante resultante no sea idéntico al valor entero. Esto se debe a que el formato de punto flotante usando internamente por la computadora no puede representar con precisión todos los números enteros posibles.

Conversiones explícitas con modificadores de tipo

Una *conversión explícita de tipo* (type cast) usa al operador de conversión explícita (cast) para controlar el tipo de conversión en su programa. Una conversión explícita de tipo consiste en un nombre de tipo, entre paréntesis, antes de una expresión. La conversión explícita de tipo puede ser ejecutada en expresiones aritméticas y en apuntadores.

Conversión explícita de expresiones aritméticas

La conversión explícita de tipo de una expresión aritmética le dice al compilador que represente el valor de la expresión de determinada manera. Efectivamente, la conversión explícita de tipo es similar a una promoción, la cual fue tratada anteriormente. Sin embargo, la conversión explícita de tipo está bajo el control de usted y no del compilador. Por ejemplo, si `i` es un tipo `int`, la expresión

```
(float)i
```

especifica que `i` debe ser tipo `float`. En otras palabras, el programa hace una copia interna del valor de i en formato de punto flotante.

¿Cuándo podría usar una conversión explícita de tipo con una expresión aritmética? El uso más común es para evitar la pérdida de la parte fraccionaria de la respuesta en una división entera. El programa del listado 20.1 ilustra esto. Usted debe compilar y ejecutar el programa.

Captura

Listado 20.1. Una división entera pierde la parte fraccionaria de la respuesta.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:      int i1 = 100, i2 = 40;
6:      float f1;
7:
8:      f1 = i1/i2;
9:
10:      printf("%lf", f1);
11: }
```

Salida

```
2.000000
```

Análisis

La "respuesta" desplegada por el programa es 2.000000. Pero 100/40 da como resultado 2.5. ¿Qué pasó?

La expresión `i1/i2` de la línea 8 contiene dos variables tipo `int`. De acuerdo con las reglas especificadas anteriormente en este capítulo, el valor de la expresión es, por lo tanto, de tipo `int`. Como tal, puede representar solamente números enteros, por lo que se pierde la parte fraccionaria de la respuesta.

Tal vez piense que la asignación del resultado de `i1/i2` a una variable de tipo `float` la promueva a tipo `float`. Esto es correcto, pero su aplicación llega tarde, cuando la parte fraccionaria de la respuesta se ha perdido.

Para evitar este tipo de inexactitudes se debe dar una conversión explícita de tipo a una de las variables, de tipo `int` a tipo `float`. Si alguna de las variables ha sido especificada a tipo `float`, las reglas anteriores le dicen que la otra variable será promovida automáticamente a tipo `float` y que el valor de la expresión también será tipo `float`. Por lo tanto, la parte fraccionaria de la respuesta es preservada. Para demostrar esto cambie la línea 8 en el código fuente, para que el enunciado de asignación diga

```
f1 = (float)i1/i2;
```

Luego, el programa desplegará la respuesta correcta.

Conversión explícita de tipo para apuntadores

Ya se ha visto la conversión explícita de tipo para apuntadores. Como se vio en el Día 18, "Cómo obtener más de las funciones", un apuntador tipo `void` es un apuntador genérico, ya que puede apuntar a cualquier cosa. Antes de que pueda usar un apuntador `void`, debe darle la conversión explícita de tipo al tipo adecuado. Observe que no se necesita dar una asignación de tipo a un apuntador para asignarle un valor o para compararlo con `NULL`. Sin embargo, se le debe especificar el tipo antes de desreferenciarlo o de ejecutar aritmética de apuntadores con él. Para mayores detalles sobre la conversión explícita de tipo para apuntadores, revise el Día 18, "Cómo obtener más de las funciones".

DEBE	NO DEBE
DEBE Usar una conversión explícita de tipo para promover o degradar valores de variables.	
NO DEBE Usar una conversión explícita de tipo para impedir un mensaje de aviso del compilador. Tal vez encuentre que el uso de una conversión explícita de tipo lo libra de un mensaje de error, pero antes de quitar el mensaje de error en esta forma asegúrese de entender por qué se está obteniendo el mensaje de error.	

Asignación de espacio de almacenamiento en memoria

La biblioteca del C contiene funciones para la asignación de espacio de almacenamiento en memoria al momento de ejecución, en un proceso llamado *asignación dinámica de memoria.* Esta técnica puede tener muchas ventajas sobre la asignación explícita de memoria en el código fuente del programa (como declarar un arreglo). Con el último método se debe saber exactamente, al momento de escribir el programa, qué tanta memoria se necesita. La

asignación dinámica de memoria le permite al programa reaccionar, mientras está ejecutando, ante las demandas de memoria como la entrada de datos del usuario.

Todas las funciones para el manejo de la asignación dinámica de memoria requieren el archivo de encabezado STDLIB.H. Observe que todas las funciones de asignación regresan un apuntador tipo `void`. A este tipo de apuntador se le debe dar una conversión de tipo al tipo adecuado antes de que pueda ser usado.

La función *malloc()*

En capítulos anteriores aprendió la manera de usar la función de biblioteca `malloc()` para asignar espacio de almacenamiento para cadenas. La función `malloc()` no está limitada a cadenas, por supuesto, ya que puede asignar espacio para cualquier necesidad de almacenamiento. Esta función asigna memoria a nivel byte. Recuerde que el prototipo de `malloc()` es

```
void *malloc(size_t num);
```

El argumento `size_t` está definido en STDLIB.H como `unsigned`. La función `malloc()` asigna la cantidad `num` de bytes de espacio de almacenamiento y regresa un apuntador al primer byte. La función regresa NULL si el espacio solicitado no puede ser asignado o si `num == 0`. Revise la sección sobre `malloc()` en el Día 10, "Caracteres y cadenas", si todavía no recuerda muy bien su operación.

La función *calloc()*

La función `calloc()` también asigna memoria, pero en vez de asignar un grupo de bytes, como lo hace `malloc()`, `calloc()` asigna espacio para un grupo de objetos. El prototipo de función es:

```
void *calloc(size_t num, size_t size);
```

Recuerde que `size_t` es, en la mayoría de los compiladores, un sinónimo para `unsigned`. El argumento `num` es la cantidad de objetos que se han de asignar y `size` es el tamaño (en bytes) de cada objeto. Si la asignación es satisfactoria, toda la memoria asignada es borrada (puesta a 0) y la función regresa un apuntador al primer byte. Si la asignación falla, o si `num` o `size` son `0`, la función regresa `NULL`.

El programa del listado 20.2 ilustra el uso de `calloc()`.



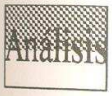
Listado 20.2. Uso de la función `calloc()` para asignar espacio de almacenamiento en memoria dinámicamente.

```
1: /* Demuestra a calloc(). */
2:
3: #include <stdlib.h>
```

```
4: #include <stdio.h>
5:
6: main()
7: {
8:     unsigned num;
9:     int *ptr;
10:
11:     printf("Enter the number of type int to allocate: ");
12:     scanf("%d", &num);
13:
14:     ptr = calloc(num, sizeof(int));
15:
16:     if (ptr != NULL)
17:         puts("Memory allocation was successful.");
18:     else
19:         puts("Memory allocation failed.");
20: }
```




```
>list2002
Enter the number of type int to allocate: 100
Memory allocation was successful.

>list2002
Enter the number of type int to allocate: 99999999
Memory allocation failed.
```



Este programa pide un valor en las línea 11 y 12. La cantidad determina qué tanto espacio es asignado. El programa trata de asignar suficiente memoria (línea 14) para guardar la cantidad especificada de variables `int`. Si la asignación falla, el valor de retorno de `calloc()` es `NULL` y, en caso contrario, es un apuntador a la memoria asignada. En el caso de este programa, el valor de retorno de `calloc()` es puesto en el apuntador a `int`, `ptr`. Un enunciado `if` en las líneas 16 a 19 revisa el estado de la asignación con base en el valor de `ptr` e imprime un mensaje pertinente.

Teclee diferentes valores y vea qué tanta memoria puede ser asignada satisfactoriamente. La máxima cantidad asignada depende, hasta cierto punto, de la configuración de su sistema. En algunos sistemas la asignación de espacio para 25,000 instancias de tipo `int` es satisfactoria, y en cambio falla para 30,000.

La función *realloc()*

La función `realloc()` cambia el tamaño de un bloque de memoria que ha sido asignado previamente con `malloc()` o `calloc()`. El prototipo de función es

```
void *realloc(void *ptr, size_t size);
```

El argumento `ptr` apunta al bloque original de memoria. El nuevo tamaño deseado, en bytes, es especificado por `size`. Los resultados posibles de `realloc()` son los siguientes:

- Si existe espacio suficiente para expandir el bloque de memoria apuntado por `ptr` es asignada la memoria adicional y la función regresa `ptr`.
- Si no existe suficiente espacio para expandir el bloque actual, se asigna un nuevo bloque del tamaño indicado por `size`, y los datos existentes son copiados del bloque anterior al inicio del nuevo bloque. El bloque anterior es liberado y la función regresa un apuntador al nuevo bloque.
- Si el argumento `ptr` es `NULL`, la función actúa en forma similar a `malloc()`, asignando un bloque de bytes del tamaño indicado por `size` y regresando un apuntador a él.
- Si el tamaño del argumento es 0, la memoria a la que apunta `ptr` es liberada y la función regresa `NULL`.
- Si la memoria es insuficiente para la reasignación (ya sea expandiendo el bloque anterior o asignando uno nuevo), la función regresa `NULL` y el bloque original no sufre cambios.

El uso de `realloc()` se muestra en el programa del listado 20.3.

Captura

Listado 20.3. Uso de `realloc()` para aumentar el tamaño de un bloque de memoria asignada dinámicamente.

```
1: /* Uso de realloc() para cambiar la asignación de memoria. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: main()
8: {
9:     char buf[80], *message;
10:
11:     /* Recibe una cadena. */
12:
13:     puts("Enter a line of text.");
14:     gets(buf);
15:
16:     /* Asigna el bloque inicial y copia la cadena a él. */
17:
18:     message = realloc(NULL, strlen(buf)+1);
19:     strcpy(message, buf);
20:
21:     /* Despliega el mensaje. */
22:
23:     puts(message);
24:
25:     /* Recibe otra cadena del usuario. */
```

```
26:
27:     puts("Enter another line of text.");
28:     gets(buf);
29:
30:     /* Aumenta la asignación y luego le concatena */
31:
32:     message = realloc(message,
        (strlen(message)+strlen(buf)+1));
33:     strcat(message, buf);
34:
35:     /* Despliega el nuevo mensaje. */
36:
37:     puts(message);
38: }
```

Salida

```
Enter a line of text.
This is the first line of text.
This is the first line of text.
Enter another line of text.
This is the second line of text.
This is the first line of text.This is the second line of text.
```

Análisis Este programa recibe una cadena en la línea 14. Esta cadena es leída hacia un arreglo de caracteres llamado `buf`. Luego, este valor es copiado a una posición de memoria apuntada por `message` (línea 19). A `message` le fue asignado espacio usando `realloc()` en la línea 18. `realloc()` fue llamada aunque no había habido una asignación anterior. Al pasarle `NULL` como primer parámetro, `realloc()` sabe que ésta es una asignación inicial.

La línea 28 obtiene una segunda cadena en el buffer `buf`. Esta cadena es concatenada con la cadena que ya se encuentra en `message`. Debido a que `message` sólo es lo suficientemente grande como para guardar la primera cadena, se necesita que sea reasignado para que haya espacio para guardar tanto la primera como la segunda cadena. Esto es exactamente lo que hace la línea 32. El programa termina imprimiendo la cadena final concatenada.

La función *free()*

Cuando se asigna memoria, ya sea con `malloc()` o `calloc()`, es tomada del espacio de memoria dinámico disponible para el programa. A este espacio algunas veces se le llama *heap* y es limitado. Cuando el programa termina de usar un bloque particular de memoria asignada, tal vez quiera "desasignar" o liberar la memoria para tenerla disponible para asignaciones futuras. Para liberar memoria que fue asignada dinámicamente use `free()`. Su prototipo es:

```
void free(void *ptr);
```

La función `free()` libera la memoria apuntada por `ptr`. Esta memoria debe haber sido asignada con `malloc()`, `calloc()` o `realloc()`. Si `ptr` es `NULL`, `free()` no hace nada. El listado 20.4 muestra la función `free()`.

Captura

Listado 20.4. Uso de `free()` para liberar memoria asignada previamente en forma dinámica.

```
1: /* Uso de free() para liberar memoria asignada dinámicamente. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: #define BLOCKSIZE 30000
8:
9: main()
10: {
11:     void *ptr1, *ptr2;
12:
13:     /* Asigna un bloque. */
14:
15:     ptr1 = malloc(BLOCKSIZE);
16:
17:     if (ptr1 != NULL)
18:         printf("\nFirst allocation of %d bytes successful.",
                BLOCKSIZE);
19:     else
20:     {
21:         printf("\nAttempt to allocate %d bytes failed.",
                BLOCKSIZE);
22:         exit(1);
23:     }
24:
25:     /* Trata de asignar otro bloque. */
26:
27:     ptr2 = malloc(BLOCKSIZE);
28:
29:     if (ptr2 != NULL)
30:     {
31:        /* Si la asignación es satisfactoria imprime un mensaje y termina. */
32:
33:         printf("\nSecond allocation of %d bytes successful.",
34:                BLOCKSIZE);
35:         exit(0);
36:     }
37:
38:     /* Si no es satisfactoria libera el primer bloque y reintenta. */
39:
40:     printf("\nSecond attempt to allocate %d bytes failed.",
               BLOCKSIZE);
```

```
41:     free(ptr1);
42:     printf("\nFreeing first block.");
43:
44:     ptr2 = malloc(BLOCKSIZE);
45:
46:     if (ptr2 != NULL)
47:         printf("\nAfter free(), allocation of %d bytes \
                successful.",
48:                BLOCKSIZE);
49: }
```

Salida

```
First allocation of 30000 bytes successful.
Second allocation of 30000 bytes successful.
```

Análisis Este programa trata de asignar dinámicamente dos bloques de memoria. Usa la constante definida `BLOCKSIZE` para determinar qué tanto debe asignar. La línea 15 hace la primera asignación usando `malloc()`. Las líneas 17 a 23 revisan el estado de la asignación, viendo si el valor de retorno fue igual a `NULL`. Se despliega un mensaje indicando el estado de la asignación. Si la asignación falla el programa termina. La línea 27 trata de asignar un segundo bloque de memoria, revisando nuevamente para ver si la asignación fue satisfactoria (líneas 29 a 36). Si la segunda asignación fue satisfactoria, una llamada a `exit()` termina el programa. Si no fue satisfactoria, un mensaje indica que falló el intento de asignar memoria. Luego, es liberado el primer bloque con `free()` (línea 41) y se hace un nuevo intento para asignar el segundo bloque.

Tal vez necesite modificar el valor de la constante simbólica `BLOCKSIZE`. En algunos sistemas, el valor de 30,000 produce la siguiente salida del programa:

```
First allocation of 30000 bytes successful.
Second attempt to allocate 30000 bytes failed.
Freeing first block.
After free(), allocation of 30000 bytes successful.
```


Uso de argumentos de la línea de comandos

El programa en C puede accesar argumentos pasados al programa en la línea de comandos. Esto se refiere a la información tecleada después del nombre del programa cuando se arrancó al programa. Si se va a arrancar el programa desde la línea de comandos del DOS, se podría teclear:

```
C>nomprog arg1 arg2 ... argn
```

Los diversos argumentos (`arg1` y siguientes) pueden ser recuperados por el programa durante la ejecución. Se puede considerar esta información como argumentos pasados a la función `main()` del programa. Los argumentos de la línea de comandos solamente pueden ser recuperados dentro de `main()`. Para hacerlo, declare a `main()` de la manera siguiente:

```
main(int argc, char *argv[])
{
   /* Aquí van los enunciados */
}
```

El primer parámetro, `argc`, es un entero que da la cantidad de argumentos de la línea de comandos disponibles. Este valor siempre es por lo menos `1`, debido a que se cuenta al nombre del programa. El parámetro `argv[]` es un arreglo de apuntadores a cadenas. Los subíndices válidos para este arreglo van desde `0` hasta argc - `1`. El apuntador `argv[0]` apunta al nombre del programa (incluida la información de ruta), `argv[1]` apunta al primer argumento que está a continuación del nombre del programa y así sucesivamente.

La línea de comandos es dividida en argumentos discretos por cualquier espacio en blanco. Si se necesita pasar un argumento que incluya un espacio se debe encerrar el argumento completo entre comillas dobles. Por ejemplo, si se teclea

```
c>nomprog arg1 "arg dos"
```

`arg1` es el primer argumento (apuntador por `argv[1]`) y `arg dos` es el segundo (apuntador por `argv[2]`). El programa del listado 20.5 muestra la manera de accesar argumentos de la línea de comandos.

Captura

Listado 20.5. Paso de argumentos de la línea de comandos a `main()`.

```
1: /* Acceso de argumentos de la línea de comandos. */
2:
3: #include <stdio.h>
4:
5: main(int argc, char *argv[])
6: {
7:    int count;
8:
9:    printf("Program name: %s\n", argv[0]);
10:
11:    if (argc > 1)
12:    {
13:        for (count = 1; count < argc; count++)
14:            printf("Argument %d: %s\n", count, argv[count]);
15:    }
16:    else
17:        puts("No command line arguments entered.");
18: }
```

Salida

```
E:\BOOK\X>list2005
Program name: E:\BOOK\X\LIST2005.EXE
No command line arguments entered.

E:\BOOK\X>list2005 first second 3 4
Program name: E:\BOOK\X\LIST2005.EXE
Argument 1: first
Argument 2: second
Argument 3: 3
Argument 4: 4
```

Análisis

Este programa no hace más que imprimir los parámetros de la línea de comandos dados por el usuario. Observe que la línea 5 usa los parámetros `argc` y `argv` mostrados anteriormente. La línea 9 imprime el primer parámetro de la línea de comandos que siempre se tiene, el nombre del programa. Observe que es `argv[0]`. La línea 11 revisa para ver si hay más de un parámetro de la línea de comandos ¿Por qué más de uno y no más de cero? Debido a que el primero es el nombre del programa. Si hay argumentos adicionales un ciclo `for` imprime cada uno de ellos en la pantalla (líneas 13 y 14). En caso contrario se imprime un mensaje pertinente (línea 17).

Los argumentos de la línea de comandos caen en dos categorías: algunos son requeridos, debido a que el programa no puede trabajar sin ellos, y en cambio otros pueden ser "indicadores" opcionales, que le dan instrucciones al programa para que se comporte de determinada forma. Por ejemplo, imagine que se tiene un programa que ordena los datos de un archivo. Si se escribe el programa para que reciba el archivo de entrada desde la línea de comandos, el nombre es información requerida. Si al usuario se le olvida dar el nombre de archivo de entrada en la línea de comandos, el programa debe manejar de alguna manera esta situación. El programa también podría ver si se encuentra el argumento `/r`, que indicaría un ordenamiento inverso. Este argumento no es requerido. El programa lo busca, y se comporta de una manera en caso de encontrarlo y de otra si no lo encuentra.

DEBE	NO DEBE
DEBE Liberar la memoria asignada cuando haya terminado de usarla.	
	NO DEBE Asumir que una llamada a `malloc()`, `calloc()` o `realloc()` ha sido satisfactoria. En otras palabras, siempre revise para ver si la memoria fue asignada.
DEBE Usar `argc` y `argv` para los nombres de variable de los argumentos de la línea de comandos para `main()`. La mayoría de los programadores del C están familiarizados con estos nombres.	
	NO DEBE Suponer que los usuarios del programa han dado los parámetros de la línea de comandos. Revise para asegurarse de que lo hayan hecho. Si no lo hicieron, despliegue un mensaje indicando la manera de usar el programa.

Operaciones sobre bits

Los operadores *a nivel bit* del C le permiten manipular bits individuales de variables enteras. Estos operadores pueden usarse solamente con los tipos enteros: `char`, `int` y `long`. Antes de continuar con esta sección debe estar familiarizado con la *notación binaria,* es decir, la manera en que la computadora almacena internamente a los enteros. Si necesita un repaso sobre la notación binaria, véase el apéndice D, "Notación binaria y hexadecimal", antes de continuar.

Los operadores a nivel bit se usan más frecuentemente cuando el programa en C interactúa directamente con el hardware del sistema, un tema que está más allá del alcance de este libro. Sin embargo, tienen otros usos, por lo que debe familiarizarse con ellos.

Los operadores de desplazamiento

Dos *operadores de desplazamiento* desplazan los bits en una variable entera en una cantidad especificada de posiciones: el operador `<<` desplaza los bits hacia la izquierda y el operador `>>` desplaza los bits hacia la derecha. La sintaxis para estos operadores binarios es:

```
x << n
x >> n
```

Cada operador desplaza los bits que se encuentran en `x` por `n` posiciones en la dirección especificada. Cuando se trata de un desplazamiento a la derecha, se ponen ceros en los `n` bits de orden superior de la variable. Cuando se trata de un desplazamiento a la izquierda, se ponen ceros en los `n` bits de orden inferior de la variable. A continuación se presentan unos cuantos ejemplos:

El binario `00001100` (`12` decimal) desplazado a la derecha por 2 evalúa al binario `00000011` (`3` decimal).

El binario `00001100` (`12` decimal) desplazado a la izquierda por 3 evalúa al binario `01100000` (`96` decimal).

El binario `00001100` (`12` decimal) desplazado a la derecha por 3 evalúa al binario `00000001` (`1` decimal).

El binario `00110000` (`48` decimal) desplazado a la izquierda por 3 evalúa al binario `10000000` (`128` decimal).

Bajo ciertas circunstancias los operadores de desplazamiento pueden usarse para multiplicar y dividir un valor por una potencia de 2. Desplazar a la izquierda un entero por `n` lugares tiene el mismo efecto que multiplicarlo por 2^n, y desplazar hacia la derecha un entero tiene el mismo efecto que dividirlo entre 2^n. El resultado de una multiplicación por desplazamiento a la izquierda mantiene la precisión solamente si no hay desbordamiento, es decir, si en el desplazamiento no se "pierden" bits de las posiciones de orden superior. Una división por desplazamiento a la derecha es una división entera, perdiéndose cualquier parte fraccionaria. El programa del listado 20.6 muestra los operadores de desplazamiento.

Captura **Listado 20.6. Uso de los operadores de desplazamiento.**

```
1:/* Demostración de los operadores de desplazamiento. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     unsigned char y, x = 255;
8:     int count;
9:
10:     printf("Decimal\t\tshift left by\tresult\n");
11:
12:     for (count = 1; count < 8; count++)
13:     {
14:          y = x << count;
15:          printf("%d\t\t%d\t\t%d\n", x, count, y);
16:     }
17: }
```

Salida

```
Decimal         shift left by   result
255               1               254
255               2               252
255               3               248
255               4               240
255               5               224
255               6               192
255               7               128
```

Los operadores lógicos a nivel de bit

Se usan tres *operadores lógicos a nivel bit* para manejar bits individuales en un tipo de dato entero. Estos operadores tienen nombres similares a los operadores lógicos CIERTO/FALSO sobre los que se aprendió en capítulos anteriores, pero sus operaciones difieren en cada uno. Los operadores lógicos a nivel bit se listan en la tabla 20.1.

Tabla 20.1. Los operadores lógicos a nivel de bit.

Operador	Acción
&	AND
|	OR inclusivo
^	OR exclusivo

Todos ellos son operadores binarios, poniendo bits en el resultado a `1` o a `0` dependiendo de los bits de sus operandos. Funcionan de la manera siguiente:

- El `AND` a nivel bit pone un bit en el resultado a 1 solamente si los bits correspondientes en ambos operandos son 1. En caso contrario, pone el bit a 0. El operador `AND` se usa para desactivar uno o más bits en un valor.
- El `OR` inclusivo a nivel bit pone un bit en el resultado a 0 solamente si los bits correspondientes en ambos operandos son 0. En caso contrario, el bit es puesto a 1. El operador `OR` se usa para encender uno o más bits en un valor.
- El `OR` exclusivo a nivel bit pone un bit en el resultado a 1 si los bits correspondientes en los operandos son diferentes (uno a 1, y el otro a 0). En caso contrario, el bit es puesto a 0.

A continuación se dan ejemplos sobre la manera en que funcionan estos operadores.

```
AND:                    11110000
                   &    01010101
                        --------
                        01010000

OR inclusivo:           11110000
                   |    01010101
                        --------
                        11110101

OR exclusivo:           11110000
                   ^    01010101
                        --------
                        10100101
```

El operador de complemento

El último operador a nivel bit es el *operador de complemento* (~). Este es un operador unario. Su acción es invertir cada uno de los bits de su operando, cambiando todos los `0` a `1` y viceversa. Por ejemplo, `~254` (binario `11111110`) evalúa a `1` (binario `00000001`).

Campos de bits en estructuras

El último tema relacionado con bits es el uso de *campos de bits* en estructuras. En el Día 11, "Estructuras", se aprendió la manera de definir las estructuras de datos, personalizándolas para que se ajusten a los datos que necesita el programa. Con campos de bits se puede lograr una personalización mayor, así como ahorrar espacio de memoria.

Un campo de bits es un miembro de estructura que contiene una cantidad especificada de bits. Se puede declarar un campo de bits para que contenga 1 bit, 2 bits, o cualquier cantidad de bits que se requiera para guardar los datos del campo. ¿Qué ventaja proporciona esto?

Supongamos que se está programando una base de datos de empleados que lleva registro de los empleados de una compañía. Muchos de los conceptos de información que guarda la base de datos son del tipo "SI" o "NO", como: "¿Está inscrito el empleado en el plan dental?" o "¿Se graduó el empleado en la universidad?". Cada parte de información SI/NO puede ser guardada en un solo bit, representando el `1` SI y el `0` NO.

Con los tipos de datos estándar del C el tipo más pequeño que se puede usar en una estructura es el tipo `char`. Claro que podría usar un miembro de estructura tipo `char` para guardar datos SI/NO, pero siete de los ocho bits del `char` serían espacio desperdiciado. Mediante el uso de campos de bits se pueden guardar ocho valores SI/NO en un solo `char`.

Los campos de bits no están limitados a los valores SI/NO. Continuando con este ejemplo de base de datos, imagine que la empresa tiene tres diferentes planes de seguros de salud. La base de datos necesita guardar datos acerca del plan, en caso de haberlo, en el que está inscrito cada empleado. Se podría representar la falta de seguro de salud por `0` y los tres planes por valores del `1` al `3`. Un campo de bits que contenga dos bits es suficiente, debido a que dos bits pueden representar valores del `0` al `3`. De manera similar, un campo de bits tres bits puede guardar valores en el rango de `0` a `7`, cuatro bits pueden guardar valores en el rango de `0` a `15`, y así sucesivamente.

Los campos de bits son nombrados y accesados en forma similar a los miembros de estructuras. Todos los campos de bits tienen tipo `unsigned int`, y el tamaño del campo (en bits) es especificado a continuación del nombre del miembro, con un signo de dos puntos y la cantidad de bits. Para definir una estructura con un miembro de 1 bit llamado `dental`, otro miembro de 1 bit llamado `universidad` y un miembro de 2 bits llamado `salud`, escriba lo siguiente:

```
struct emp_dato {
   unsigned dental       : 1;
   unsigned universidad  : 1;
   unsigned salud        : 2;
...
};
```

Los puntos suspensivos ... indican espacio para otros miembros de la estructura, que pueden ser campos de bits o campos de tipos de dato regulares. Para accesar los campos de bits use el operador de miembro de estructura, en forma similar a como lo hace con cualquier miembro de estructura. Por ejemplo, se puede expandir la definición de estructura a algo más útil.

```
struct emp_dato {
   unsigned dental       : 1;
   unsigned universidad  : 1;
   unsigned salud        : 2;
```

```
    char nombre[20];
    char apellido[20];
    char rfc[10];
};
```

Luego, declare un arreglo de estructuras:

```
struct emp_dato trabajadores[100];
```

Para asignar valores al primer elemento del arreglo se escribe algo parecido a lo siguiente:

```
trabajadores[0].dental = 1;
trabajadores[0].universidad = 0;
trabajadores[0].salud = 2;
strcpy(trabajadores[0].nombre, "Patricia");
```

El código se hace más claro, por supuesto, si se usan constantes simbólicas, como `SI` y `NO`, en vez de los valores `1` y `0`. En cualquier caso se trata cada campo de bit como un entero pequeño sin signo con la cantidad indicada de bits. El rango de valores que pueden ser asignados a un campo de bits con `n` bits es de 0 a $2^n - 1$. Si se intenta asignar un valor fuera de rango a un campo de bits, el compilador no reporta un error sino que se obtienen resultados impredecibles.

DEBE	NO DEBE
DEBE Usar constantes definidas como SI y NO, o CIERTO y FALSO, cuando trabaje con bits. Es más fácil de leer y comprender que 1 y 0.	**NO DEBE** Definir un campo de bits que ocupe 8 o 16 bits. Estos son lo mismo que otras variables disponibles, como `char` o `int`.

Resumen

Este capítulo trató diversos temas de la programación en C. Se aprendió la manera de asignar, reasignar y liberar memoria al momento de ejecución, comandos que le dan flexibilidad para la asignación de espacio de almacenamiento para los datos del programa. También vio la manera y el momento en que debe usar especificaciones de tipo con variables y apuntadores, y la manera en que el programa puede usar a `argc` y `argv[]` para accesar los argumentos de la línea de comandos del DOS.

Este capítulo también trató la manera en que un programa en C puede manejar bits individuales. Los operadores a nivel bit le permiten manejar bits individuales en variables enteras, y se pueden usar campos de bits en estructuras para maximizar la eficiencia del almacenamiento de datos.

Preguntas y respuestas

1. ¿Realmente se gana mucho con el uso de campos de bits?

   Sí, puede ganar bastante con campos de bits. Considere una circunstancia similar al ejemplo de este capítulo, donde un archivo contiene información de una investigación. A la gente se le preguntó si estaba de acuerdo o en desacuerdo con las preguntas planteadas. Suponga que se hicieron 100 preguntas a 10,000 personas. Si las respuestas se guardan en caracteres como `C` o `F`, se necesitarían 10,000 veces 100 bytes de almacenamiento (suponiendo que un carácter ocupa 1 byte). Esto es 1,000,000 de bytes de almacenamiento. Si, en vez de eso, se usan campos de bits, se necesitan 100 dividido entre 8 para cada registro, o sea 13 bytes. Esta cantidad por las 10,000 gentes son 130,000 bytes de datos. 130,000 es bastante menos que un millón.

2. ¿Por qué tendré alguna vez que liberar memoria?

   Cuando comienza a aprender a usar el C los programas no son muy grandes. Conforme crecen los programas, su uso de memoria también crece. Se debe tratar de escribir los programas para que usen la memoria de la manera más eficiente posible. Cuando se ha terminado de usar la memoria se le debe liberar. Si se escriben programas que trabajen en un ambiente de multitareas, tal vez otras aplicaciones necesiten la memoria que usted no está usando.

3. ¿Qué pasa si vuelvo a utilizar una cadena sin llamar a `realloc()`?

   No necesita llamar a `realloc()` si a la cadena que se está usando le fue asignado suficiente espacio. Llame a `realloc()` cuando la cadena actual no sea lo suficientemente grande. Recuerde que el compilador C le permite hacer casi cualquier cosa. Usted puede sobreescribir la cadena original con una cadena más grande. El problema es que también está sobreescribiendo cualquier cosa que se encuentre después de la cadena. Esto puede ser nada o pueden ser datos vitales. Si necesita la asignación de una sección de memoria mayor llame a `realloc()`.

4. ¿Es aceptable también el siguiente encabezado cuando se usa `main()` con parámetros de la línea de comandos?

   ```
   main(int argc, char **argv);
   ```

   Probablemente usted puede responder esto por sí solo. Esta declaración usa un apuntador a un apuntador de carácter, en vez de un apuntador a un arreglo de caracteres. Debido a que un arreglo es un apuntador esta definición es virtualmente la misma que la que se presentó en este capítulo. Esta declaración también se usa comúnmente. (Véase el Día 8, "Arreglos numéricos", y el Día 10 "Caracteres y cadenas" para mayores detalles.)

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿Cuál es la diferencia entre las funciones de asignación de memoria `malloc()` y `calloc()`?
2. ¿Cuál es la razón más frecuente para usar una conversión explícita de tipo con una variable numérica?
3. ¿A qué tipo de variable evalúan las siguientes expresiones?

   Supongamos que c = variable `char`, i = variable `int`, l = variable `long` y f = variable `float`.

   a. (c + i + l)

   b. (i + 32)

   c. (c + 'A')

   d. (i + 32.0)

   e. (100 + 1.0)
4. ¿Qué significa memoria *asignada dinámicamente?*
5. ¿A qué apunta argv[0]?
6. Imagine que su programa usa una estructura que debe guardar (como uno de sus miembros) el día de la semana como un valor entre 1 y 7. ¿Cuál es la manera más eficiente en memoria para hacerlo?
7. ¿Cuál es la menor cantidad de memoria en que puede ser guardada la fecha actual?

   (Consejo: día/mes/año, piense en el año con un desplazamiento a partir de 1900.)
8. ¿A qué evalúa 10010010 << 4?
9. ¿A qué evalúa 10010010 >> 4?
10. Describa la diferencia en el resultado de las dos siguientes expresiones:

    (01010101 ^ 11111111) y (~01010101)

Ejercicios

1. Escriba un comando `malloc()` que asigne memoria para 1000 elementos de tipo `long`.
2. Escriba un comando `calloc()` que asigne memoria para 1000 elementos de tipo `long`.
3. BUSQUEDA DE ERRORES: ¿Está permitida la siguiente estructura?

```
struct respuestas_cuestionario {
char nombre_estudiante[15];
unsigned respuesta1    : 1;
unsigned respuesta2    : 1;
unsigned respuesta3    : 1;
unsigned respuesta4    : 1;
unsigned respuesta5    : 1;
     }
```

4. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en el siguiente código?

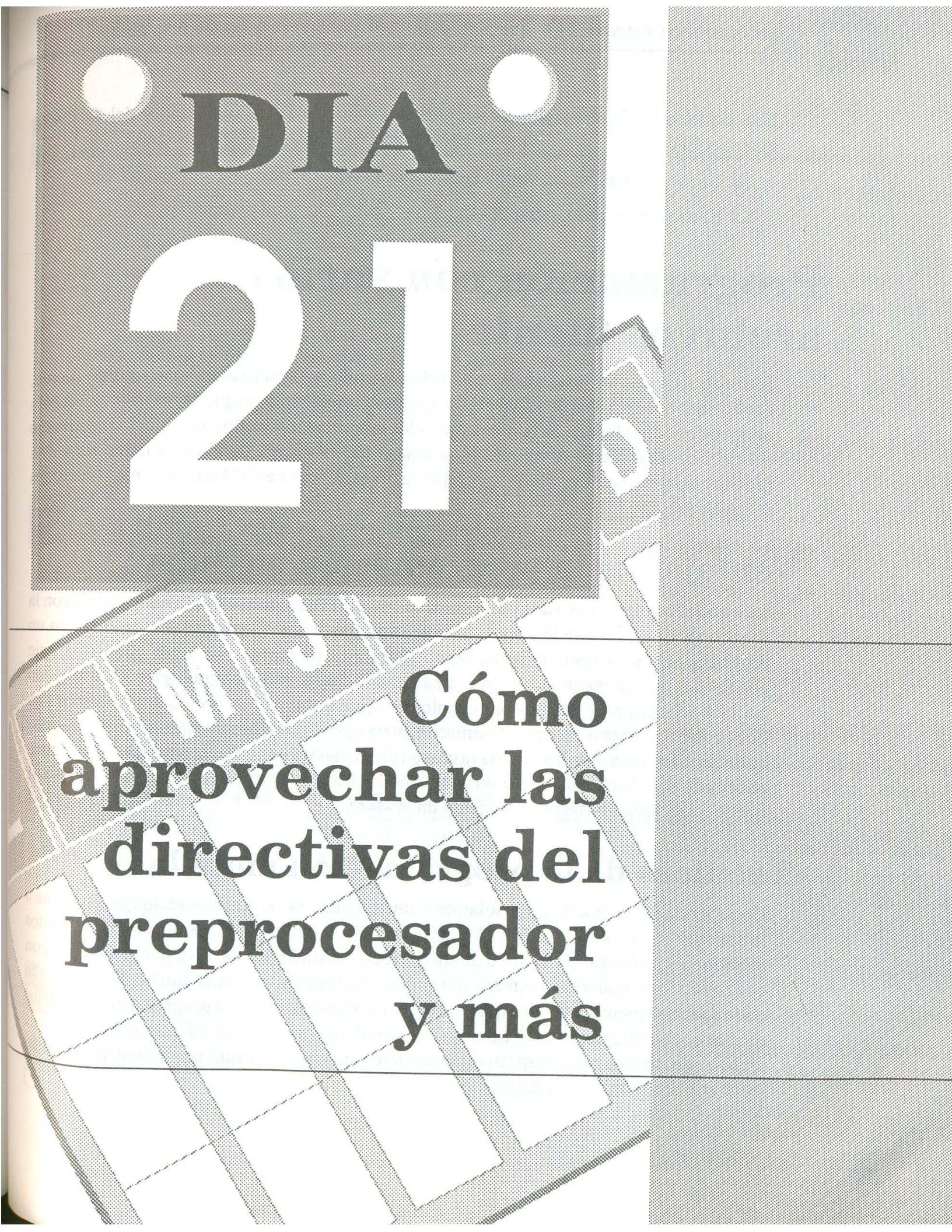
```
void func()
{
   int numero1 = 100, numero2 = 3;
   float respuesta;
   respuesta = numero1 / numero2;
   printf("%d/%d = %lf", numero1, numero2, respuesta)
}
```

Debido a las diversas soluciones posibles, no se proporcionan respuestas para los siguientes ejercicios.

5. Escriba un programa que tome dos nombres de archivo como parámetros de la línea de comandos. El programa debe copiar el primer archivo al segundo. (Véase el Día 16, "Uso de archivos de disco", en caso de necesitar ayuda para el manejo de archivos.)
6. Escriba una función que regrese la cantidad máxima de bytes disponible para ser asignada en un momento. (Consejo: la función llama tanto a `malloc()` como a `free()` varias veces.)
7. Escriba un programa que acepte dos o más valores de punto flotante como argumentos de la línea de comandos y luego despliegue su suma.
8. Escriba un programa que copie la hora de la estructura `tm` a la estructura de campo de bits descrita en la pregunta 6 del cuestionario.

9. Escriba un programa que use cada uno de los operadores lógicos a nivel bit. El programa debe aplicar el operador a nivel bit a un número y luego volverlo a aplicar al resultado. Se debe observar la salida.

10. Escriba un programa que despliegue el valor binario de un número. Por ejemplo, si se teclea 3, el programa debe desplegar 00000011. Probablemente necesite usar los operadores a nivel bit para lograr esto.



Cómo aprovechar las directivas del preprocesador y más

Este capítulo final trata algunas características adicionales del compilador del C. Usted aprenderá hoy:

- La programación con varios archivos de código fuente.
- El uso del preprocesador del C.

Programación con varios archivos fuente

Hasta ahora todos los programas en C han consistido en un solo archivo de código fuente, excluyendo los archivos de encabezado. Frecuentemente todo lo que se necesita es un solo archivo de código fuente, en particular para los programas pequeños, pero también se puede dividir el código fuente de un solo programa entre dos o más archivos, llamándose a esta práctica *programación modular.* ¿Por qué se desearía hacer esto? Las siguientes secciones lo explican.

Ventajas de la programación modular

La razón principal del uso de la programación modular está íntimamente relacionada con la programación estructurada y su dependencia en funciones. Conforme se convierta en un programador más experimentado, desarrollará más funciones con fines generales, que puede usar no solamente en el programa para el cual fueron escritas originalmente sino también en otros programas. Por ejemplo, tal vez escriba una colección de funciones con fines generales para desplegar información en la pantalla. Manteniendo estas funciones en un archivo separado puede volverlas a usar en diferentes programas que también desplieguen información en la pantalla. Cuando se escribe un programa que consiste en varios archivos fuente, cada archivo fuente es llamado un *módulo.*

Técnicas de la programación modular

Un programa en C puede tener solamente una función `main()`. El módulo que contiene a la función `main()` es llamado el *módulo principal,* y los otros módulos son llamados *módulos secundarios.* Un archivo de encabezado separado es asociado, por lo general, con cada módulo secundario (aprenderá por qué posteriormente, en este capítulo). Por ahora vea unos cuantos ejemplos simples que ilustran los puntos básicos de la programación en varios módulos. Los listados 21.1 a 21.3 muestran el módulo principal, el módulo secundario y el archivo de encabezado, respectivamente, para un programa que recibe un número del usuario y despliega su cuadrado.

Captura

Listado 21.1. SQUARE.C, el módulo principal.

```
1:/* Recibe un número y despliega su cuadrado.*/
2:
3: #include <stdio.h>
4: #include "calc.h"
5:
6: main()
7: {
8:    int x;
9:
10:    printf("Enter an integer value: ");
11:    scanf("%d", &x);
12:
13:    printf("\nThe square of %d is %ld.", x, sqr(x));
14: }
```

Captura

Listado 21.2. CALC.C, el módulo secundario.

```
1: /* Módulo que contiene las funciones de cálculo. */
2:
3: #include "calc.h"
4:
5: long sqr(int x)
6: {
7:    return ((long)x * x);
8: }
```

Captura

Listado 21.3. CALC.H, el archivo de encabezado para CALC.C.

```
1: /* CALC.H, archivo de encabezado para CALC.C. */
2:
3: long sqr(int x);
4:
5: /* fin de CALC.H */
```

Salida

```
Enter an integer value: 100
The square of 100 is 10000.
```

Análisis

Ahora veamos los componentes de estos tres archivos a mayor detalle.

- El archivo de encabezado CALC.H contiene el prototipo para la función `sqr()` que está en CALC.C. Debido a que cualquier módulo que use `sqr()` necesita saber el prototipo de `sqr()`, el módulo debe incluir a CALC.H.
- El archivo de módulo secundario, CALC.C, contiene la definición de la función `sqr()`. La directiva `#include` se usa para incluir el archivo de encabezado, CALC.H. Observe que el nombre de archivo de encabezado está encerrado entre comillas en vez de paréntesis angulares. (Aprenderá la razón de esto posteriormente, en este capítulo.)
- El módulo principal, SQUARE.C, contiene la función `main()`. Este módulo también incluye al archivo de encabezado CALC.H.

Después de que haya usado el editor para crear estos tres archivos, ¿cómo compilará y enlazará el programa ejecutable final? El compilador controla esto para usted. En la línea de comandos teclee

```
tcc square.c calc.c
```

donde `tcc` es el comando del compilador. Esto le da indicaciones a los componentes del compilador para que ejecute las siguientes tareas:

1. Compile a SQUARE.C creando SQUARE.OBJ (o SQUARE.O en un sistema UNIX). Si encuentra algún error, el compilador desplegará mensajes descriptivos del mismo.
2. Compile a CALC.C creando CALC.OBJ (o CALC.O en un sistema UNIX). Nuevamente, de ser necesario, aparecerán mensajes de error.
3. Enlacen a SQUARE.OBJ, CALC.OBJ y cualquier función necesaria de la biblioteca estándar para crear el programa ejecutable final SQUARE.EXE.

Componentes de los módulos

Como puede ver, la mecánica para compilar y enlazar un programa de varios módulos es bastante simple. La única pregunta real es lo que hay que poner en cada archivo. Los siguientes párrafos le dan algunas indicaciones generales.

El módulo secundario debe contener funciones de uso general, esto es, funciones que tal vez quiera usar en otros programas. Una práctica común es crear un módulo secundario para cada tipo de función, por ejemplo, TECLADO.C para las funciones de teclado, PANTALLA.C para las funciones de desplegado en la pantalla, etc. Para compilar y enlazar más de dos módulos liste todos los archivos fuente en la línea de comandos.

```
tcc modprin.c pantalla.c teclado.c
```

El módulo principal debe contener a `main()`, por supuesto, y cualquier otra función que sea específica para ese programa (entendiéndose por esto que no tenga uso general).

Hay, por lo general, un archivo de encabezado para cada módulo secundario. Cada archivo tiene el mismo nombre que el módulo asociado, pero con la extensión .H. En el archivo de encabezado ponga:

- Prototipos para las funciones que estén en el módulo secundario.
- Directivas `#define` para cualquier constante simbólica y macros usadas en el módulo.
- Definiciones para cualquier estructura o variable externa usada en el módulo.

Debido a que este archivo de encabezado puede ser incluido en más de un archivo fuente, tal vez quiera prevenir que partes de él se compilen más de una vez. Puede lograr esto usando las directivas del *preprocesador* para la compilación condicional (tratada posteriormente en este capítulo).

Variables externas y la programación modular

En muchos casos la única comunicación de datos entre el módulo principal y el módulo secundario es mediante los argumentos pasados a, y regresados de, la función. En este caso no necesita tomar medidas especiales en relación con la visibilidad de los datos, ¿pero qué hay acerca de las variables externas que necesitan ser visibles en ambos módulos?

Recuerde del Día 12, "Alcance de las variables", que una variable externa es aquella declarada fuera de cualquier función. Una variable externa es visible a lo largo de todo el archivo de código fuente donde es declarada. Sin embargo, no es visible automáticamente en otros módulos. Para hacerla visible se debe declarar a la variable en cada módulo, usando la palabra clave `extern`. Por ejemplo, si tiene una variable externa declarada en el módulo principal como

```
float tasa_interés;
```

puede hacer que `tasa_interés` sea visible en un módulo secundario incluyendo la siguiente declaración en ese módulo (fuera de cualquier función):

```
extern float tasa_interés;
```

La palabra clave `extern` le dice al compilador que la declaración original de `tasa_interés` (aquella que reserva espacio de almacenamiento para ella) se encuentra en cualquier otro lugar, pero que la variable debe ser visible en este módulo. Todas las variables `extern` tienen duración estática y son visibles para todas las funciones del módulo. La figura 21.1 ilustra el uso de la palabra clave `extern` en un programa de varios módulos.

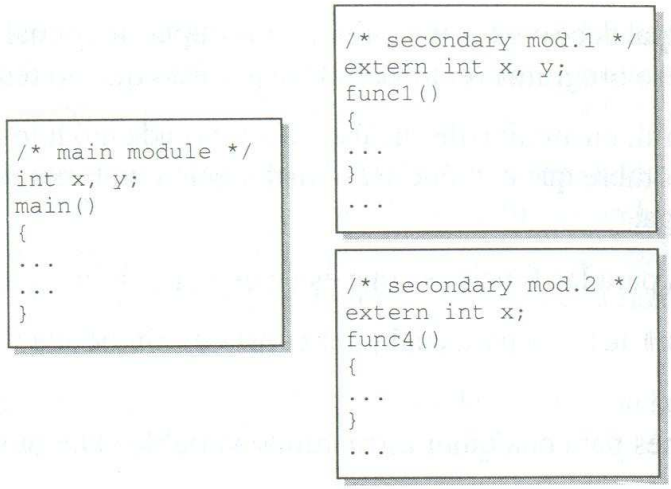


Figura 21.1. *Uso de la palabra clave* `extern` *para hacer visible una variable externa entre módulos.*

En la figura 21.1 la variable `x` es visible a lo largo de los tres módulos. Por el contrario, `y` es visible solamente en el módulo principal y en el módulo secundario 1.

Uso de archivos .OBJ

Después de escribir y depurar un módulo secundario, no necesita volverlo a compilar cada vez que lo use en un programa. Una vez que tiene el archivo objeto del código del módulo, todo lo que necesita es enlazarlo con cada programa que use las funciones del módulo.

Cuando se compila un programa, el compilador crea un archivo objeto, que tiene el mismo nombre que el archivo de código fuente de C junto con la extensión .OBJ. Digamos, por ejemplo, que se está desarrollando un módulo llamado TECLADO.C y compilándolo junto con el módulo principal BASEDATO.C con el siguiente comando:

```
tcc database.c keyboard.c
```

El archivo TECLADO.OBJ también se encuentra en el disco. Una vez que sabe que las funciones de TECLADO.C funcionan adecuadamente, puede dejar de compilarlo cada vez que recompile a BASEDATO.C (o a cualquier otro programa que lo use) y, en vez de ello, enlazar el archivo objeto existente. Para hacerlo, use este comando:

```
tcc basedato.c teclado.obj
```

Luego, el compilador compila a BASEDATO.C, y enlaza el archivo objeto resultante, BASEDATO.OBJ, con TECLADO.OBJ para crear el archivo ejecutable final, BASEDATO.EXE. Esto ahorra tiempo, debido a que el compilador no tiene que recompilar el código de TECLADO.C. Sin embargo, si modifica el código de TECLADO.C debe

recompilarlo. Además, si modifica un archivo de encabezado, debe recompilar todos los módulos que lo utilicen.

DEBE	NO DEBE

NO DEBE Tratar de compilar varios archivos fuente juntos si más de un módulo contiene una función `main()`. Solamente puede tener un `main()`.

DEBE Crear funciones genéricas en sus propios archivos fuentes. De esta manera, pueden ser enlazadas con cualquier otro programa que las necesite.

NO DEBE Usar siempre los archivos fuente de C cuando compile varios archivos juntos. Si compila un archivo fuente para tener un archivo objeto, recompílelo solamente cuando cambie el archivo. Esto ahorra gran cantidad de tiempo.

El preprocesador de C

El *preprocesador* es una parte de todos los paquetes del compilador de C. Cuando se compila un programa en C el preprocesador es el primer componente del compilador que procesa al programa. En la mayoría de los compiladores C el preprocesador es parte del programa compilador. Cuando se ejecuta al compilador, automáticamente ejecuta el preprocesador.

El preprocesador cambia el código fuente basándose en las instrucciones o *directivas del preprocesador* que se encuentren en el código fuente. La salida del preprocesador es un archivo de código fuente modificado, que luego se usa como entrada para el siguiente paso de compilación. Por lo general nunca se ve este archivo, debido a que es borrado automáticamente por el compilador después de que se usa. Sin embargo, posteriormente, en este capítulo, aprenderá la manera de ver este archivo intermedio. Primero necesita ver las directivas del preprocesador, que comienzan todas ellas con el símbolo `#`.

La directiva del preprocesador *#define*

La directiva `#define` tiene dos usos: la creación de *constantes simbólicas* y la creación de macros.

Macros de sustitución simple con *#define*

Se aprendió acerca de las macros de sustitución en el Día 3, "Variables y constantes numéricas", aunque el término usado para describirlas fue *constantes simbólicas*. Se crea una *macro de sustitución* usando `#define` para reemplazar un texto con otro texto. Por ejemplo, para reemplazar `texto1` con `texto2` se escribe:

```
#define texto1 texto2
```

La directiva causa que el compilador revise todo el archivo de código fuente reemplazando cada aparición de `texto1` con `texto2`. La única excepción sucede si `texto1` se encuentra entre comillas dobles, en cuyo caso el compilador no hace cambios.

El uso más frecuente para las macros de sustitución es la creación de constantes simbólicas, explicadas en el Día 3, "Variables y constantes numéricas". Si el programa contiene las líneas

```
#define MAX 1000

x = y * MAX;
z = MAX - 12;
```

después de preprocesar al código fuente dice:

```
x = y * 1000;
z = 1000 - 12;
```

El efecto es el mismo que si se usara la característica de búsqueda y reemplazo del editor para cambiar cada aparición de `MAX` a `1000`. Observe que `#define` no está limitado a la creación de constantes numéricas simbólicas. Se podría escribir, por ejemplo,

```
#define ZINGBOFFLE printf

ZINGBOFFLE("Hello, world.");
```

aunque hay poca razón para hacerlo. También debe saber que algunos autores se refieren a las constantes simbólicas definidas con `#define` como siendo *macros* en sí mismas. (Las constantes simbólicas son también llamadas constantes manifiestas.) Sin embargo, en este libro la palabra *macro* está reservada para el tipo de construcción que se describe a continuación.

Creación de macros de función con *#define*

También puede usar la directiva `#define` para crear macros de función. Una *macro de función* es un tipo de abreviatura, el uso de algo simple para representar algo más complejo. La razón de darle el nombre de "función" es que este tipo de macro puede aceptar argumentos, de manera similar a como lo hacen las funciones reales del C. Una ventaja de las macros de función es que sus argumentos no toman en cuenta al tipo. Por lo tanto se puede pasar cualquier tipo de variable numérica a una macro de función que espere un argumento numérico.

Vea este ejemplo. La directiva del preprocesador

```
#define MITAD(valor) ((valor)/2)
```

define a una macro llamada MITAD que toma un parámetro llamado valor. Cada vez que el preprocesador encuentra el texto MITAD(valor) en el código fuente lo reemplaza con el texto de la definición, e inserta el argumento como se necesita. Por lo tanto, la línea de código fuente:

```
resultado = MITAD(10);
```

es reemplazada por:

```
resultado = ((10)/2);
```

De manera similar, la línea de programa:

```
printf("%f", MITAD(x[1]+y[2]));
```

es reemplazada por:

```
printf("%f", ((x[1]+y[2])/2));
```

Una macro puede tener más de un parámetro, y cada parámetro puede ser usado más de una vez en el texto de reemplazo. Por ejemplo, la siguiente macro, que calcula el promedio de cinco valores, tiene cinco parámetros:

```
#define PROM5(v, w, x, y, z) (((v)+(w)+(x)+(y)+(z))/5)
```

Esta macro, donde el operador ternario determina el mayor de dos valores, también usa cada uno de sus parámetros dos veces.

```
#define MAYOR(x, y) ((x) > (y) ? (x) : (y))
```

Una macro puede tener tantos parámetros como se necesite, pero todos los parámetros de la lista deben usarse en la cadena de sustitución. Por ejemplo, la definición de macro

```
#define SUMA(x, y, z) ((x) + (y))
```

es inválida, debido a que el parámetro z no se usa en el texto de sustitución. También, cuando se llame a la macro se le deben pasar la cantidad correcta de argumentos.

Cuando se escribe una definición de macro el paréntesis izquierdo debe seguir inmediatamente al nombre de macro, sin que haya espacios en blanco. El paréntesis izquierdo le dice al preprocesador que está siendo definida una macro de función, y no una sustitución del tipo de una simple constante simbólica. Vea la siguiente definición:

```
#define SUMA (x, y, z) ((x)+(y)+(z))
```

Debido al espacio que hay entre SUMA y (, el preprocesador trata esto como una simple macro de sustitución. Cada aparición de SUMA en el código fuente es reemplazada con (x, y, z) ((x)+(y)+(z)), que obviamente no es lo que se quiere.

También observe que en la cadena de sustitución, cada parámetro está encerrado entre paréntesis. Esto es necesario para evitar efectos secundarios indeseados cuando se pasan expresiones como argumentos a la macro. Vea el siguiente ejemplo de una macro definida sin paréntesis:

```
#define CUADRADO(x) x*x
```

Si se llama a la macro con una variable simple como argumento, no hay problema. ¿Pero qué pasa si la expresión es un argumento?

```
resultado = CUADRADO(x + y);
```

La expansión resultante de la macro es:

```
resultado = x + y * x + y;
```

lo cual no da el resultado adecuado. Si se usan paréntesis se puede evitar el problema:

```
#define CUADRADO(x) (x)*(x)
```

Esta definición se expande a:

```
resultado = (x + y) * (x + y);
```

lo cual da el resultado adecuado.

El operador *encadenador* # (algunas veces llamado el operador de *cadena literal)* da flexibilidad adicional en las definiciones de macro. Cuando un parámetro de macro es precedido por # en la cadena de sustitución, el argumento es convertido en una cadena entrecomillada cuando la macro es expandida. Por lo tanto, si se define una macro como:

```
#define SALIDA(x) printf(#x)
```

y se la llama con el enunciado:

```
SALIDA(Hola Mamá);
```

se expande a:

```
printf("Hola Mamá");
```

La conversión realizada por el operador encadenador toma en cuenta los caracteres especiales. Si un carácter del argumento requiere, por lo general, un carácter de escape, el # lo precede con una diagonal invertida. Continuando con el ejemplo anterior, la llamada:

```
SALIDA("Hola Mamá");
```

se expande a:

```
printf("\"Hola Mamá\"");
```

Puede ver una demostración del operador # en el listado 21.4. Primero necesita ver otro operador usado en macros, el operador de *concatenación* ##. Este operador concatena, o une, dos cadenas en la expansión de la macro. No incluye comillas ni da tratamiento especial a los caracteres de escape. Su uso principal es crear secuencias de código fuente C. Por ejemplo, si define y llama a una macro como:

```
#define CHOP(x) func ## x
salad = CHOP(3)(q, w);
```

la macro llamada en la segunda línea es expandida a:

```
salad = func3 (q, w);
```

Puede ver que al usar el operador ## se determina cuál función es llamada. De hecho, se ha modificado el código fuente en C.

El programa del listado 21.4 muestra una forma de usar el operador #.



Listado 21.4. Uso del operador # en expansión de macros.

```
1: /* Demuestra al operador # en expansión de macro. */
2:
3: #include <stdio.h>
4:
5: #define OUT(x) printf(#x " is equal to %d.", x)
6:
7: main()
8: {
9:    int value = 123;
10:
11:    OUT(value);
12: }
```




```
value is equal to 123.
```



Con el operador #, en la línea 5, la llamada a la macro se expande con el nombre de variable value como una cadena entrecomillada pasada a la función `printf()`. Después de la expansión, en la línea 9, la macro `OUT` se ve de esta forma:

```
printf("value" " is equal to %d.", value );
```

Macros versus funciones

Ya ha visto que las macros de función pueden usarse en vez de funciones reales, por lo menos en situaciones cuando el código resultante es relativamente pequeño. Las macros de función

se pueden extender a más de una línea, pero, por lo general, llegan a ser poco prácticas más allá de unas cuantas líneas. Cuando se puede usar una función o una macro, ¿cuál debo usar? Es un compromiso entre velocidad de programa y tamaño de programa.

La definición de una macro es expandida en el código cada vez que se encuentra a la macro en el código fuente. Si el programa llama a una macro 100 veces, en el programa final va a haber 100 copias de la macro expandida. Por el contrario, el código de una función existe solamente como una sola copia. Por lo tanto, en términos de tamaño de programa, la ventaja le toca a una función real.

Cuando un programa llama a una función se requiere una cierta cantidad de sobrecarga en el procesamiento, para pasar la ejecución al código de la función y luego hacer que regrese la ejecución al programa que la llama. No hay sobrecarga de procesamiento en la "llamada" a una macro, debido a que el código está ahí mismo en el programa. En términos de velocidad una macro de función lleva ventaja.

Estas consideraciones de tamaño/velocidad no son, por lo general, de mucha importancia para el programador novato. Solamente en las aplicaciones grandes con tiempos críticos resultan importantes.

Visión de la expansión de macros

Tal vez quiera ver algunas veces cómo se ven las macros expandidas, en particular cuando no están trabajando adecuadamente. Para ver las macros expandidas se le dan instrucciones al compilador, para que cree un archivo de listado que incluya la expansión de macros después del primer paso del compilador por el código. Tal vez no pueda hacer esto con una interfaz integrada de programación (IDE). Tal vez tenga que hacerlo desde la línea de comandos. La mayoría de los compiladores tienen un indicador que debe ser puesto durante la compilación. Este indicador es pasado al compilador como un parámetro de la línea de comandos. Para precompilar un programa llamado PROGRAM.C con el compilador Zortech, se teclea:

```
ztc1 -e -1 program.c
```

En un sistema UNIX, se teclea:

```
cc -E program.c
```

Consulte el manual del compilador para determinar qué indicador necesita añadir para obtener el archivo precompilado.

El compilador ejecuta la primera pasada sobre el código fuente y produce un archivo temporal. Este archivo es un archivo de texto ASCII que contiene el código fuente preprocesado. Todos los archivos de encabezado están incluidos, las macros `#define` están expandidas y se han realizado otras directivas del preprocesador. Puede utilizar su editor para revisar este archivo y ver cómo se ven las macros expandidas.

DEBE	NO DEBE

DEBE Usar `#define` especialmente para las constantes simbólicas. Las constantes simbólicas hacen que el código sea más fácil de leer. Un ejemplo de cosas que deben ponerse en constantes definidas son los colores, CIERTO/FALSO, SI/NO, las teclas del teclado y valores máximos. Las constantes simbólicas se usan a lo largo de este libro.

NO DEBE Abusar en la utilización de las macros de función. Uselas donde sean necesarias, pero asegúrese de que sean una mejor alternativa que una función normal.

La directiva *#include*

Ya ha aprendido la manera de usar la directiva del preprocesador `#include` para incluir archivos de encabezado en el programa. Cuando encuentra una directiva `#include`, el preprocesador lee el archivo especificado y lo inserta en la posición de la directiva. ¿No se pueden usar los comodines * o ? para leer un grupo de archivos con una sola directiva `#include`. Sin embargo, se pueden anidar directivas `#include`. Esto es, un archivo incluido puede contener directivas `#include` que a su vez pueden contener directivas `#include`. La mayoría de los compiladores limitan la cantidad de niveles de profundidad del anidamiento pero, por lo general, se pueden anidar hasta 10 niveles.

Hay dos maneras de especificar el nombre de archivo para una directiva `#include`. Si el nombre de archivo está encerrado en paréntesis angulares, como `#include <stdio.h>` (como lo ha visto a lo largo de este libro), el preprocesador primero busca el archivo en el directorio estándar. Si no lo encuentra, o si no hay directorio estándar especificado, el preprocesador busca el archivo en el directorio de trabajo.

Tal vez se pregunte, "¿Qué es el directorio estándar?". En el DOS es el directorio o directorios especificados por la variable de ambiente INCLUDE del DOS. La documentación del DOS contiene información completa sobre el ambiente del DOS. Sin embargo, en pocas palabras, se pone una variable de ambiente con el comando SET (por lo general, aunque no es obligatorio, en el archivo AUTOEXEC.BAT). La mayoría de los compiladores ponen automáticamente la variable INCLUDE en el archivo AUTOEXEC.BAT cuando se instala el compilador.

El segundo método para especificar el archivo a ser incluido es encerrando el nombre de archivo entre comillas dobles: `#include "miarch.h"`. En este caso el preprocesador no busca los directorios *estándares,* sino que en vez de ello lo hace en el directorio que contiene el archivo de código fuente que está siendo compilado. En términos generales, los archivos de encabezado que usted escriba deberán guardarse en el mismo directorio que los archivos de código fuente, y son incluidos usando las comillas dobles. El directorio estándar está reservado para los archivos de encabezado proporcionados por el compilador.

Uso de *#if, #elif, #else* y *#endif*

Estas cuatro directivas del preprocesador controlan la compilación condicional. El término *compilación condicional* significa que bloques de código fuente de C son compilados solamente si se satisfacen determinadas condiciones. En muchas formas la familia `#if` de directivas del preprocesador operan en forma similar a los enunciados `if` del lenguaje C. La diferencia es que `if` controla si se ejecutan ciertos enunciados y en cambio `#if` controla el que ellos sean compilados.

La estructura de un bloque `#if` es la siguiente:

```
#if condición_1
    bloque_enunciado_1
#elif condición_2
    bloque_enunciado_2
...
#elif condición_n
    bloque_enunciado_n
#else
    bloque_enunciado_por_omisión
#endif
```

La expresión de prueba que usa `#if` puede ser casi cualquier expresión que evalúe a una constante. No se puede usar el operador `sizeof()`, especificaciones de tipo o el tipo `float`. La mayoría de las veces se usa `#if` para probar constantes simbólicas creadas con la directiva `#define`.

Cada *bloque_enunciado* consiste en uno o más enunciados del C de cualquier tipo, incluyendo directivas del preprocesador. No necesitan estar encerrados entre llaves, aunque pueden estarlo.

Las directivas `#if` y `#endif` son obligatorias, pero `#elif` y `#else` son opcionales. Se pueden tener tantas directivas `#elif` como se quiera, pero solamente una `#else`. Cuando el compilador llega a una directiva `#if`, evalúa la condición asociada. Si evalúa a `CIERTO` (diferente de cero), son compilados los enunciados que están a continuación de `#if`. Si evalúa a `FALSO` (cero), el compilador evalúa, en orden, las condiciones asociadas con cada directiva `#elif`. Son compilados los enunciados asociados con el primer `#elif CIERTO`. Si ninguna de las condiciones evalúa a `CIERTO`, son compilados los enunciados que están a continuación de la directiva `#else`.

Observe que, a lo mucho, se compila un solo bloque de enunciados dentro de la construcción `#if...#endif`. Si el compilador no encuentra directiva `#else` tal vez no compile ningún enunciado.

Los usos posibles para estas directivas de compilación condicional están limitados solamente por su imaginación. Este es un ejemplo. Supongamos que se está escribiendo un programa que usa gran cantidad de información específica del país. Esta información está contenida en un archivo de encabezado para cada país. Cuando se compila el programa para ser usado en diferentes países, se puede usar una construcción `#if...#endif` de la manera siguiente:

```
#if INGLATERRA == 1
    #include "inglat.h"
#elif FRANCIA == 1
    #include "francia.h"
#elif ITALIA == 1
    #include "italia.h"
#else
    #include "usa.h"
#endif
```

Luego, usando `#define` para definir la constante simbólica adecuada, se puede controlar cuál archivo de encabezado se incluye durante la compilación.

Uso de *#if...#endif* para ayudarse en la depuración

Otro uso común para `#if...#endif` es la inclusión de código condicional para la depuración del programa. Se puede definir una constante simbólica `DEPURA`, puesta a `1` o a `0`. A lo largo del programa se puede insertar código para depuración de la manera siguiente:

```
#if DEPURA == 1
   aquí va código para depuración
#endif
```

Durante el desarrollo del programa, si se define a `DEPURA` como `1`, se incluye el código de depuración para que ayude a localizar cualquier error. Una vez que el programa está funcionando adecuadamente se puede redefinir a `DEPURA` como `0` y volver a compilar el programa sin el código de depuración.

El operador `defined()` es útil cuando se escriben directivas de compilación condicional. Este operador revisa para ver si está definido un nombre particular. Por lo tanto, la expresión

```
defined( NOMBRE )
```

evalúa a `CIERTO` o `FALSO` dependiendo de si *NOMBRE* se encuentra definido o no. Con `defined()` se puede controlar la compilación con base en definiciones anteriores, sin tomar en cuenta el valor específico de un nombre. Refiriéndonos al ejemplo anterior del código de depuración, se podría reescribir la sección `#if...#endif` de la manera siguiente:

```
#if defined( DEPURA )
     aquí va código para depuración
#endif
```

También puede usar a `defined()` para asignar una definición a un nombre solamente si no ha sido definido con anterioridad. Use el operador `NOT`, `!`, de la manera siguiente:

```
#if !defined( CIERTO )     /* Si CIERTO no está definido. */
   #define CIERTO 1
#endif
```

Observe que el operador `defined()` no requiere que un nombre esté definido a nada en particular. Por ejemplo, después de la línea de programa

```
#define ROJO
```

se encuentra definido el nombre `ROJO`, pero sin *ser* nada. Aunque esto sea así, la expresión defined (`ROJO`) todavía evalúa a `CIERTO`. Por supuesto que las apariciones de `ROJO` en el código fuente son quitadas y no son reemplazadas con nada, por lo que debe usarlo con precaución.

Cómo evitar la inclusión múltiple de archivos de encabezado

Conforme crece el programa, o conforme se usen más frecuentemente los archivos de encabezado, se corre el riesgo de incluir accidentalmente un archivo de encabezado más de una vez. Esto puede hacer que el compilador caiga en confusiones. Usando las directivas que se han aprendido se puede evitar fácilmente este problema. Vea el ejemplo del listado 21.5.

Captura

Listado 21.5. Uso de las directivas del preprocesador con archivos de encabezado.

```
1: /* PROG.H. Archivo de encabezado con revisión para prevenir inclusiones
         múltiples. */
2:
3. #if defined( PROG_H )
4: /* El archivo ya ha sido incluido. */
5: #else
6: #define PROG_H
7:
8: /* Aquí va la información del archivo de encabezado. */
9:
10:
11:
12: #endif
```

Análisis

Examine lo que hace este archivo de encabezado. En la línea 3 revisa si `PROG_H` está definido. Observe que `PROG_H` es similar al nombre del archivo de encabezado. Si `PROG_H` está definido, se incluye un comentario en la línea 4 y el programa busca a `#endif` al final del archivo de encabezado. Esto significa que no se hace nada más.

¿Cómo se logra definir a `PROG_H`? Es definido en la línea 6. La primera vez que es incluido este encabezado el preprocesador revisa para ver si está definido `PROG_H`. No lo estará, por

lo que el control pasa al enunciado `#else`. La primera cosa que se hace después de `#else` es definir a `PROG_H`, por lo que cualquier otra inclusión de este archivo se brinca el cuerpo del archivo. Las líneas 7 a 11 pueden contener cualquier cantidad de comandos o declaraciones.

La directiva *#undef*

La directiva `#undef` es el opuesto de `#define`, ya que quita la definición de un nombre. Este es un ejemplo:

```
#define DEPURA 1

/* En esta sección del programa las apariciones de DEPURA */
/* son reemplazadas con 1 y la expresión defined( DEPURA ) */
/* evalúa a CIERTO.*/

#undef DEPURA

/* En esta sección del programa las apariciones de DEPURA */
/* no son reemplazadas y la expresión defined( DEPURA ) */
/* evalúa a FALSO.*/
```

Puede usar `#undef` y `#define` para crear un nombre que sea definido solamente en partes del código fuente. Puede usar esto en combinación con la directiva `#if`, como se explicó anteriormente, para un mayor control sobre las compilaciones condicionales.

Macros predefinidas

La mayoría de los compiladores tienen una cantidad de macros predefinidas. Las más útiles de éstas son `__DATE__`, `__TIME__`, `__LINE__` y `__FILE__`. Observe que cada una de éstas está precedida y seguida de dobles subrayados. Esto se hace para prevenir que se las redefina.

Estas macros funcionan de manera similar a las macros descritas anteriormente, en este capítulo. Cuando el precompilador llega a alguna de estas macros, reemplaza la macro con el código de la macro. `__DATE__` y `__TIME__` son reemplazadas con la fecha y la hora de trabajo. Esto es, la fecha y la hora en que el archivo fuente es precompilado, lo cual es información útil para controlar versiones.

Las otras dos macros son todavía más valiosas. `__LINE__` es reemplazada por el número de línea de trabajo del archivo fuente. `__FILE__` es reemplazado con el nombre de archivo fuente de trabajo. Estas dos macros son utilizadas mejor cuando se trata de depurar un programa o manejar los errores. Considere el siguiente enunciado `printf()`:

```
31:
32: printf( "Programa %s: (%d) Error al abrir archivo
            __FILE__, __LINE__);
33:
```

Si estas líneas fueran parte de un programa llamado MIPROG.C, se imprimiría:

```
Programa MIPROG.C: (32) Error al abrir archivo
```

Tal vez esto no le parezca importante en este momento, pero conforme crecen los programas y se distribuyen entre varios archivos fuente, encontrar errores llega a ser cada vez más difícil. Al usar `__LINE__` y `__FILE__` se facilita la depuración.

DEBE	NO DEBE

DEBE Usar las macros `__LINE__` y `__FILE__` para lograr que los mensajes de error sean más útiles.

NO DEBE Olvidar `#endif` cuando use el enunciado `#if`.

DEBE Poner paréntesis alrededor del valor que ha de ser pasado a una macro. Esto previene errores. Por ejemplo:

```
#define CUBO(x)   (x)*(x)*(x)
```

en vez de:

```
#define CUBO(x)   x*x*x
```

Resumen

Este capítulo trató algunas de las herramientas de programación más avanzadas disponibles en los compiladores C. Aprendió a escribir programas que tienen el código fuente dividido en varios archivos o módulos. Esta práctica, llamada programación modular, facilita la reutilización de funciones de fines generales en más de un programa. Ya vio la manera en que puede usar las directivas del preprocesador para crear macros de función, para la compilación condicional y para otras tareas. Por último, vio que el compilador le proporciona algunas macros de función.

Preguntas y respuestas

1. Cuando se compilan varios archivos, ¿cómo sabe el compilador cuál nombre de archivo debe usar para el archivo ejecutable?

   Tal vez piense que el compilador usa el nombre de archivo que contiene la función `main()`. Sin embargo, no es el caso general. Cuando se compila desde la línea de comandos, el primer archivo listado se usa para determinar el nombre. Por ejemplo, si se compiló lo siguiente con el Turbo C de Borland, el archivo ejecutable será llamado ARCH1.EXE.

   ```
   tcc arch1.c main.c prog.c
   ```

2. ¿Necesitan tener los archivos de encabezado la extensión .H?

   No. Se le puede dar al archivo de encabezado cualquier nombre que se desee. Es práctica estándar usar la extensión .H.

3. Cuando se incluyen archivos de encabezado, ¿se puede usar una ruta explícita?

   Sí. Si se quiere especificar la ruta donde se encuentra un archivo que ha de ser incluido, sí se puede. En este caso se pone entre comillas el nombre del archivo que se ha de incluir.

4. ¿Han sido presentadas todas las macros predefinidas y las directivas del preprocesador?

   No. Las macros predefinidas y directivas presentadas en este capítulo son las más comunes en la mayoría de los compiladores. Sin embargo, la mayoría de los compiladores también tienen macros y constantes adicionales.

Taller

El taller proporciona un cuestionario que le ayudará a reafirmar su comprensión del material tratado así como ejercicios para darle experiencia en el uso de lo aprendido.

Cuestionario

1. ¿A qué se refiere el término *programación modular?*
2. En la programación modular, ¿qué es el módulo principal?
3. Cuando se define una macro, ¿por qué debe encerrarse cada argumento entre paréntesis?

4. ¿Cuáles son los pros y los contras del uso de una macro en vez de una función regular?
5. ¿Qué hace el operador `defined()`?
6. ¿Qué debe ser usado siempre que se usa `#if`?
7. ¿Qué extensión tienen los archivos ya compilados? (Suponga que no han sido enlazados.)
8. ¿Qué es lo que hace `#include`?
9. ¿Cuál es la diferencia entre

```
#include <miarch.h>
```

y

```
#include "miarch.h"?
```

10. ¿Para qué se usa __DATE__?

Ejercicios

Debido a que hay muchas soluciones posibles, no se proporcionan respuestas para los siguientes ejercicios.

1. Use su compilador para compilar varios archivos fuente en un solo archivo ejecutable. (Puede usar los listados 21.1 a 21.3 o sus propios listados.)
2. Escriba una rutina de error que reciba un número de error, número de línea y nombre de módulo. La rutina debe imprimir un mensaje de error formateado y luego terminar el programa. Use las macros predefinidas para el número de línea y el nombre de módulo (pase el número de línea y el nombre de módulo del lugar donde sucede el error). Un ejemplo posible para un error formateado pudiera ser:

```
módulo.c (Línea ##): Número de error ##
```

3. Modifique el ejercicio anterior para hacer más descriptivo el error. Con su editor cree un archivo de texto que contenga un número de error y un mensaje. Llame a este archivo ERRORES.TXT. Puede contener información como la siguiente:

```
1    Error número 1
2    Error número 2
90   Error en la apertura de archivo
100  Error en la lectura de archivo
```

Haga que la rutina de error busque en este archivo y despliegue el mensaje de error adecuado con base en el número que se le pase.

4. Algunos archivos de encabezado pueden ser incluidos más de una vez cuando se está escribiendo un programa modular. Use directivas del preprocesador para escribir el esqueleto de un archivo de encabezado que compile solamente la primera vez que es encontrado durante la compilación.

5. Este es el último ejercicio del libro y la determinación de su contenido le toca a usted. Seleccione una tarea de programación que le interese y que también satisfaga una necesidad real que tenga. Por ejemplo, podría escribir programas para hacer el catálogo de su colección de discos, llevar el registro de su chequera o hacer cálculos financieros relacionados con la compra de la casa que desea. No hay mejor cosa que enfrentar un problema real de programación para afinar sus habilidades de programador y ayudarle a recordar todas las cosas que aprendió en este libro.

3

Revisión de la semana 3

Ya ha terminado su tercera y última semana de aprendizaje sobre la manera de programar en C. Comenzó la semana tratando temas avanzados, tales como apuntadores y archivos de disco. A mitad de la semana vio unas cuantas de las muchas funciones contenidas en la mayoría de las bibliotecas de funciones de los compiladores de C. Terminó la semana descubriendo detalles adicionales, necesarios para obtener el máximo del compilador y del lenguaje C. El siguiente programa debe reunir muchos de estos temas.

Nota: Los números a la izquierda de los números de línea indican el capítulo donde se tratan los conceptos presentados en esa línea. Si no comprende bien la línea, vea el capítulo de referencia para mayor información.

Listado R3.1. Listado de revisión de la semana tres.

```
1: /* Nombre del programa: week3.c                                    */
2: /* Programa para dar nombres y números de teléfono.               */
3: /* Esta información es escrita a un archivo de disco              */
4: /* especificado con un parámetro de la línea de comandos.         */
5:
6: #include <conio.h>
7: #include <stdlib.h>
8: #include <stdio.h>
9: #include <time.h>
10: #include <string.h>
11:
12: /*** constantes definidas ***/
13: #define YES   1
14: #define NO    0
15: #define REC_LENGTH  54
16:
17:/*** variables ***/
18:
19: struct record {
20:    char fname[15+1];              /* nombre + NULL                */
21:    char lname[20+1];              /* apellido paterno + NULL      */
22:    char mname[10+1];              /* apellido materno + NULL      */
23:    char phone[9+1];               /* número de teléfono + NULL    */
24: } rec;
25:
26: /* prototipos de función */
27:
28: int  main(int argc, char *argv[]);
29: void display_usage(char *filename);
30: int  display_menu(void);
31: void get_data(FILE *fp, char *progname, char *filename);
32: void display_report(FILE *fp);
33: int  continue_function(void);
34: int  look_up( FILE *fp );
35:
36: /* inicio del programa *
37:  *----------*/
38:
39: int main(int argc, char *argv[])
40: {
```

CH19 CH18 CH20 CH20

```
CH20    41:     FILE *fp;
        42:     int  cont = YES;
        43:     int  ch;
        44:
CH20    45:     if( argc < 2 )
        46:     {
        47:        display_usage(argv[0]);
        48:        exit(1);
        49:     }
        50:
CH16    51:     if ((fp = fopen( argv[1], "a+")) == NULL)  /* abre el archivo */
        52:     {
        53:         fprintf( stderr, "%s(%d)--Error opening file %s",
        54:                                    argv[0],__LINE__, argv[1]);
        55:         exit(1);
        56:     }
        57:
        58:     while( cont == YES )
        59:     {
        60:        switch( display_menu() )
        61:        {
CH18    62:          case '1': get_data(fp, argv[0], argv[1]); /* Día 18 */
        63:                    break;
        64:           case '2': display_report(fp);
        65:                    break;
        66:          case '3': look_up(fp);
        67:                    break;
        68:          case '4': printf("\n\nThank you for using this \
                                    program!");
        69:                    cont = NO;
        70:                    break;
        71:          default:  printf("\n\nInvalid choice, Please select \
                                  1 to 4!");
        72:                    break;
        73:        }
        74:     }
        75:
CH16    76:     fclose(fp);       /* cierra el archivo */
        77:     return(0);
        78: }
        79:
```

Listado R3.1. continuación

```
80: /*-------------------------------------------------*
81:  *   display_menu()                                 *
82:  *-------------------------------------------------*/
83:
84: int display_menu(void)
85: {
86:     int ch;
87:
88:     printf( "\n");
89:     printf( "\n     MENU");
90:     printf( "\n   ========\n");
91:     printf( "\n1.  Enter names");
92:     printf( "\n2.  Print report");
93:     printf( "\n3.  Look up number");
94:     printf( "\n4.  Quit");
95:     printf( "\n\nEnter Selection ==> ");
96:
97:     return(getch());
98: }
99:
100: /*-------------------------------------------------*
101:  * Función: get_data() *
102:  *-------------------------------------------------*/
103:
104: void get_data(FILE *fp, char *progname, char *filename)
105: {
106:    int cont = YES;
107:
108:    while( cont == YES )
109:    {
110:       printf("\n\nPlease enter information:" );
111:
112:       printf("\n\nEnter first name: ");
113:       gets(rec.fname);
114:
115:       printf("\nEnter middle name: ");
116:       gets(rec.mname);
117:
118:       printf("\nEnter last name: ");
```

```
119:       gets(rec.lname);
120:
121:       printf("\nEnter phone in 123-4567 format: ");
122:       gets(rec.phone);
CH16 123:
124:       if (fseek( fp, 0, SEEK_END ) == 0)
125:          if( fwrite(&rec, 1, sizeof(rec), fp) != sizeof(rec))
126:          {
CH21 127:           fprintf( stderr, "%s(%d)--Error writing to file %s",
128:                                progname,_ _LINE_ _, filename);
129:              exit(2);
130:          }
131:       cont = continue_function();
132:    }
133: }
134: /*-------------------------------------------------*
135:  * Función:  display_report()                      *
136:  * Objetivo: Imprimir los nombres formateados de   *
137:  *           las personas en el archivo           *
138:  *-------------------------------------------------*/
139:
140: void display_report(FILE *fp)
141: {
142:     time_t rtime;
143:     int num_of_recs = 0;
144:
145:     time(&rtime);
146:
CH19 147:     fprintf(stdout, "\n\nRun Time: %s", ctime( &rtime));
148:     fprintf(stdout, "\nPhone number report\n");
CH16 149:
150:     if(fseek( fp, 0, SEEK_SET ) == 0)
151:     {
152:        fread(&rec, 1, sizeof(rec), fp);
153:        while(!feof(fp))
154:        {
155:           fprintf(stdout,"\n\t%s, %s %c %s", rec.lname,
156:                                  rec.fname, rec.mname[0],
                                       rec.phone);
157:           num_of_recs++;
158:           fread(&rec, 1, sizeof(rec), fp);
```

Listado R3.1. continuación

```
159:        }
160:        fprintf(stdout, "\n\nTotal number of records: %d",
                    num_of_recs);
161:        fprintf(stdout, "\n\n* * * End of Report * * *");
162:     }
163:     else
164:        fprintf( stderr, "\n\n*** ERROR WITH REPORT ***\n");
165: }
166: /*-------------------------------------------------*
167:  * Función: continue_function()                    *
168:  *-------------------------------------------------*/
169:
170: int continue_function( void )
171: {
172:     char ch;
173:
174:     do
175:     {
176:       printf("\n\nDo you wish to enter another? (Y)es/(N)o ");
177:         ch = getch();
178:     } while( strchr( "NnYy", ch) == NULL );
179:
180:     if(ch == 'n' || ch == 'N')
181:         return(NO);
182:     else
183:         return(YES);
184: }
185:
186: /*-------------------------------------------------*
187:  * Función: display_usage()                        *
188:  *-------------------------------------------------*/
189:
190: void display_usage( char *filename )
191: {
192:     printf("\n\nUSAGE: %s filename", filename );
193:     printf("\n\n       where filename is a file to store \
                people\'s names");
194:     printf("\n       and phone numbers.\n\n");
195: }
196:
```

CH17

```
197: /*-------------------------------------------------*
198:  * Función: look_up()                              *
199:  * Regresa: cantidad de nombres encontrados        *
200:  *-------------------------------------------------*/
201:
202: int look_up( FILE *fp )
203: {
204:     char tmp_lname[20+1];
205:     int  ctr = 0;
206:
207:    fprintf(stdout, "\n\nPlease enter last name to be found: ");
208:     gets(tmp_lname);
209:
210:     if( strlen(tmp_lname) != 0 )
CH17 211:     {
212:        if (fseek( fp, 0, SEEK_SET ) == 0)
CH16 213:        {
214:           fread(&rec, 1, sizeof(rec), fp);
215:           while( !feof(fp))
216:           {
217:              if( strcmp(rec.lname, tmp_lname) == 0 )
CH17                /* sí concuerda */
218:              {
219:                 fprintf(stdout, "\n%s %s %s - %s", rec.fname,
                                                        rec.mname,
220:                                                    rec.lname,
                                                        rec.phone);
221:                 ctr++;
222:              }
223:              fread(&rec, 1, sizeof(rec), fp);
224:           }
225:        }
226:        fprintf( stdout, "\n\n%d names matched.", ctr );
227:     }
228:     else
229:     {
230:        fprintf( stdout, "\nNo name entered." );
231:     }
232:     return(ctr);
233: }
```

Tal vez considere que éste es un programa bastante largo, mas, sin embargo, apenas hace lo que se necesita que haga. Este programa es similar a los programas presentados en las revisiones de las semanas 1 y 2. Se han quitado unos cuantos de los conceptos de datos que se vieron en los programas de revisión de la semana 2. Este programa le permite al usuario dar información sobre gente. La información que ha de ser registrada es el nombre y apellidos y el número de teléfono. La principal diferencia que observará en este programa es que no hay límite a la cantidad de gente cuyos datos pueden ser tecleados en el programa. Esto se debe a que se usa un archivo de disco.

Este programa le permite al usuario especificar el nombre de un archivo que ha de ser usado con el programa. `main()` se inicia en la línea 39 con los argumentos requeridos, `argc` y `argv`, para obtener los parámetros de la línea de comandos. Se vio esto en el Día 20, "Otras funciones". La línea 45 revisa el valor de `argc` para ver qué tantos parámetros se dieron en la línea de comandos. Si `argc` es menor que 2, sólo fue dado un parámetro (el comando para ejecutar el programa). Debido a que no se dio un nombre de archivo, se llama a `display_usage()` con `argv[0]` como argumento. `argv[0]`, el primer parámetro dado en la línea de comandos, es el nombre del programa.

La función `display_usage()` está en las líneas 190 a 195. Cada vez que escriba un programa que toma argumentos de la línea de comandos es buena idea incluir una función similar a `display_usage()`, que muestra la manera de usar el programa. ¿Por qué no escribe la función simplemente el nombre del programa en vez de usar la variable `filename`? La respuesta es simple. Al usar una variable no hay que preocuparse si el usuario cambió el nombre del programa, ya que la descripción usada siempre es la adecuada.

Tenga presente que la mayoría de los conceptos nuevos en este programa vienen del Día 16, "Uso de archivos de disco". La línea 41 declara un apuntador a archivo, llamado `fp`, que es usado a lo largo del programa para accesar el archivo que fue proporcionado en la línea de comandos. La línea 51 trata de abrir este archivo con un modo de `"a+"` (`argv[1]` es el segundo elemento listado en la línea de comandos). El modo `"a+"` es usado debido a que se quiere tener la capacidad de añadir datos al archivo y leer cualquier registro existente. Si la apertura falla, las líneas 53 y 54 despliegan un mensaje de error, antes de que la línea 55 dé por terminado el programa. Observe que el mensaje de error contiene información descriptiva. También observe que `__LINE__`, tratada en el Día 21, "Cómo aprovechar las directivas del preprocesador y más", indica el número de línea donde sucedió el error. Una vez que el archivo ha sido abierto se presenta un menú. Cuando el usuario selecciona la opción para salir del programa, en la línea 76 se cierra el archivo con `fclose()`, antes de que el programa regrese el control al sistema operativo.

En la función `get_data()` hay unos cuantos cambios significativos. La línea 104 contiene el encabezado de función. La función ahora acepta tres apuntadores. El primer apuntador es el más importante, ya que es la manija para el archivo al que se le va a escribir. Las líneas 108 a 132 contienen un ciclo `while`, que continúa obteniendo datos hasta que el usuario decide terminar. Las líneas 110 a 122 piden los datos en la misma forma que el programa de revisión de la semana 2. La línea 124 llama a `fseek()`, para ajustar el apuntador del archivo de disco al final y así escribir la nueva información. Observe que el programa no hace nada si falla el posicionamiento. Un buen programa debe manejar fallas de este tipo. La línea 125 escribe los datos al archivo de disco con una llamada a `fwrite()`.

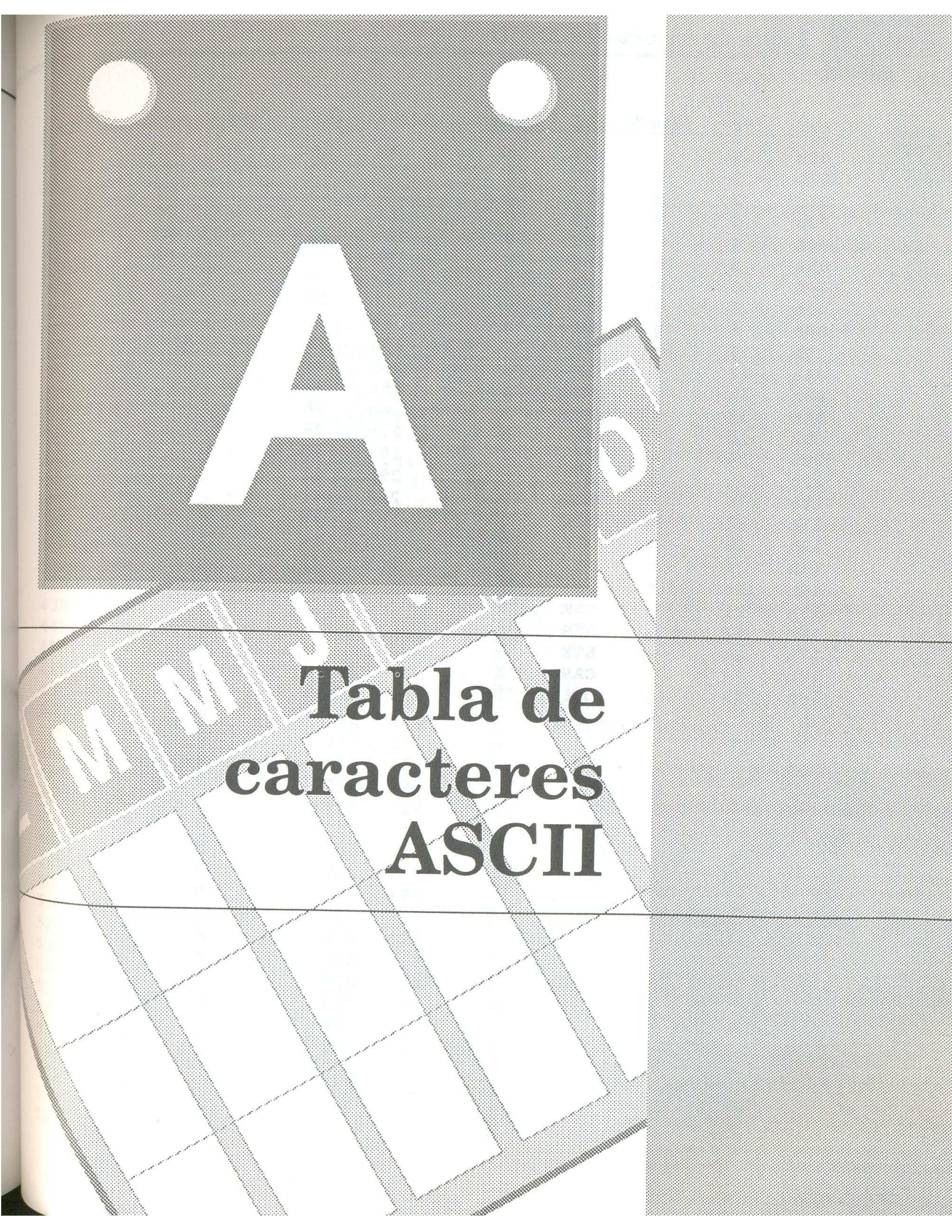
El informe presentado en este programa también ha cambiado. Un cambio, típico de la mayoría de los informes del "mundo real", es la adición de la fecha y la hora de trabajo en la parte superior del informe. En la línea 142 es declarada la variable `rtime`. Esta variable es pasada a `time()`, y luego desplegada usando la función `ctime()`. Estas funciones de tiempo fueron presentadas en el Día 19, "Exploración de la biblioteca de funciones".

Antes de que el programa pueda iniciar la impresión de registros en el archivo, necesita reposicionar el apuntador de archivo al inicio del archivo. Esto se logra en la línea 150 con otra llamada a `fseek()`. Una vez que el apuntador de archivo está posicionado se pueden leer los registros. La línea 152 hace la primera lectura. Si la lectura es satisfactoria, el programa inicia un ciclo `while`, que continúa hasta que se llega el final del archivo (cuando `feof()` regresa un valor diferente de cero). Si no se ha llegado al final del archivo, las líneas 155 y 156 imprimen información, la línea 157 cuenta el registro y la línea 158 trata de leer el siguiente registro. Debe observar que las funciones son usadas sin revisar su valor de retorno. Para proteger al programa contra errores, las llamadas de función deben contener revisiones para asegurarse de que no hay errores.

La función `continue_function()` contiene una pequeña modificación. La línea 178 ha sido cambiada para usar la función `strchr()`. Esta función hace que la línea de código sea más fácil de entender.

La última función del programa es nueva. Las líneas 202 a 233 contienen la función `look_up()`, que busca en el archivo de disco todos los registros que tengan un apellido dado. Las líneas 207 y 208 piden el nombre que se ha de encontrar y lo guardan en una variable local, llamada `tmp_lname`. Si `tmp_lname` no está en blanco (línea 210) el apuntador del archivo es puesto al principio del archivo. Luego, es leído cada registro. Con `strcmp()` (línea 217) es comparado el apellido del registro contra `tmp_lname`. Si los nombres concuerdan, se imprime el registro (líneas 219 y 220). Esto continúa hasta que se llega al final del archivo. Nuevamente debe observar que no se revisa el valor de retorno de todas las funciones. Usted siempre debe revisar los valores de retorno.

Usted debe ser capaz de hacer modificaciones a este programa para crear sus propios archivos que puedan guardar cualquier información. Con las funciones que aprendió en la tercera semana, junto con las otras funciones que tiene su biblioteca, debe ser capaz de crear casi cualquier programa que desee.



A

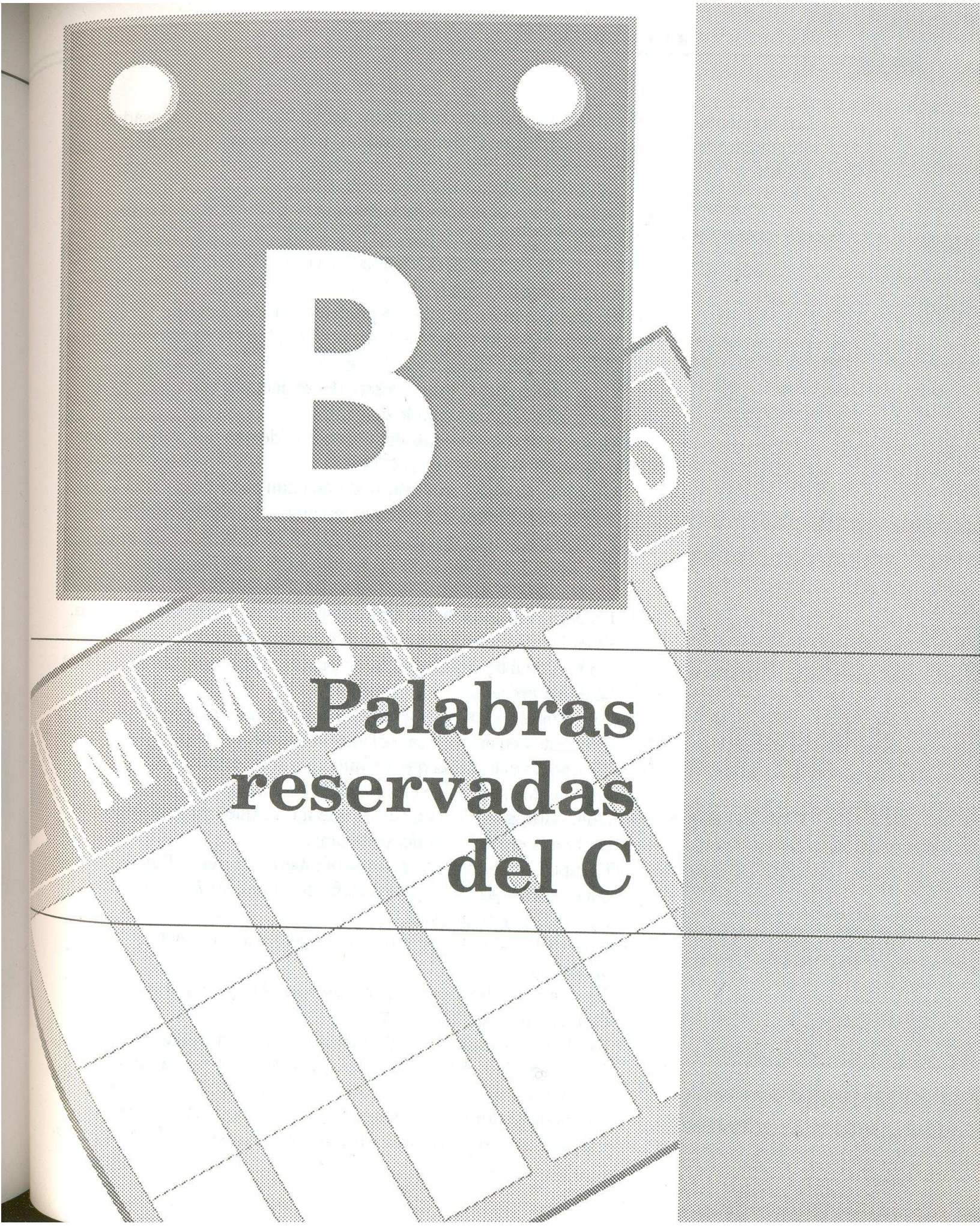
Tabla de caracteres ASCII

Tabla de caracteres ASCII

Hex.	*Dec.*	*Pantalla*	*Ctrl.*	*Tecla*
00h	0		NUL	^@
01h	1	☺	SOH	^A
02h	2	☻	STX	^B
03h	3	♥	ETX	^C
04h	4	♦	EOT	^D
05h	5	♣	ENQ	^E
06h	6	♠	ACK	^F
07h	7	•	BEL	^G
08h	8	◘	BS	^H
09h	9	○	HT	^I
0Ah	10	◙	LF	^J
0Bh	11	♂	VT	^K
0Ch	12	♀	FF	^L
0Dh	13	♪	CR	^M
0Eh	14	♫	SO	^N
0Fh	15	☼	SI	^O
10h	16	►	DLE	^P
11h	17	◄	DC1	^Q
12h	18	↕	DC2	^R
13h	19	‼	DC3	^S
14h	20	¶	DC4	^T
15h	21	§	NAK	^U
16h	22	▬	SYN	^V
17h	23	↨	ETB	^W
18h	24	↑	CAN	^X
19h	25	↓	EM	^Y
1Ah	26	→	SUB	^Z
1Bh	27	←	ESC	^[
1Ch	28	L	FS	^\
1Dh	29	↔	GS	^]
1Eh	30	▲	RS	^^
1Fh	31	▼	US	^_
20h	32			
21h	33	!		
22h	34	"		
23h	35	#		
24h	36	$		
25h	37	%		
26h	38	&		
27h	39	'		
28h	40	(		
29h	41	)		
2Ah	42	*		
2Bh	43	+		
2Ch	44	,		
2Dh	45	-		

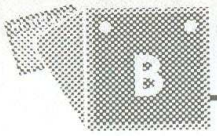
Hex.	Dec.	Pantalla
2Eh	46	.
2Fh	47	/
30h	48	0
31h	49	1
32h	50	2
33h	51	3
34h	52	4
35h	53	5
36h	54	6
37h	55	7
38h	56	8
39h	57	9
3Ah	58	:
3Bh	59	;
3Ch	60	<
3Dh	61	=
3Eh	62	>
3Fh	63	?
40h	64	@
41h	65	A
42h	66	B
43h	67	C
44h	68	D
45h	69	E
46h	70	F
47h	71	G
48h	72	H
49h	73	I
4Ah	74	J
4Bh	75	K
4Ch	76	L
4Dh	77	M
4Eh	78	N
4Fh	79	O
50h	80	P
51h	81	Q
52h	82	R
53h	83	S
54h	84	T
55h	85	U
56h	86	V
57h	87	W
58h	88	X
59h	89	Y
5Ah	90	Z
5Bh	91	[
5Ch	92	\
5Dh	93	]
5Eh	94	^
5Fh	95	_
60h	96	`
61h	97	a
62h	98	b
63h	99	c
64h	100	d
65h	101	e
66h	102	f
67h	103	g
68h	104	h
69h	105	i
6Ah	106	j
6Bh	107	k
6Ch	108	l
6Dh	109	m
6Eh	110	n
6Fh	111	o
70h	112	p
71h	113	q
72h	114	r
73h	115	s
74h	116	t
75h	117	u
76h	118	v
77h	119	w
78h	120	x
79h	121	y
7Ah	122	z
7Bh	123	{
7Ch	124	|
7Dh	125	}
7Eh	126	~
7Fh	127	Δ
80h	128	Ç
81h	129	ü
82h	130	é
83h	131	â
84h	132	ä
85h	133	à
86h	134	å
87h	135	ç
88h	136	ê
89h	137	ë

Hex.	*Dec.*	*Pantalla*
E6h	230	μ
E7h	231	τ
E8h	232	Φ
E9h	233	Θ
EAh	234	Ω
EBh	235	δ
ECh	236	∞
EDh	237	φ
EEh	238	∈
EFh	239	∩
F0h	240	≡
F1h	241	±
F2h	242	≥
F3h	243	≤
F4h	244	⌠
F5h	245	⌡
F6h	246	÷
F7h	247	≈
F8h	248	°
F9h	249	•
FAh	250	·
FBh	251	√
FCh	252	n
FDh	253	²
FEh	254	■
FFh	255	



B

Palabras reservadas del C

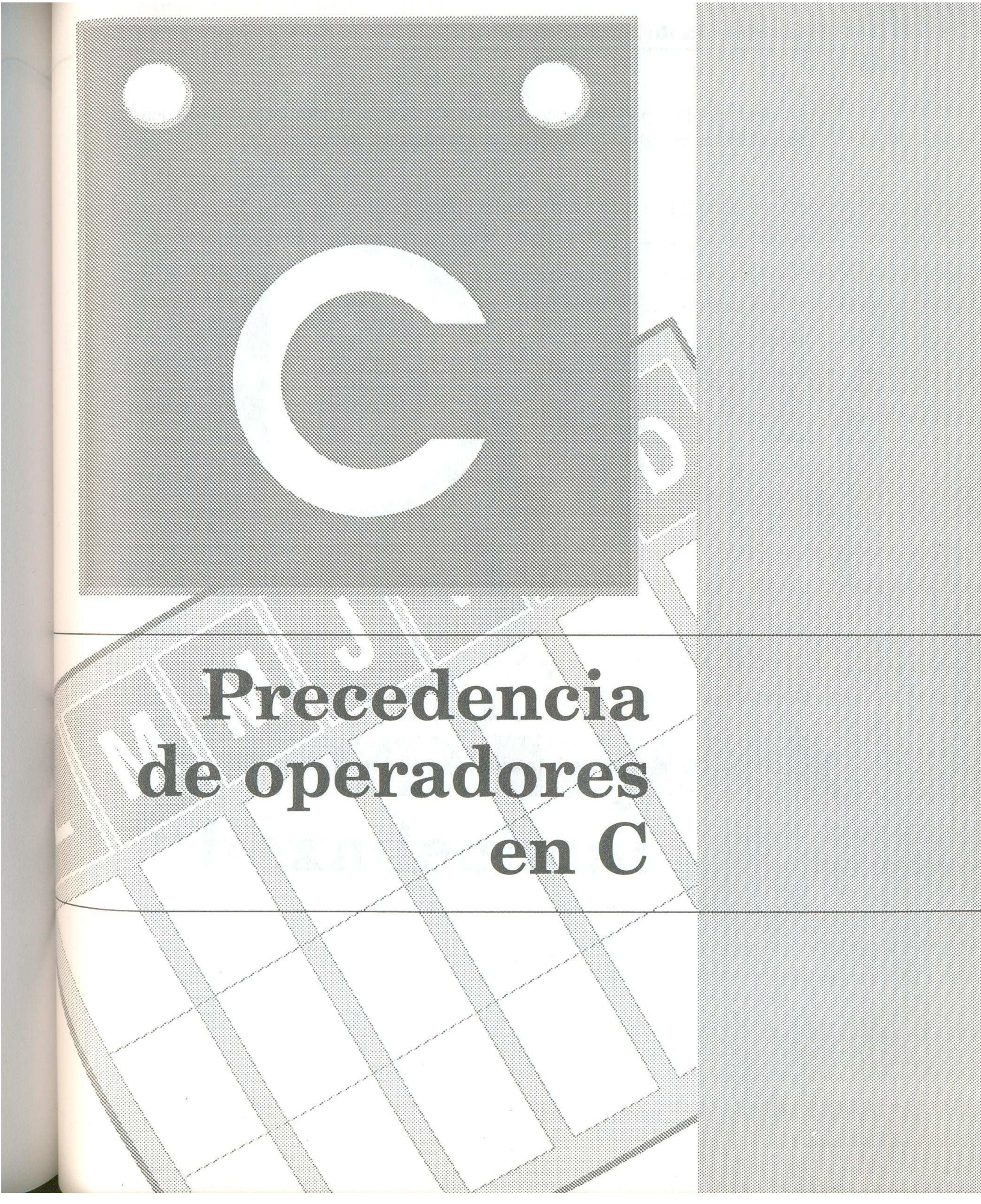


Palabras reservadas del C

Los siguientes identificadores son palabras reservadas del C. No deberán ser usados para ningún otro objetivo en un programa en C. Son permitidos, por supuesto, dentro de comillas dobles.

Palabra reservada	Descripción
`asm`	Una palabra clave del C que indica código de lenguaje ensamblador en línea.
`auto`	La clase de almacenamiento predeterminado.
`break`	Un comando del C que sale incondicionalmente de los enunciados `for`, `while`, `switch` y `do...while`.
`case`	Un comando del C usado dentro del enunciado `switch`.
`char`	El tipo de dato más simple del C.
`const`	Un modificador de datos del C que impide que una variable sea cambiada. Vea `volatile`.
`continue`	Un comando del C que reajusta a un enunciado `for`, `while` o `do...while` para la siguiente iteración.
`default`	Un comando del C usado dentro del enunciado `switch` para pescar cualquier instancia que no ha sido especificada con un enunciado `case`.
`do`	Un comando de ciclo del C usado junto con el enunciado `while`. El ciclo siempre ejecutará por lo menos una vez.
`double`	Un tipo de dato del C que puede guardar valores de punto flotante de doble precisión.
`else`	Un enunciado que señala enunciados alternativos que serán ejecutados cuando un enunciado `if` evalúe a `FALSO`.
`enum`	Un tipo de dato del C que permite que sean declaradas variables que aceptan solamente determinados valores.
`extern`	Un modificador de datos del C que indica que una variable será declarada en otra área del programa.
`float`	Un tipo de dato del C usado para números de punto flotante.
`for`	Un comando de ciclo que contiene secciones de *inicialización*, *incremento* y *condición*.
`goto`	Un comando del C que provoca un salto a una etiqueta predefinida.
`if`	Un comando del C usado para cambiar el flujo del programa con base en una decisión `CIERTO/FALSO`.
`int`	Un tipo de dato del C usado para guardar valores enteros.
`long`	Un tipo de dato del C usado para guardar valores enteros más grandes que `int`.
`register`	Un modificador de almacenamiento que especifica que una variable debe ser guardada, en caso de ser posible, en un `registro`.

Palabra reservada	Descripción
`return`	Un comando del C que hace que el flujo del programa salga de la función de trabajo y regrese a la función que la ha llamado. También puede ser usado para regresar un solo valor.
`short`	Un tipo de dato del C que es usado para guardar enteros. No es usado comúnmente y es del mismo tamaño que un `int` en la mayoría de las computadoras.
`signed`	Un modificador del C que es usado para indicar que una variable puede tener valores tanto positivos como negativos.
`sizeof`	Un operador del C que regresa el tamaño (cantidad de bytes) de un concepto.
`static`	Un modificador del C que es usado para indicar que el compilador debe retener el valor de la variable.
`struct`	Una palabra clave del C usada para combinar en un grupo variables del C de cualquier tipo de dato.
`switch`	Un comando del C usado para cambiar el flujo del programa en varias direcciones. Es usado junto con el enunciado `case`.
`typedef`	Un modificador del C usado para crear nuevos nombres para tipos existentes de variables y funciones.
`union`	Una palabra clave del C usada para permitir que varias variables compartan el mismo espacio de memoria.
`unsigned`	Un modificador del C que es usado para indicar que una variable contendrá solamente valores positivos. Vea `signed`.
`void`	Una palabra clave del C usada para indicar ya sea que una función no regresa nada o que un apuntador que está siendo usado es considerado genérico, es decir, capaz de apuntar a cualquier tipo de dato.
`volatile`	Un modificador del C que indica que una variable puede ser cambiada. Véase `const`.
`while`	Un enunciado de ciclo del C que ejecuta una sección de código mientras una condición permanezca `CIERTA`.



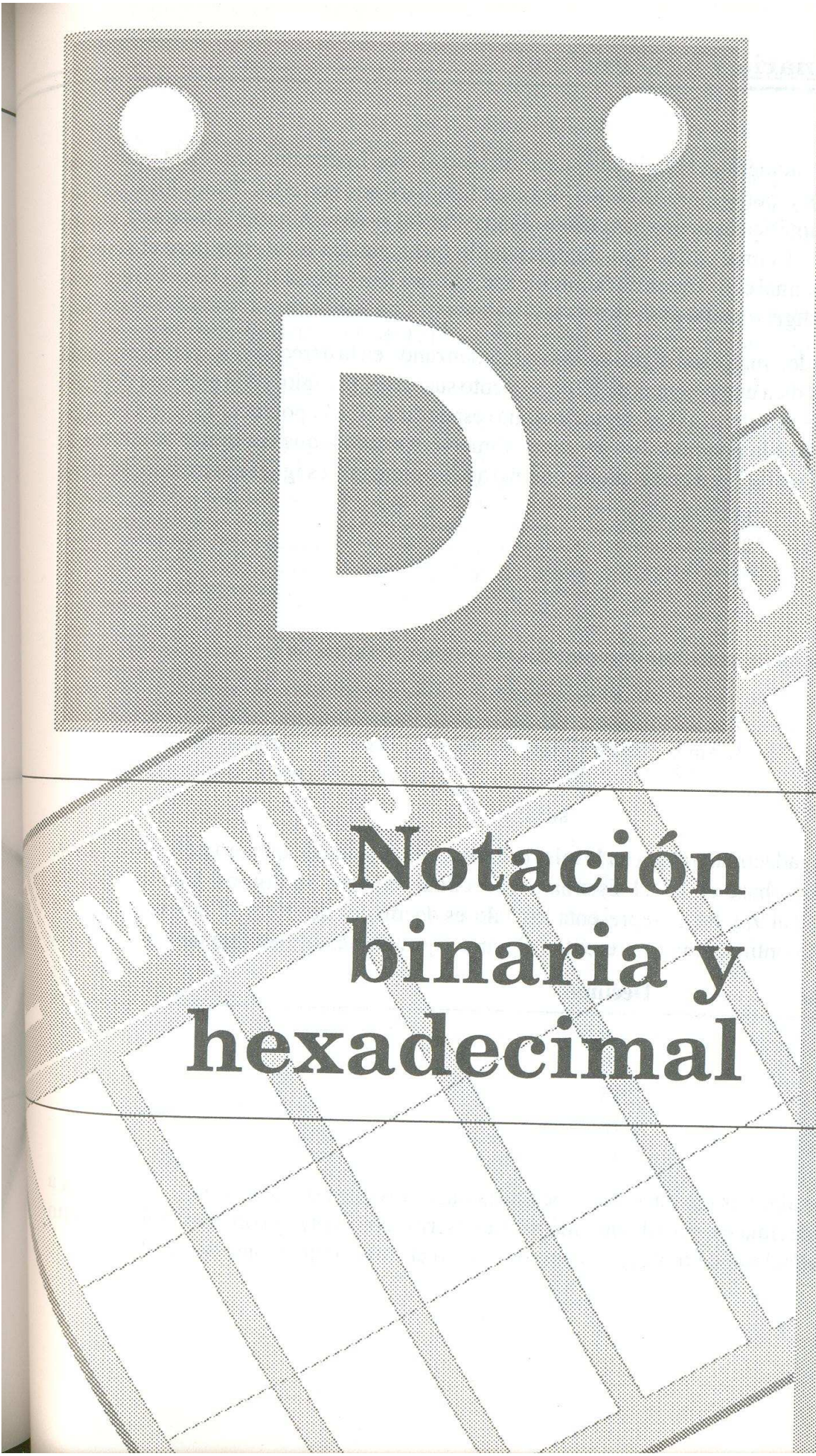
Precedencia de operadores en C

Lo siguiente lista todos los operadores del C en orden descendente de precedencia. Los operadores en el mismo renglón tienen la misma precedencia.

```
Nivel  1: () [] -> .
Nivel  2: ! ~ ++ – *(indirección) &(dirección de) (tipo)
          sizeof +(unario) -(unario)
Nivel  3: *(multiplicación) / %
Nivel  4: + -
Nivel  5: <<  >>
Nivel  6: < <=  > >=
Nivel  7: == !=
Nivel  8: &(AND a nivel bit)
Nivel  9: ^
Nivel 10: ¦
Nivel 11: &&
Nivel 12: ¦¦
Nivel 13: ?:
Nivel 14: = += -= *= /= %= &= ^= ¦= <<= >>=
Nivel 15: ,
```

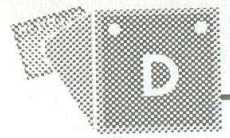


Nota: () es el operador de función y [] es el operador de arreglo.



D

Notación binaria y hexadecimal



Notación binaria y hexadecimal

Las notaciones numéricas binaria y hexadecimal son usadas frecuentemente en el mundo de la computación y, por lo tanto, es buena idea estar familiarizado con ellas. Todos los sistemas de notación numérica usan una base determinada. Para el sistema binario la base es 2 y para el sistema hexadecimal es 16. Para comprender lo que significa *base* considere el sistema de notación decimal con el que está familiarizado, que usa una base de 10. La base 10 requiere 10 diferentes dígitos, del 0 al 9.

En un número decimal, cada dígito sucesivo (comenzando en la derecha y moviéndose hacia la izquierda) indica una potencia de 10 en aumento sucesivo. El dígito de la extrema derecha especifica 10 a la potencia 0, el segundo dígito especifica 10 a la potencia 1, el tercer dígito especifica 10 a la potencia 2, y así sucesivamente. Debido a que cualquier número a la potencia 0 es igual a 1, y a que cualquier número a la potencia 1 es igual a sí mismo, se tiene

```
primer dígito:   10^0 = unidades
segundo dígito:  10^1 = decenas
tercer dígito:   10^2 = centenas
...
enésimo dígito: 10^(n-1)
```

Ahora, por ejemplo, podemos descomponer el número decimal 382.

```
382 (decimal)
___ ..........  2 x 10^0 = 2 x 1      =   2
__ ...........  8 x 10^1 = 8 x 10     =  80
_ ............  3 x 10^2 = 3 x 100    = 300
                                        -----
                                suma    = 382
```

El sistema hexadecimal trabaja en la misma forma, a excepción de que usa potencias de 16. Debido a que la base es 16, el sistema hexadecimal requiere 16 dígitos. Usa los dígitos normales del 0 al 9, y luego representa los valores decimales del 10 al 15 con las letras de la A a la F. A continuación se presentan algunos equivalentes hexadecimales/decimales:

Hexadecimal	Decimal
9	9
A	10
F	15
10	16
1F	31

Debido a que algunos números hexadecimales (aquellos que no tienen letras) se parecen a los números decimales, para distinguirlos se les escribe con el prefijo 0X. El siguiente es un número hexadecimal de tres dígitos descompuesto en sus componentes decimales:

```
2DA (hexadecimal)
___ ............ 10 x 16^0 = 10 x 1    =  10 (todo en decimal)
__ ............. 13 x 16^1 = 13 x 16   = 208
_ ..............  2 x 16^2 =  2 x 256  = 512
                                        ------
                                         730
```

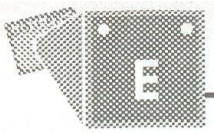
El sistema binario es de base 2 y, como tal, requiere solamente dos dígitos, 0 y 1. Cada lugar es un número binario representa una potencia de 2. Al descomponer un número binario se obtiene lo siguiente:

```
10010111 (binario)
________ ----------  1 x 2^0 = 1 x 1      =   1
_______ -----------  1 x 2^1 = 1 x 2      =   2
______ ------------  1 x 2^2 = 1 x 4      =   4
_____ -------------    x 2^3 = 0 x 8      =   0
____ --------------  1 x 2^4 = 1 x 16     =  16
___ ---------------    x 2^5 = 0 x 32     =   0
__ ----------------    x 2^6 = 0 x 64     =   0
_ -----------------  1 x 2^7 = 1 x 128    = 128
                                          -----
                                            151
```

Como puede ver, la notación binaria requiere muchos más dígitos que la decimal o hexadecimal para representar un valor dado. Sin embargo, es útil ya que la manera en que los números binarios representan valores es muy similar a la manera en que la computadora guarda valores enteros en memoria.

11

Prototipos de función y archivos de encabezado



Prototipos de función y archivos de encabezado

Este apéndice lista los prototipos de función contenidos en cada uno de los archivos de encabezado que se proporcionan en la mayoría de los compiladores de C. Las funciones marcadas con un asterisco en la columna izquierda son tratadas en el texto de esta obra.

Las funciones están listadas en orden alfabético, de acuerdo a su nombre, dentro del archivo de encabezado en donde son declaradas. Después a continuación de cada nombre está el prototipo completo. Observe que los prototipos de los archivos de encabezado usan una notación diferente a la que fue usada en todo el libro. Por cada parámetro que se use en una función, sólo se proporciona el tipo en el prototipo y no se incluye el nombre del parámetro. A continuación se presentan dos ejemplos:

```
int func1(int, int *);
int func1(int x, int *y);
```

Ambas declaraciones especifican dos parámetros: el primero es del tipo `int` y el segundo un apuntador a tipo `int`. Por lo que concierne al compilador, las dos declaraciones son equivalentes.

Tabla E.1. STDLIB.H: Funciones generales de la biblioteca estándar.

Tratada en este libro	Función	Prototipo de función
*	abort:	`void abort(void);`
	abs:	`int abs(int);`
*	atexit:	`int atexit(void (*)(void));`
*	atof:	`double atof(const char *);`
*	atoi:	`int atoi(const char *);`
*	atol:	`long atol(const char *);`
*	bsearch:	`void *bsearch(const void*, const void *, size_t,size_t, int (*)(const void *,const void *) );`
*	calloc:	`void *calloc(size_t,size_t);`
	div:	`div_t div(int, int);`
*	exit:	`void exit(int);`
*	free:	`void free(void *);`
	getenv:	`char *getenv(const char *);`

Tratada en este libro	Función	Prototipo de función
	labs:	`long int labs(long int);`
	ldiv:	`ldiv_t div(long int, long int);`
*	malloc:	`void *malloc(size_t);`
	mblen:	`int mblen(const char *, size_t);`
	mbstowcs:	`size_t mbstowcs(wchar_t *, const char *, size_t);`
	mbtowc:	`int mbtowc(wchar_t *, const char *, size_t);`
*	qsort:	`void qsort(void *,size_t,size_t,int (*)(const void*, const void *));`
	rand:	`int rand(void);`
*	realloc:	`void *realloc(void *,size_t);`
	srand:	`void srand(unsigned);`
	strtod:	`double strtod(const char *,char **);`
	strtol:	`long strtol(const char *, char **, int);`
	strtoul:	`unsigned long strtoul(const char *, char **, int );`
*	system:	`int system(const char *);`
	wcstombs:	`size_t wcstombs(char *, const wchar_t *, size_t);`
	wctomb:	`int wctomb(char *, wchar_t);`

Tabla E.2. Funciones estándar de entrada/salida de: STDIO.H.

Tratada en este libro	Función	Prototipo de función
	clearerr:	void clearerr(FILE *);
*	fclose:	int fclose(FILE *);
*	fcloseall:	int fcloseall(void);
*	feof:	int feof(FILE *);
*	fflush:	int fflush(FILE *);
*	fgetc:	int fgetc(FILE *);
	fgetpos:	int fgetpos(FILE *, fpos_t *);
*	fgets:	char *fgets(char *,int,FILE *);
*	flushall:	int flushall(void);
*	fopen:	FILE *fopen(const char *, const char *);
*	fprintf:	int fprintf(FILE *, const char *,...);
*	fputc:	int fputc(int,FILE *);
*	fputs:	int fputs(const char *,FILE *);
*	fread:	size_t fread(void *, size_t,size_t,FILE *);
	freopen:	FILE *freopen(const char *, const char *, FILE *);
*	fscanf:	int fscanf(FILE *,const char *,...);
*	fseek:	int fseek(FILE *,long,int);
	fsetpos:	int fsetpos(FILE *, const fpos_t *);
*	ftell:	long ftell(FILE *);
*	fwrite:	size_t fwrite(const void*, size_t,size_t,FILE *);

Tratada en este libro	Función	Prototipo de función
*	getc:	`int getc(FILE *);`
*	getch:	`int getch(void);`
*	getchar:	`int getchar(void);`
*	getche:	`int getche(void);`
*	gets:	`char *gets(char *);`
*	perror:	`void perror(const char *);`
*	printf:	`int printf(const char *,...);`
*	putc:	`int putc(int,FILE *);`
*	putchar:	`int putchar(int);`
*	puts:	`int puts(const char *);`
*	remove:	`int remove(const char *);`
*	rename:	`int rename(const char *,const char *);`
*	rewind:	`void rewind(FILE *);`
*	scanf:	`int scanf(const char *,...);`
	setbuf:	`void setbuf(FILE *, char *);`
	setvbuf:	`int setvbuf(FILE *, char *, int, size_t);`
	sprintf:	`int sprintf(char *, const char *,...);`
	sscanf:	`int sscanf(const char *, const char *,...);`
	tmpfile:	`FILE *tmpfile(void);`
*	tmpnam:	`char *tmpnam(char *);`
*	ungetc:	`int ungetc(int,FILE *);`

Tabla E.2. continuación

Tratada en este libro	Función	Prototipo de función
	vfprintf:	`int vfprintf(FILE *, const char *,...);`
	vprintf:	`int vprintf(FILE *, const char *,...);`
	vsprintf:	`int vsprintf(char *, const char *,...);`

Tabla E.3. TIME.H: Funciones de tiempo y fecha.

Tratada en este libro	Función	Prototipo de función
*	asctime:	`char *asctime(const struct tm *);`
*	clock:	`clock_t clock(void);`
*	ctime:	`char *ctime(const time_t *);`
	difftime:	`double diff_time(time_t, time_t);`
	gmtime:	`struct tm *gmtime(const time_t*);`
*	localtime:	`struct tm *localtime(const time_t *);`
*	mktime:	`time_t mktime(struct tm *);`
*	sleep:	`void sleep(time_t);`
*	strftime:	`size_t strftime(char *,size_t,const char *, const struct tm *);`
*	time:	`time_t time(time_t *);`

Tabla E.4. STRING.H: Funciones de cadena y carácter.

Tratada en este libro	Función	Prototipo de función
	memchr:	void *memchr(const void *, int, size_t);
	memcmp:	int memcmp(const void *, const void *, size_t);
	memcpy:	void *memcpy(void *, const void*, size_t);
	memmove:	void *memmove(void *, const void *, size_t);
	memset:	void *memset(void *, int, size_t);
*	strcat:	char *strcat(char *,const char *);
*	strchr:	char *strchr(const char *,int);
*	strcmp:	int strcmp(const char *,const char *);
*	strcmpi:	int strcmpi(const char *,const char *);
*	strcpy:	char *strcpy(char *,const char *);
*	strcspn:	size_t strcspn(const char *,const char *);
*	strdup:	char *strdup(const char *);
	strerror:	char *strerror(int);
*	strlen:	size_t strlen(const char *);
*	strlwr:	char *strlwr(char *);
*	strncat:	char *strncat(char *,const char *,size_t);

Tabla E.4. continuación

Tratada en este libro	Función	Prototipo de función
*	strncmp:	int strncmp(const char *, const char *,size_t);
*	strncpy:	char *strncpy(char *,const char *, size_t);
*	strnset:	char *strnset(char *,int,size_t);
*	strpbrk:	char *strpbrk(const char *,const char *);
*	strrchr:	char *strrchr(const char *,int);
*	strspn:	size_t strspn(const char *,const char *);
*	strstr:	char *strstr(const char *,const char *);
	strtok:	char *strtok(char *, const char *);
*	strupr:	char *strupr(char *);

Tabla E.5. CTYPE.H: Macros de clasificación y conversión de caracteres.

Tratada en este libro	Función	Prototipo de función
*	isalnum:	int isalnum(int);
*	isalpha:	int isalpha(int);
*	isascii:	int isascii(int);
*	iscntrl	int iscntrl(int);
*	isdigit:	int isdigit(int);
*	isgraph:	int isgraph(int);

Tratada en este libro	Función	Prototipo de función
*	islower:	int islower(int);
*	isprint:	int isprint(int);
*	ispunct:	int ispunct(int);
*	isspace:	int isspace(int);
*	isupper:	int isupper(int);
*	isxdigit:	int isxdigit(int);
	tolower:	int tolower(int);
	toupper:	int toupper(int);

Tabla E.6. MATH.H: Funciones matemáticas.

Tratada en este libro	Función	Prototipo de función
*	acos:	double acos(double);
*	asin:	double asin(double);
*	atan:	double atan(double);
*	atan2:	double atan2(double,double);
*	atof:	double atof(const char *);
*	ceil:	double ceil(double);
*	cos:	double cos(double);
*	cosh:	double cosh(double);
*	exp:	double exp(double);
*	fabs:	double fabs(double);
*	floor:	double floor(double);
*	fmod:	double fmod(double,double);

Tabla E.6. continuación

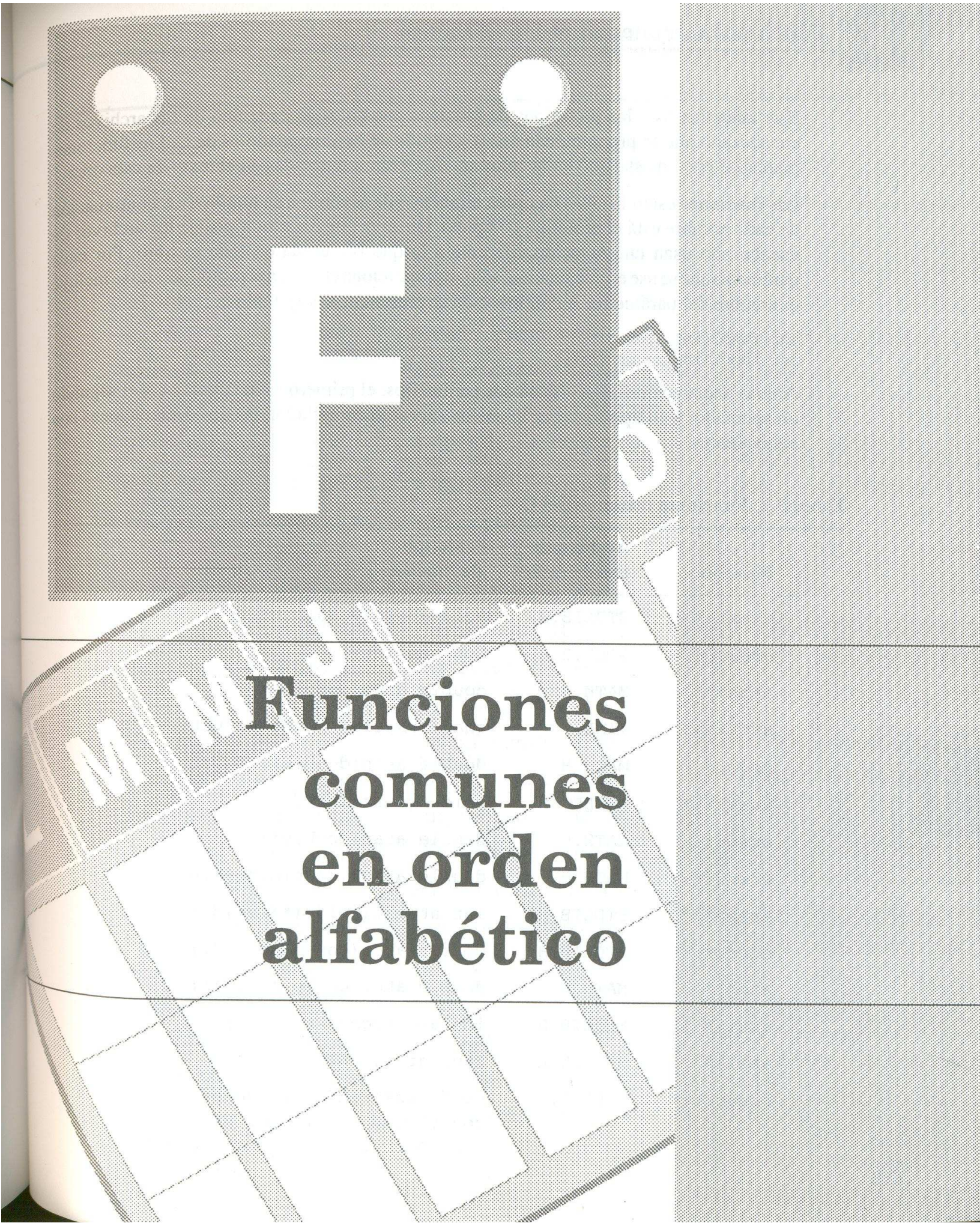
Tratada en este libro	Función	Prototipo de función
*	frexp:	`double frexp(double,int *);`
	ldexp:	`double ldexp(double, int);`
*	log:	`double log(double);`
*	log10:	`double log10(double);`
	modf:	`double modf(double, double *);`
*	pow:	`double pow(double,double);`
*	sin:	`double sin(double);`
*	sinh:	`double sinh(double);`
*	sqrt:	`double sqrt(double);`
*	tan:	`double tan(double);`
*	tanh:	`double tanh(double);`

Tabla E.7. ASSERT.H: Funciones de diagnóstico.

Tratada en este libro	Función	Prototipo de función
*	assert:	`void assert(int);`

Tabla E.8. STDARG.H: Macros de control de argumento variable.

Tratada en este libro	Función	Prototipo de función
*	va_arg:	`(type) va_arg(va_list, (type));`
*	va_end:	`void va_end(va_list);`
*	va_start:	`void va_start(va_list, lastfix);`



F

Funciones comunes en orden alfabético



Funciones comunes en orden alfabético

Este apéndice lista los prototipos de función contenidos en cada uno de los archivos de encabezado que se proporcionan con la mayoría de los compiladores de C. Las funciones marcadas con un asterisco en la columna izquierda son tratadas en el texto de esta obra.

Las funciones están listadas en orden alfabético de acuerdo a su nombre. A continuación de cada nombre está el prototipo completo. Observe que los prototipos de los archivos de encabezado usan una notación diferente a la que fue usada en todo el libro. Por cada parámetro que se use en una función, sólo se proporciona el tipo en el prototipo y no se incluye el nombre del parámetro. A continuación se presentan dos ejemplos:

```
int func1(int, int *);
int func1(int x, int *y);
```

Ambas declaraciones especifican dos parámetros: el primero es del tipo `int` y el segundo un apuntador a un tipo `int`. Por lo que concierne al compilador, las dos declaraciones son equivalentes.

Tabla F.1. Funciones comunes del C.

Función	Archivo de encabezado	Prototipo de función
abort:*	STDLIB.H	void abort(void);
abs:	STDLIB.H	int abs(int);
acos:*	MATH.H	double acos(double);
asctime:*	TIME.H	char *asctime(const struct tm *);
asin:*	MATH.H	double asin(double);
assert:*	ASSERT.H	void assert(int);
atan:*	MATH.H	double atan(double);
atan2:*	MATH.H	double atan2(double,double);
atexit:*	STDLIB.H	int atexit(void (*)(void));
atof:*	STDLIB.H	double atof(const char *);
atof:*	MATH.H	double atof(const char *);
atoi:*	STDLIB.H	int atoi(const char *);
atol:*	STDLIB.H	long atol(const char *);
bsearch:*	STDLIB.H	void *bsearch(const void*, const void *, size_t, size_t, int (*)(const void *, const void *));

Función	Archivo de encabezado	Prototipo de función
calloc:*	STDLIB.H	void *calloc(size_t,size_t);
ceil:*	MATH.H	double ceil(double);
clearerr:	STDIO.H	void clearerr(FILE *);
clock:*	TIME.H	clock_t clock(void);
cos:*	MATH.H	double cos(double);
cosh:*	MATH.H	double cosh(double);
ctime:*	TIME.H	char *ctime(const time_t *);
difftime:	TIME.H	double diff_time(time_t, time_t);
div:	STDLIB.H	div_t div(int, int);
exit:*	STDLIB.H	void exit(int);
exp:*	MATH.H	double exp(double);
fabs:*	MATH.H	double fabs(double);
fclose:*	STDIO.H	int fclose(FILE *);
fcloseall:*	STDIO.H	int fcloseall(void);
feof:*	STDIO.H	int feof(FILE *);
fflush:*	STDIO.H	int fflush(FILE *);
fgetc:*	STDIO.H	int fgetc(FILE *);
fgetpos:	STDIO.H	int fgetpos(FILE *, fpos_t *);
fgets:*	STDIO.H	char *fgets(char *, int,FILE *);
floor:*	MATH.H	double floor(double);
flushall:*	STDIO.H	int flushall(void);
fmod:*	MATH.H	double fmod(double,double);
fopen:*	STDIO.H	FILE *fopen(const char *, const char *);
fprintf:*	STDIO.H	int fprintf(FILE *,const char *, ...);
fputc:*	STDIO.H	int fputc(int,FILE *);
fputs:*	STDIO.H	int fputs(const char *,FILE *);

Tabla F.1. continuación

Función	Archivo de encabezado	Prototipo de función
fread:*	STDIO.H	size_t fread(void *, size_t,size_t,FILE *);
free:*	STDLIB.H	void free(void *);
freopen:	STDIO.H	FILE *freopen(const char *, const char *, FILE *);
frexp:*	MATH.H	double frexp(double,int *);
fscanf:*	STDIO.H	int fscanf(FILE *,const char *,...);
fseek:*	STDIO.H	int fseek(FILE *,long,int);
fsetpos:	STDIO.H	int fsetpos(FILE *,const fpos_t *
ftell:*	STDIO.H	long ftell(FILE *);
fwrite:*	STDIO.H	size_t fwrite(const void*, size_t,size_t, FILE *);
getc:*	STDIO.H	int getc(FILE *);
getch:*	STDIO.H	int getch(void);
getchar:*	STDIO.H	int getchar(void);
getche:*	STDIO.H	int getche(void);
getenv:	STDLIB.H	char *getenv(const char *);
gets:*	STDIO.H	char *gets(char *);
gmtime:	TIME.H	struct tm *gmtime(const time_t *);
isalnum:*	CTYPE.H	int isalnum(int);
isalpha:*	CTYPE.H	int isalpha(int);
isascii:*	CTYPE.H	int isascii(int);
iscntrl:*	CTYPE.H	int iscntrl(int);
isdigit:*	TYPE.H	int isdigit(int);
isgraph:*	CTYPE.H	int isgraph(int);
islower:*	CTYPE.H	int islower(int);

Función	Archivo de encabezado	Prototipo de función
isprint:*	CTYPE.H	int isprint(int);
ispunct:*	CTYPE.H	int ispunct(int);
isspace:*	CTYPE.H	int isspace(int);
isupper:*	CTYPE.H	int isupper(int);
isxdigit:*	CTYPE.H	int isxdigit(int);
labs:	STDLIB.H	long int labs(long int);
ldexp:	MATH.H	double ldexp(double, int);
ldiv:	STDLIB.H	ldiv_t div(long int, long int);
localtime:*	TIME.H	struct tm *localtime(const time_t *);
log:*	MATH.H	double log(double);
log10:*	MATH.H	double log10(double);
malloc:*	STDLIB.H	void *malloc(size_t);
mblen:	STDLIB.H	int mblen(const char *, size_t);
mbstowcs:	STDLIB.H	size_t mbstowcs(wchar_t *, const char *, size_t);
mbtowc:	STDLIB.H	int mbtowc(wchar_t *, const char *, size_t);
memchr:	STRING.H	void *memchr(const void *, int, size_t);
memcmp:	STRING.H	int memcmp(const void *, const void *, size_t);
memcpy:	STRING.H	void *memcpy(void *, const void *, size_t);
memmove:	STRING.H	void *memmove(void *, const void*, size_t);
memset:	STRING.H	void *memset(void *, int, size_t);
mktime:*	TIME.H	time_t mktime(struct tm *);
modf:	MATH.H	double modf(double, double *);

Tabla F.1. continuación

Función	Archivo de encabezado	Prototipo de función
perror:*	STDIO.H	void perror(const char *);
pow:*	MATH.H	double pow(double,double);
printf:*	STDIO.H	int printf(const char *,...);
putc:*	STDIO.H	int putc(int,FILE *);
putchar:*	STDIO.H	int putchar(int);
puts:*	STDIO.H	int puts(const char *);
qsort:*	STDLIB.H	void qsort(void*, size_t,size_t,int (*)(const void*, const void *));
rand:	STDLIB.H	int rand(void);
realloc:*	STDLIB.H	void *realloc(void *,size_t);
remove:*	STDIO.H	int remove(const char *);
rename:*	STDIO.H	int rename(const char *, const char *);
rewind:*	STDIO.H	void rewind(FILE *);
scanf:*	STDIO.H	int scanf(const char *,...);
setbuf:	STDIO.H	void setbuf(FILE *, char *);
setvbuf:	STDIO.H	int setvbuf(FILE *, char *, int, size_t);
sin:*	MATH.H	double sin(double);
sinh:*	MATH.H	double sinh(double);
sleep:*	TIME.H	void sleep(time_t);
sprintf:	STDIO.H	int sprintf(char *, const char *,...);
sqrt:*	MATH.H	double sqrt(double);
srand:	STDLIB.H	void srand(unsigned);
sscanf:	STDIO.H	int sscanf(const char *, const char *,...);

Función	Archivo de encabezado	Prototipo de función
strcat:*	STRING.H	char *strcat(char *,const char *);
strchr:*	STRING.H	char *strchr(const char *,int);
strcmp:*	STRING.H	int strcmp(const char *, const char *);
strcmpi:*	STRING.H	int strcmpi(const char *, const char *);
strcpy:*	STRING.H	char *strcpy(char *, const char *);
strcspn:*	STRING.H	size_t strcspn(const char *,const char *);
strdup:*	STRING.H	char *strdup(const char *);
strerror:	STRING.H	char *strerror(int);
strftime:*	TIME.H	size_t strftime(char *, size_t,const char *, const struct tm *);
strlen:*	STRING.H	size_t strlen(const char *);
strlwr:*	STRING.H	char *strlwr(char *);
strncat:*	STRING.H	char *strncat(char *,const char *,size_t);
strncmp:*	STRING.H	int strncmp(const char *,const char *,size_t);
strncpy:*	STRING.H	char *strncpy(char *,const char *, size_t);
strnset:*	STRING.H	char *strnset(char *,int, size_t);
strpbrk:*	STRING.H	char *strpbrk(const char *,const char *);
strrchr:*	STRING.H	char *strrchr(const char *, int);
strspn:*	STRING.H	size_t strspn(const char *, const char *);
strstr:*	STRING.H	char *strstr(const char *, const char *);
strtod:	STDLIB.H	double strtod(const char *, char **);
strtok:	STRING.H	char *strtok(char *, const char *);

Tabla F.1. continuación

Función	Archivo de encabezado	Prototipo de función
strtol:	STDLIB.H	long strtol(const char *, char **, int);
strtoul:	STDLIB.H	unsigned long strtoul(const char *, char **, int);
strupr:*	STRING.H	char *strupr(char *);
system:*	STDLIB.H	int system(const char *);
tan:*	MATH.H	double tan(double);
tanh:*	MATH.H	double tanh(double);
time:*	TIME.H	time_t time(time_t *);
tmpfile:	STDIO.H	FILE *tmpfile(void);
tmpnam:*	STDIO.H	char *tmpnam(char *);
tolower:	CTYPE.H	int tolower(int);
toupper:	CTYPE.H	int toupper(int);
ungetc:*	STDIO.H	int ungetc(int,FILE *);
va_arg:*	STDARG.H	(type) va_arg(va_list, (type));
va_end:*	STDARG.H	void va_end(va_list);
va_start:*	STDARG.H	void va_start(va_list, lastfix)
vfprintf:	STDIO.H	int vfprintf(FILE *, const char *,...);
vprintf:	STDIO.H	int vprintf(FILE *,const char *,...
vsprintf:	STDIO.H	int vsprintf(char *, const char *,...);
wcstombs:	STDLIB.H	size_t wcstombs(char *, const wchar_t *, size_t);
wctomb:	STDLIB.H	int wctomb(char *, wchar_t);

G

Respuestas

Este apéndice lista las respuestas para las secciones de cuestionarios y ejercicios que están al final de cada capítulo. Observe que es posible más de una solución para los ejercicios. En la mayoría de los casos solamente se da una de las muchas respuestas posibles. En otros casos hay información adicional para ayudarle a resolver el ejercicio.

Respuestas para el Día 1 "Comienzo"

Cuestionario

1. El C es poderoso, popular y portátil.
2. El compilador traduce el código fuente del C a instrucciones de lenguaje de máquina que la computadora puede entender.
3. Edición, compilación, enlace y prueba.
4. La respuesta a esta pregunta depende del compilador que usted tenga. Consulte los manuales.
5. La respuesta a esta pregunta depende del compilador que se tenga. Consulte los manuales.
6. La extensión adecuada para los archivos fuente del C es .C.
7. El nombre FILENAME.TXT sí compilaría. Sin embargo, es más adecuado usar una extensión .C en vez de .TXT.
8. Debe hacer cambios al código fuente para corregir los problemas. Luego recompile y vuelva a enlazar. Después de haber vuelto a enlazar ejecute el programa nuevamente, para ver si las correcciones de usted repararon el programa.
9. El lenguaje de máquina son instrucciones digitales o binarias que la computadora puede entender. Debido a que la computadora no puede entender el código fuente del C, un compilador traduce el código fuente a código de máquina, también llamado código objeto.
10. El enlazador combina el código objeto del programa con el código objeto de la biblioteca de funciones y crea un archivo ejecutable.

Ejercicios

1. Cuando se observa el archivo objeto, se ven muchos caracteres de control y otros galimatías. Entre los galimatías se puede llegar a ver partes del archivo fuente.

2. El programa calcula el área de un círculo. Le pide el radio al usuario y luego despliega el área. Este es el programa al que se hace referencia en el capítulo.

3. Este programa imprime un bloque de 10 por 10, hecho con el carácter `X`. Un programa similar se usa y explica en el Día 6, "Control básico del programa".

4. Este programa genera un error de compilación. Por ello, usted obtendrá un mensaje similar al siguiente:

```
Error: ch1ex4.c: Declaration terminated incorrectly
```

   Este error lo ocasiona el punto y coma al final de la línea 3. Si usted quita el punto y coma, este programa deberá compilar y enlazar correctamente.

5. Este programa compila bien, pero genera un error del enlazador. Por ello, usted obtendrá un mensaje parecido al siguiente:

```
Error: Undefined symbol _do_it in module...
```

   Este error surge debido a que el enlazador no puede encontrar una función llamada `do_it`. Para corregir el programa cambie `do_it` por `printf`.

6. En vez de un bloque de 10 por 10 rellenado con el carácter `X`, ahora el programa imprime un bloque de 10 por 10 con caras sonrientes.

7. Con este ejercicio usted tecleó un programa que se puede usar para imprimir listados. No requiere respuesta.

Respuestas para el Día 2 "Los componentes de un programa C"

Cuestionario

1. Un bloque.

2. La función `main()`.

3. Cualquier texto que esté entre /* y */ es un comentario de programa y es ignorado por el compilador. Use comentarios de programa para hacer notas acerca de la estructura y operación del programa.

4. Una función es una sección independiente de código de programa que ejecuta determinada tarea y a la cual se le ha asignado un nombre. Mediante el uso del nombre de la función, un programa puede ejecutar el código de la función.

5. Una función definida por el usuario la crea el programador. Una función de biblioteca se proporciona con el compilador de C.
6. Una directiva `#include` da instrucciones al compilador para que añada el código de otro archivo de disco al código fuente, durante el proceso de compilación.
7. Los comentarios no deben estar anidados. Aunque algunos compiladores le permiten hacerlo, otros no. Para mantener portátil al código no se les debe anidar.
8. Sí. Los comentarios pueden ser tan largos como se necesite. Un comentario da inicio con /* y no termina sino hasta que aparece un */.
9. A un archivo de inclusión también se le conoce como archivo de encabezado.
10. Un archivo de inclusión es un archivo en disco separado que contiene la información necesaria para que el compilador pueda usar diversas funciones.

Ejercicios

1. Recuerde que solamente se requiere la función `main()` en los programas en C. El siguiente es el programa más pequeño posible, pero no hace nada.

```
void main()
{
}
```

   También podría haberse escrito así:

```
void main(){}
```

2. Para una explicación a estas respuestas, revise el capítulo.

   a. Los enunciados están en las líneas 8, 9, 10, 12, 20 y 21.

   b. La única definición de variable está en la línea 18.

   c. El único prototipo de función (para `display_line()`) está en la línea 4.

   d. La definición de función para `display_line()` está en las líneas 16 a 22.

   e. Los comentarios están en las líneas 1, 15 y 23.

3. Un comentario es cualquier texto incluido entre /* y */. Los ejemplos pueden ser

```
/* Este es un comentario */
/*???*/
/*
Este es un
    tercer comentario */
```

4. El ejercicio cuatro es un programa que imprime el alfabeto en letras mayúsculas. Usted comprenderá mejor este programa cuando haya llegado al Día 10, "Caracteres y cadenas".

   La salida:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

5. Este programa cuenta e imprime la cantidad de caracteres y espacios que se teclean. Este programa también le será más claro después de que termine el Día 10.

Respuestas para el Día 3 "Variables y constantes numéricas"

Cuestionario

1. Una variable entera puede contener un número entero (un número sin parte fraccionaria), y una variable de punto flotante puede contener un número real (un número con parte fraccionaria).
2. Una variable tipo `double` tiene un rango mayor que una tipo `float` (puede guardar valores más grandes y más pequeños). Una variable tipo `double` también es más precisa que una tipo `float`.
3. Los cinco criterios que proporciona ANSI son los siguientes:
   1. El tamaño de un `char` es 1 byte.
   2. El tamaño de un `short` es menor o igual al tamaño de un `int`.
   3. El tamaño de un `int` es menor o igual al tamaño de un `long`.
   4. El tamaño de un `unsigned` es igual al tamaño de un `int`.
   5. El tamaño de un `float` es menor o igual al tamaño de un `double`.
4. Los nombres de constantes simbólicas hacen que sea más fácil de leer el código fuente. También facilitan el cambio de valores de constantes.
5.
```
#define MAXIMUM 100
```
   o
```
const int MAXIMUM = 100;
```

6. Las letras, los números y el símbolo de subrayado.
7. Los nombres para variables y constantes deben ser descriptivos de los datos que guardan. Los nombres de variable deben estar en minúsculas y los nombres de constantes en mayúsculas.
8. Las constantes simbólicas son símbolos que representan constantes literales.
9. Si es un `unsigned int`, que es de 2 bytes de longitud, el valor mínimo que puede guardar es 0. Si es signed, el mínimo es -32,768.

Ejercicios

1. a. Debido a que la edad de una persona puede ser considerada un número entero, y también a que una persona no puede tener una edad negativa, se sugiere un `unsigned int`.

   b. `unsigned int`.

   c. `float`

   d. Si lo que espera ganar en el año no es mucho, tal vez pueda funcionar una variable simple `unsigned int`. Si cree que puede ganar más de N$65,535, probablemente deba usar una variable `long`. (Tenga fe en sí mismo, use una `long`.)

   e. float (No olvide los lugares decimales para los centavos.)

   f. Dado que la calificación más alta siempre será 100, es una constante. Use un enunciado `const int` o uno `#define`.

   g. `float` (Si va a usar solamente números enteros emplee `int` o `long`.)

   h. Definitivamente un campo con signo. Puede ser `int`, `long` o `float`. Vea la respuesta para la pregunta d.

   i. `double`

2. (Las respuestas para los ejercicios 2 y 3 están combinadas.)

   Recuerde que un nombre de variable debe ser representativo del valor que guarda. Una declaración de variable es el enunciado que crea inicialmente a la variable. La declaración puede inicializar o no a la variable para que tenga algún valor. Puede usar cualquier nombre para la variable, excepto las palabras claves del C.

   a. `unsigned int edad;`

   b. `unsigned int peso;`

   c. `float radio = 3;`

```
d. long salario_anual;
e. float costo = 29.95;
f. const int calificación_máxima = 100;
   o
   #define CALIFICACION_MAXIMA 100
g. float temperatura;
h. long valor_neto = -30000;
i. double distancia_estrella;
```

3. (Vea las respuestas al ejercicio 2.)
4. Los nombres de variable válidos son b, c, e, g, h, i y j.

   Sin embargo, observe que aunque j es correcto no es prudente usar nombres de variable tan largos. (Además, ¿a quién le gustaría teclearlo?) La mayoría de los compiladores no toman en cuenta el nombre completo que se proporciona en j. En vez de ello, solamente toman en cuenta a los primeros caracteres. La mayoría de los compiladores diferencian solamente los primeros 31 caracteres.

   Los siguientes no son válidos:

   a. El nombre de una variable no puede comenzar con un número.
   d. No se puede usar un signo de gato (#) en un nombre de variable.
   f. No se puede usar un guión (-) en un nombre de variable.

Respuestas para el Día 4 "Enunciados, expresiones y operadores"

Cuestionario

1. Es un enunciado de asignación que le da instrucciones a la computadora para que sume 5 y 8 y asigne el resultado a la variable x.
2. Una expresión es cualquier cosa que evalúa a un valor numérico.
3. La precedencia relativa de los operadores.

4. Después del primer enunciado el valor de `a` es `10` y el valor de `x` es `11`. Después del segundo enunciado tanto `a` como `x` tienen el valor `11`. (Los enunciados se deben ejecutar por separado.)
5. 1
6. 19
7. (5 + 3) * 8 / (2 + 2)
8. 0
9. En el apéndice C, "Precedencia de operadores en C", se muestran los operadores del C y su precedencia.
   a. < tiene mayor precedencia que ==
   b. * tiene mayor precedencia que +
   c. != y == tienen la misma precedencia, por lo tanto, se evalúan de izquierda a derecha.
   d. >= tiene la misma precedencia que >. Use paréntesis si necesita emplear más de un operador relacional en un enunciado o expresión.
10. Los operadores de asignación compuesta le permiten a usted combinar una operación matemática binaria con una operación de asignación, proporcionando así una notación abreviada. Los operadores compuestos, presentados en el Día 4, "Enunciados, expresiones y operadores", son +=, -=, /=, *= y %=.

Ejercicios

1. El listado debió haber funcionado, aunque no hay estado muy bien estructurado. El propósito del listado fue demostrar que el espacio en blanco es irrelevante en lo que se refiere a la ejecución del programa. El espacio en blanco se debe usar para hacer más legibles los programas.
2. La siguiente es una mejor manera de estructurar el listado del ejercicio 1:

```
#include <stdio.h>

int x, y;

main()

{

    printf("\nEnter two numbers");

    scanf( "%d %d",&x,&y);
```

```
    printf("\n\n%d is bigger",(x>y)?x:y);
    return 0;
}
```

El listado pide dos números, x y y, y luego imprime al que sea mayor.

3. Los únicos cambios que se necesitan en el listado 4.1 son los siguientes:

```
16:     printf("\n%d   %d", a++, ++b);
17:     printf("\n%d   %d", a++, ++b);
18:     printf("\n%d   %d", a++, ++b);
19:     printf("\n%d   %d", a++, ++b);
20:     printf("\n%d   %d", a++, ++b);
```

4. El siguiente fragmento de código es solamente uno de los muchos ejemplos posibles. Sirve para verificar si x es mayor o igual a 1 y si x es menor o igual a 20. Si se satisfacen estas dos condiciones, x es asignada a y. Si las condiciones no se satisfacen, x no es asignado a y, y por lo tanto, y permanece igual.

```
if ((x >= 1) && (x <= 20))
 y = x;
```

5. `y = ((x >= 1) && (x <= 20)) ? x : y;`

Nuevamente, si el enunciado es cierto, x es asignada a y; en caso contrario, y se asigna a sí misma y, por lo tanto, no tiene efecto.

6.
```
if (x < 1 && x > 10)
        enunciado;
```

7.
```
a. (1 + 2 * 3) = 7
b. 10 * 3 * 3 - (1 + 2) = 0
c. ((1 + 2) * 3) = 9
d. (5 == 5) = 1 (Cierto)
e. (x = 5) = 5
```

8. Si x = 4, y = 6 y z = 2, determine cuáles de las siguientes se evalúan como verdadero o falso:

a. `cierto`

b. `falso`

c. Cierto. Observe que sólo hay un signo de igual, haciendo que el `if` sea una *asignación* en vez de una *relación*.

d. `Cierto`

9. No hicimos lo que pide este ejercicio, mas, sin embargo, puede obtener la respuesta a esta pregunta a partir de este `if` anidado. (También lo podría haber indentado en forma diferente.)

```
if (edad < 21 )
    printf("Usted no es un adulto");
else if( edad >= 65 )
        printf("Usted es un anciano");
    else
        printf( "Usted es un adulto" );
```

10. Este programa tiene cuatro errores. El primero está en la línea 4. El enunciado de asignación debe terminar con un punto y coma y no con dos puntos. El segundo error es el punto y coma al final del enunciado `if` en la línea 8. El tercer error es común: en el enunciado `if` el operador de asignación (=) fue usado en vez del operador relacional (==). El último error es la palabra `otherwise` en la línea 10. Debe ser `else`.

```
 1: /* a program with problems... */
 2: #include <stdio.h>
 3:
 4: int x = 1;
 5:
 6: main()
 7: {
 8:     if( x == 1)
 9:         printf(" x equals 1" );
10:     else
11:         printf(" x does not equal 1");
12:
13:     return
14: }
```

Respuestas para el Día 5 "Funciones: lo básico"

Cuestionario

1. ¡Claro que sí! (Bueno, está bien, ésta es una pregunta capciosa, pero más le vale que haya respondido "sí", si quiere llegar a ser un buen programador de C.)
2. La programación estructurada toma un problema complejo de programación y lo divide en varias tareas más simples, y más fáciles de manejar de una en una.
3. Después de que usted haya dividido el programa en varias tareas más simples, escriba una función para ejecutar cada tarea.
4. La primera línea de una definición de función debe ser el encabezado de tal función. Contiene el nombre de la función, su tipo de retorno y su lista de parámetros.
5. Una función puede regresar un valor o ningún valor. El valor puede ser de cualquiera de los tipos de variables del C. En el Día 18, "Obteniendo más de las funciones", verá la manera de regresar varios valores desde una función.
6. Una función que no regresa nada debe ser del tipo `void`.
7. Una definición de función es la función completa, incluidos el encabezado y todos los enunciados de la función. La definición determina qué acciones se llevan a cabo cuando la función se ejecuta. El prototipo es una sola línea, idéntica al encabezado de función, pero termina con un punto y coma. El prototipo le informa al compilador el nombre de la función, el tipo de retorno y la lista de parámetros.
8. Una variable local es declarada dentro de una función.
9. Las variables locales son independientes de otras variables en el programa.

Ejercicios

1. `float hazlo(char a, char b, char c)`

   Añada un punto y coma al final y tendrá el prototipo de función. Para hacer un encabezado de función deberá tener a continuación los enunciados de la función encerrados entre llaves.

2. Esta es una función `void`. Como en el ejercicio uno, para crear el prototipo añada un punto y coma al final. En un programa real el encabezado es seguido por los enunciados de la función.

```
void imprime_un_número( int un_número)
```

3. a. `int`

   b. `long`

4. Hay dos problemas en este listado. El primero es que la función es declarada como `void` y, sin embargo, regresa un valor. El enunciado `return` debe eliminarse. El segundo problema está en la línea 5. La llamada `print_msg()` pasa un parámetro (una cadena). El prototipo indica que esta función tiene una lista de parámetros `void` y, por lo tanto, no se le debe pasar nada. El siguiente es el listado, ya corregido:

```
#include <stdio.h>
void print_msg( void );
main()
#include <stdio.h>
void print_msg(void);
main()
{
    print_msg();
}
void print_msg(void)
{
puts( -This is a message to print+ );
}
```

5. No debe haber un punto y coma al final del encabezado de función.

6. Solamente es preciso cambiar la función `larger_of()`:

```
19: int larger_of( int a, int b)
20: {
21:     int save;
22:
23:     if (a > b)
24:         save = a;
25:     else
26:         save = b;
27:
```

```
28:     return save;
29: }
```

7. Lo siguiente asume que los dos valores son enteros y que se regresa un entero:

```
int product( int x, int y )
{
    return (x * y);
}
```

8. Este listado le pide a usted que revise los valores que pasen. Nunca asuma por completo que los valores pasados son correctos.

```
int divide_em( int a, int b )
{
    int answer = 0;
    if( b == 0 )
        answer = 0;

 else
        answer = a/b;
    return answer;
}
```

9. Aunque lo siguiente usa `main()`, pudiera tratarse de cualquier función. Las líneas 7, 8 y 9 muestran las llamadas a las dos funciones. Las líneas 11 a 14 imprimen los valores.

```
 1: main()
 2: {
 3:     int number1 = 10,
 4:         number2 = 5;
 5:     int x, y, z;
 6:
 7:     x = product( number1, number2 );
 8:     y = divide_em( number1, number2 );
 9:     z = divide_em( number1, 0 );
10:
11:     printf( "\nnumber1 is %d and number2 is %d", number1,
                number2 );
12:     printf( "\nnumber1 * number2 is %d", x );
```

```
13:     printf( "\nnumber1 / number2 is %d", y );
14:     printf( "\nnumber1 / 0 is %d", z );
15:
16:     return 0;
17: }
```

10.
```
/* Promedia cinco valores float dados por el usuario. */
#include <stdio.h>

float v, w, x, y, z, answer;

float average(float a, float b, float c, float d, float e);

main()
{
    puts("Enter five numbers:");
    scanf("%f%f%f%f%f", &v, &w, &x, &y, &z);
    answer = average(v, w, x, y, z);
    printf("The average is %f.", answer);
}

float average( float a, float b, float c, float d, float e)
{
    return ((a+b+c+d+e)/5);
}
```

11. La siguiente es la respuesta usando variables tipo `int`. Sólo puede ejecutarse con valores menores o iguales a 9. Para usar valores mayores que 9 se necesita cambiar los valores a tipo `long`.

```
/* Este es un programa con una función recursiva. */
#include <stdio.h>
int three_powered( int power );
main()
{
```

```
    int a = 4;
    int b = 9;
    printf( "\n3 to the power of %d is %d", a,
            three_powered(a) );
    printf( "\n3 to the power of %d is %d", b,
            three_powered(b) );
}
int three_powered( int power )
{
    if ( power < 1 )
        return( 1 );
    else
        return( 3 * three_powered( power - 1 ));
    }
```

Respuestas para el Día 6 "Control básico del programa"

Cuestionario

1. El primer valor de índice de un arreglo en C es 0.
2. Un enunciado `for` contiene expresiones de inicialización e incremento como parte del comando.
3. Un `do...while` contiene el enunciado `while` al final, y siempre ejecuta el ciclo por lo menos una vez.
4. Sí. Un enunciado `while` puede realizar la misma tarea que un enunciado `for`. Sin embargo, usted necesitará hacer dos cosas adicionales. Deberá inicializar cualquier variable antes de iniciar el comando `while` y necesitará incrementar cualquier variable como parte del ciclo `while`.
5. No se pueden traslapar los ciclos. El ciclo anidado debe estar completamente dentro del ciclo externo.

6. Sí. Un enunciado `while` se puede anidar en un ciclo `do...while`. Se puede anidar cualquier comando dentro de cualquier otro comando.

Ejercicios

1. `long arreglo[50];`

2. Observe que en la siguiente respuesta el quincuagésimo elemento tiene índice 49. Recuerde que los arreglos comienzan en 0.

```
arreglo[49] = 123.456;
```

3. Cuando el enunciado termina, `x` es igual a 100.

4. Cuando el enunciado termina, `contador` es igual a 11. (`contador` comienza en 2 y se incrementa en 3 mientras sea menor que 10.)

5. El ciclo interno imprime 5 `X`. El ciclo externo imprime el ciclo interno 10 veces. Esto da un total de 50 `X`.

6.
```
int x;

for( x = 1; x <= 100; x += 3);
```

7.
```
int x = 1;

while( x <= 100 )
   x += 3;
```

8.
```
int ctr = 1;
do
{
   ctr += 3;
} while( ctr < 100 );
```

9. Este programa nunca termina. `record` es inicializado a 0. Luego el ciclo `while` revisa para ver si `record` es menor que 100. 0 es menor que 100 y, por lo tanto, el ciclo ejecuta imprimiendo los dos enunciados. Luego el ciclo vuelve a revisar la condición. 0 es todavía y siempre será menor que 100, por lo que el ciclo continúa. `record` necesita ser incrementado dentro de las llaves. La siguiente línea se debe añadir después de la segunda llamada a la función `printf()`:

```
record++;
```

10. El uso de una constante definida es común en los ciclos. Usted verá ejemplos similares a este fragmento de código durante las semanas dos y tres. El problema de este fragmento es simple. No debe haber punto y coma al final del enunciado `for`. Se trata de un error común.

Respuestas para el Día 7 "Entrada/salida básica"

Cuestionario

1. Existen dos diferencias entre `puts()` y `printf()`.
   a. `printf()` puede imprimir parámetros de variables.
   b. `puts()` añade automáticamente un carácter de nueva línea al final de la cadena que imprime.
2. El archivo de encabezado STDIO.H se debe incluir cuando se use `printf()`.
3. Lo siguiente es lo que hacen las secuencias de escape.
   a. `\\` imprime una diagonal invertida.
   b. `\b` imprime un retroceso
   c. `\n` imprime una nueva línea
   d. `\t` imprime un tabulador
   e. `\a` hace sonar la bocina (alerta)
4. Se deben usar los siguientes especificadores de conversión:
   a. `%c` para un solo carácter
   b. `%d` para un entero decimal con signo
   c. `%f` para un número decimal de punto flotante
5. Lo siguiente es lo que se imprime en el texto literal de `puts()`:
   a. `b` imprime el carácter literal *b*
   b. `\b` imprime un carácter de retroceso
   c. `\` examina el siguiente carácter para determinar un carácter de escape (vea la tabla 7.1)
   d. `\\` imprime una sola diagonal invertida

Ejercicios

1. `puts()` añade automáticamente la nueva línea y `printf()` no lo hace.

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

2.
```
char c1, c2;
int d1;
scanf( "%c %ud %c", &c1, &d1, &c2 );
```

3. La respuesta de usted puede variar.

```
#include <stdio.h>
int x;

main()
{
   puts( "Teclee un valor entero" );
   scanf( "%d", &x );


   printf( "\nEl valor tecleado es %d", x );

}
```

4. Es común añadir condiciones que permitan solamente la aceptación de valores específicos. La siguiente es una forma de resolver este ejercicio.

```
#include <stdio.h>
int x;

main()
{
   puts( "Teclee un valor entero par" );
   scanf( "%d", &x );
   while( x % 2 != 0)
   {
      printf( "\n%d no es par, por favor teclee un numero par: ",
x );
      scanf( "%d", &x );
   }
   printf( "\nEl valor tecleado es %d", x );

}
```

5.
```
#include <stdio.h>
int arreglo[6], x, numero;

main()
{
 /* hace ciclo 6 veces o hasta que el último elemento tecleado sea
99 */
 for( x = 0; x < 6 && número != 99; x++ )
{
   puts( "Teclee un valor entero par o 99 para terminar" );
   scanf( "%d", &número );
   while( número % 2 == 1 && número !=  99)
   {
      printf( "\n%d no es par, por favor teclee un número par: ",
              número );
      scanf( "%d", &número );
   }
   arreglo[x] = número;
   }
   /* ahora lo imprimimos... */
    for( x = 0; x < 6 && arreglo[x] != 99; x++ )
   {
      printf( "\nEl valor tecleado es %d", arreglo[x] );
   }
}
```

6. Las respuestas anteriores ya son programas ejecutables. El único cambio que se necesita hacer es en el `printf()` final. Para imprimir cada valor separado por un tabulador cambie el enunciado `printf()` para que diga:

```
printf( "%d\t", arreglo[x]);
```

7. No se puede incluir comillas dentro de comillas. Para imprimir comillas dentro de comillas use usted el carácter de escape \". Lo siguiente es la versión correcta:

```
printf( "Jack said, \"Peter Piper picked a peck of pickled \
            peppers.\"++);
```

8. Este listado tiene tres errores. El primero es la ausencia de comillas en el enunciado `printf()`. El segundo es la falta del operador (&) *dirección de* en la variable answer del `scanf()`. El último error también está en el enunciado `scanf()`. En vez de `"%f"` debiera tener `"%d"`, ya que answer es una variable tipo `int` y no tipo `float`. Lo siguiente es lo correcto:

```
int get_1_or_2( void )
{
   int answer = 0;
   while( answer < 1 || answer > 2 )
    {
      printf("Teclee 1 para Yes, 2 para No");  /* corregido */
      scanf( "%d", &answer );                  /* corregido */
    }
    return answer;
}
```

9. A continuación aparece la función completa `print_report` para el listado 7.1:

```
void print_report( void )
{
     printf( "\nSAMPLE REPORT" );
     printf( "\n\nSequence\tMeaning" );
     printf( "\n=========\t=======" );
     printf( "\n\\a\t\tbell (alert)" );
     printf( "\n\\b\t\tbackspace" );
     printf( "\n\\n\t\tnew line" );
     printf( "\n\\t\t\thorizontal tab" );
     printf( "\n\\\\t\tbackslash" );
     printf( "\n\\\?\t\tquestion mark" );
     printf( "\n\\\'\t\tsingle quote" );
     printf( "\n\\\"\t\tdouble quote" );
     printf( "\n...\t\t...");
}
```

10.
```
/* Recibe dos valores de punto flotante y */
/* despliega su producto. */

#include <stdio.h>

float x, y;

main()
{
```

```
    puts("Teclee dos valores: ");
    scanf("%f %f, &x, &y);
    printf("\nEl producto es %f", x * y);
}
```

11. El siguiente programa solicita 10 enteros y despliega la suma de los mismos.

```
/* Recibe 10 enteros y despliega su suma. */

#include <stdio.h>

int contador, temp;
long total = 0;     /* Use tipo long para asegurarse de que no se */
                    /* exceda el máximo para tipo int.*/

main()
{
   for (contador = 1; contador <=10; contador++)
      {
      printf("Teclee el entero número %d: ", contador);
      scanf("%d", &temp);
      total += temp;
      }

   printf("\n\nEl total es %d", total);
}
```

12.
```
/* Recibe enteros y los guarda en un arreglo, deteniéndose */
/* cuando se teclea un cero. Busca y despliega los valores */
/* máximo y mínimo del arreglo */

#include <stdio.h>

#define MAX 100

int arreglo[MAX];
int contador = -1, maximo, minimo, num_dado, temp;

main()
{
   puts("Teclee valores enteros, uno por línea.");
   puts("Teclee 0 cuando termine.");
```

```
   /* Recibe los valores */

do
{
   scanf("%d", &temp);
   arreglo[++contador] = temp;
} while ( contador < (MAX-1) && temp != 0 );

num_dado = contador;

/* Encuentra el mayor y menor */
/* Primero pone el máximo a un valor muy pequeño,*/
/* y el mínimo a un valor muy grande.*/

maximo = -32000;
minimo = 32000;

for (contador = 0; contador <= num_dado && arreglo[contador]!= 0;
contador++)

{
      if (arreglo[contador] > máximo)
          máximo = arreglo[contador];

      if (arreglo[contador] < mínimo )
          mínimo = arreglo[contador];
}

   printf("\nEl valor máximo es %d", máximo);
   printf("\nEl valor mínimo es %d", mínimo);
}
```

Respuestas para el Día 8 "Arreglos numéricos"

Cuestionario

1. Todos ellos, pero de uno en uno. Un arreglo solamente puede contener datos de un solo tipo.

2. 0. Sin importar el tamaño del arreglo, en el C todos los arreglos comienzan con el subíndice 0.
3. n-1.
4. El programa compila y ejecuta, pero produce resultados impredecibles.
5. En el enunciado de declaración, ponga usted después del nombre un par de corchetes para cada dimensión. Cada par de corchetes contiene el número de elementos de la dimensión en cuestión.
6. 240. Esto se determina al multiplicar 2 por 3 por 5 y por 8.

Ejercicios

1. `int uno[1000], dos[1000], tres[1000];`
2. `int arreglo[10] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };`
3. Este ejercicio puede ser resuelto de varias formas. La primera es inicializar el arreglo cuando es declarado:

```
int ochentayocho[88] = {88,88,88,88,88,88,88,...,88};
```

Esto requeriría que se pusiera ochenta y ocho veces ochenta y ocho entre las llaves en vez de usar "..." como lo hice. Este método no es recomendable para inicializar un arreglo tan grande. Lo siguiente es un mejor método.

```
ochentayocho[88];
int x;

for ( x = 0; x < 88; x++ )
  ochentayocho[x] = 88;
```

4.
```
cosa[12][10];
int sub1, sub2;

for( sub1 = 0; sub1 < 12; sub1++ )
   for( sub2 = 0; sub2 < 10; sub2++ )
      cosa[sub1][sub2] = 0;
```

5. Tenga cuidado con este fragmento. Es fácil caer en el error que aquí se presenta. Observe que el arreglo es de 10 por 3 pero se inicializó como 3 por 10.

Dicho en otras palabras, el subíndice de la izquierda es declarado como 10 y, sin embargo, el ciclo `for` usa `x` como subíndice izquierdo. `x` es incrementado con 3 valores. El subíndice derecho es declarado como 3 y, sin embargo, el segundo ciclo `for` usa `y` como el subíndice derecho. `y` es incrementado con 10 valores. Esto puede causar resultados impredecibles. Se puede componer este programa de dos formas. La primera es intercambiar a `x` y `y` en la línea que hace la inicialización.

```
int x, y;
int array[10][3];
main()
{
   for ( x = 0; x < 3; x++ )
      for ( y = 0; y < 10; y++ )
         array[y][x] = 0;                    /* modificado */
}
```

La segunda forma (que es más recomendable) consiste en alternar los valores en los ciclos `for`.

```
int x, y;
int array[10][3];
main()
{
   for ( x = 0; x < 10; x++ )                /* modificado */
      for ( y = 0; y < 3; y++ )              /* modificado */
         array[x][y] = 0;
}
```

6. Esperamos que este error haya sido fácil de corregir. Este programa inicializa un elemento del arreglo que está fuera de rango. Si se tiene un arreglo con 10 elementos sus subíndices van del 0 al 9. Este programa inicializa elementos con subíndices del 1 al 10. No se puede inicializar a `array[10]` porque no existe. El enunciado `for` debe ser cambiado para que sea como alguno de los siguientes:

```
for( x = 1; x <=9; x++ )  /* inicializa 9 de los 10 elementos */
```

o

```
for( x = 0; x <= 9; x++ )
```

Observe que `x <= 9` es lo mismo que `x < 10`. Cualquiera es adecuado, pero `x < 10` es más común.

7. La siguiente es una de las muchas respuestas posibles:

```
/* Ejercicio 8.7 - Uso de arreglos de dos dimensiones y rand() */

#include <stdio.h>
#include <stdlib.h>

/* Declara el arreglo */

int arreglo[5][4];
int a, b;

main()
{
   for ( a = 0; a < 5; a++ ,
      {
          for ( b = 0; b < 4; b++ )
          {
              arreglo[a][b] = rand();
          }
       }

       /* ahora imprime los elementos del arreglo */

      for ( a = 0; a < 5; a++ )
      {
          for ( b = 0; b < 4; b++ )
          {
               printf( "%d\t", arreglo[a][b] );

          }
          printf( "\n" );       /* avanza un renglón */
      }
            return 0;
}
```

8. La siguiente es una de las muchas respuestas posibles:

```
/* EX8-8.C. RANDOM.C usando un arreglo de una sola dimensión. */
```

```
#include <stdio.h>
#include <stdlib.h>
/* Declara un arreglo de una sola dimensión con 1000 elementos */

int random[1000];
int a, b, c;
long total = 0;

main()
{
/* Llena el arreglo con números al azar. La función de biblioteca */
/* del C, rand(), regresa un número al azar. Use un ciclo for */
/* por cada subíndice del arreglo. */

   for (a = 0; a < 1000; a++)
   {
      random[a] = rand();
      total += random[a];
   }

   /* Ahora despliega los elementos del arreglo de 10 en 10 */

   for (a = 0; a < 1000; a++)
   {
      printf("\nrandom[%d] = ", a);
      printf("%d", random[a]);

         if ( a % 10 == 0 && a > 0 )
         {
            printf("\nOprima una tecla para continuar, CTRL-C para

                  salir.");
                  getch();
          }
   }
   printf("\n\nAverage is: %ld", total/1000);

}  /* fin de main() */
```

9. A continuación se presentan dos soluciones. La primera inicializa el arreglo al momento de ser declarado, y la segunda lo inicializa en un ciclo `for`:

Respuesta 1:

```
/* EX8-9.c */

#include <stdio.h>

/* Declara un arreglo de una sola dimensión */

int elementos[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int idx;

main()
{
   for (idx = 0; idx < 10; idx++)
   {
      printf( "\nelementos[%d] = %d ", idx, elementos[idx] );
   }
}     /* fin de main() */
```

Respuesta 2:

```
/* EX8-9b.c */

#include <stdio.h>

/* Declara un arreglo de una sola dimensión */

int elementos[10];
int idx;

main()
{
   for (idx = 0; idx < 10; idx++)
        elementos[idx] = idx ;

   for (idx = 0; idx < 10; idx++)
          printf( "\nelementos[%d] = %d ", idx, elementos[idx] );
}
```

10. La siguiente es una de las muchas respuestas posibles:

```
/* EX8-10.c */
```

```
#include <stdio.h>

/* Declara un arreglo de una sola dimensión */

int elementos[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int nuevo_arreglo[10];
int idx;

main()
{
   for (idx = 0; idx < 10; idx++)
   {
      nuevo_arreglo[idx] = elementos[idx] + 10 ;
   }

   for (idx = 0; idx < 10; idx++)
   {
      printf( "\nelementos[%d] = %d \tnuevo_arreglo[%d] = %d",
              idx, elementos[idx], idx, nuevo_arreglo[idx] );
   }
}
```

Respuestas para el Día 9 "Apuntadores"

Cuestionario

1. El operador dirección_de es el signo &.
2. Se usa el operador indirección *. Cuando se precede el nombre de un apuntador con *, hace referencia a la variable a la que apunta.
3. Un apuntador es una variable que contiene la dirección de otra variable.
4. Indirección es el acto de accesar el contenido de una variable usando un apuntador a la variable.
5. Se guardan en posiciones secuenciales de memoria, con los elementos más bajos del arreglo en las direcciones más bajas.
6. `&datos[0]`

   `datos`

7. Una manera consiste en pasar la longitud del arreglo como un parámetro a la función. La segunda manera es tener un valor especial en el arreglo, como `NULL`, que indique el fin del arreglo.
8. Asignación, indirección, dirección de, incremento, resta y comparación.
9. La resta de dos apuntadores regresa la cantidad de elementos que se encuentran entre ellos. En este caso la respuesta es 1.
10. La respuesta también es 1.

Ejercicios

No se proporcionan respuestas para los ejercicios 9 y 10 debido a que hay muchas posibilidades.

1. Para declarar un apuntador a carácter haga lo siguiente:

```
char *char_ptr;
```

2. Lo siguiente declara un apuntador a `costo` y luego asigna la dirección de `costo (&costo)`, a dicho apuntador.

```
int *p_costo;
p_costo = &costo;
```

3. Acceso directo: `costo = 100;`

   Acceso indirecto: `*p_costo = 100;`

4.
```
printf( "Valor del apuntador: %d, apunta al valor: %d", p_costo,
        *p_costo);
```

5.
```
float *variable = radio;
```

6.
```
datos[2] = 100;
*(datos + 2) = 100;
```

7. (También incluye la respuesta para el ejercicio 8)

```
#include <stdio.h>

#define MAX1 5
#define MAX2 8

int arreglo1[MAX1] = {1, 2, 3, 4, 5 };
int arreglo2[MAX2] = {1, 2, 3, 4, 5, 6, 7, 8 };
int total;
```

```
int totarreglo(int x1[], int len_x1, int x2[], int len_x2);

main()
{
   total = totarreglo(arreglo1, MAX1, arreglo2, MAX2);
   printf( "El total es %d", total);
}

 int totarreglo(int x1[], int len_x1, int x2[], int len_x2)
{

   int total = 0, contador = 0;

    for (contador = 0, contador < len_x1; contador++)
      total += x1[contador];

   for (contador = 0; contador < len_x2; contador++)
      total += x2[contador];

   return total;
}
```

Respuestas para el Día 10 "Caracteres y cadenas"

Cuestionario

1. El rango de los valores del juego de caracteres ASCII va de 0 a 255. De 0 a 127 va el juego de caracteres ASCII estándar, y de 128 a 255 va el juego de caracteres ASCII extendidos.
2. Como el código ASCII del carácter.
3. Una cadena es una secuencia de caracteres que finaliza con por el carácter nulo.
4. Es una secuencia de uno o más caracteres encerrados entre comillas dobles.
5. Para guardar al carácter terminal nulo de la cadena.
6. Se interpreta como una secuencia de valores ASCII, correspondientes con los caracteres que se encuentran entre comillas, seguida de un 0 (el código ASCII para el carácter nulo).

7. a. 97

b. 65

c. 57 (Este es el carácter nueve, y no la cantidad nueve.)

d. 32

e. 206

f. 6

8. a. I

b. un espacio

c. c

d. a

e. n

f. NUL

g.

9. a. 9 bytes. (De hecho la variable es un apuntador a una cadena. La cadena requiere 9 bytes de memoria, 8 para la cadena y 1 para el nulo terminal.)

b. 9 bytes

c. 1 byte

d. 20 bytes

e. 20 bytes

10. a. A

b. A

c. 0 (NUL)

d. Esto está más allá del final de la cadena, por lo que puede tener cualquier valor.

e. !

f. Esto contiene la dirección del primer elemento de la cadena.

Ejercicios

Debido a la cantidad de respuestas posibles, no se proporcionan las respuestas para los ejercicios 5, 6, 7 y 12.

1. `char letra = '$';`
2. `char arreglo[18] = "Los apuntadores son divertidos";`
3. `char *arreglo = "Los apuntadores son divertidos";`
4.
```
char *ptr;
ptr = malloc(81);
gets(ptr);
```
8. `a_string` está declarada como un arreglo de 10 caracteres y, sin embargo, está inicializada con una cadena de más de 10 caracteres. `a_string` necesita ser mayor.
9. Si lo que se trata de hacer con esta línea de código es inicializar una cadena, está equivocado. Se debe usar ya sea `char *quote` o `char quote[100]`.
10. No.
11. Sí. Aunque se puede asignar un apuntador a otro, no se puede asignar un arreglo a otro. Se debe cambiar la asignación por un comando de copia de cadena, como `strcpy()`.

Respuestas para el Día 11 "Estructuras"

Cuestionario

1. Los conceptos de datos de un arreglo deben ser todos del mismo tipo. Una estructura puede contener conceptos de datos de diferentes tipos.
2. El operador de miembro de estructura es un punto. Se usa para accesar miembros de una estructura.
3. `struct`
4. Una etiqueta de estructura está asociada a la plantilla de una estructura y no es una variable real. Una instancia de estructura es una estructura con memoria asignada que puede guardar datos.
5. Los enunciados definen una estructura y declaran una instancia llamada `myaddress`. Luego es inicializada esta instancia. El miembro `myaddress.name`

de la estructura es inicializado a la cadena `"Bradley Jones"`, `myaddress.add1` es inicializado a `"RTSoftware"`, `myaddress.add2` es inicializado a `"P.O. Box 1213"`, `myaddress.city` es inicializado a `"Carmel"`, `myaddress.state` es inicializado a `"IN"`, y, por último, `myaddress.zip` es inicializado a `"46032-1213"`.

6. `ptr++;`
7. Un miembro debe ser un apuntador al propio tipo de la estructura.
8. La primera ventaja tiene que ver con la inserción y borrado de elementos en una lista ordenada. En un arreglo usted tendría que recorrer todos los elementos que ya se encuentran en el mismo para insertar un elemento nuevo. En una lista encadenada, sólo se necesita cambiar el apuntador del elemento anterior al que se está insertando o borrando.

   La segunda ventaja tiene que ver con el tamaño. Un arreglo debe tener declarado su tamaño por anticipado, ocasionando así desperdicio potencial de gran cantidad de espacio. Una lista encadenada asigna espacio conforme lo va necesitando. Si usted usa un arreglo es más probable que el espacio de memoria no sea suficiente.
9. El argumento para `malloc()` es la cantidad de bytes de almacenamiento necesario. Su valor de retorno es un apuntador al primer byte del espacio asignado o, si la llamada falla, `NULL`.
10. Para esto vea la figura 11.9. Si comienza por la parte de la figura con la leyenda "Después" verá que `Arreola` apunta hacia `Baez`, que a su vez apunta a `Cervantes`. Si se quiere borrar a `Baez`, todo lo que se necesita hacer es obtener el elemento que apunta a él (`Arreola`) y hacer que su apuntador (el de `Arreola`) apunte al elemento al que apunta Baez (`Cervantes`).

Ejercicios

1.
```
struct tiempo {
    int horas;
    int minutos;
    int segundos;
};
```

2.
```
struct datos {
    int valor1;
    float valor2;
    float valor3;

} info ;
```

3. `info.valor1 = 100;`

4. ```
   struct datos *ptr;
   ptr = &info;
   ```

5. ```
   ptr->valor2 = 5.5;
   (*ptr).valor2 = 5.5;
   ```

6. ```
   struct datos {
   char nombre[21];
   struct datos *ptr;
   };
   ```

7. ```
   typedef struct {
   char dirección1[31];
   char dirección2[31];
   char ciudad[11];
   char estado[3];
   char código_postal[11];
   } REGISTRO
   ```

8. Lo siguiente usa los valores de la pregunta 5 del cuestionario para la inicialización:

   ```
   REGISTRO midirección = {"RTSoftware",
                           "P.O. Box 1213",
                           "Carmel", "IN", "46032-1213" }
   ```

9. Este fragmento de código tiene dos problemas. El primero es que la estructura debiera tener una etiqueta. El segundo problema es la forma en que es inicializado sign. Los valores de inicialización deben estar entre llaves. El código correcto es

   ```
   struct zodiac {
       char zodiac_sign[21];
       int month;
   } sign = {"Leo", 8};
   ```

10. La declaración `union` tiene un solo problema. Solamente puede ser usada una variable de la unión a la vez. Esto también es cierto para la inicialización de la unión. Solamente puede ser inicializado el primer miembro de la unión. La inicialización correcta es

    ```
    /* setting up a union */
    union data{
        char a_word[4];
        long a_number;
    }generic_variable = { "WOW" };
    ```

11. Si se pretende usar la estructura en una lista encadenada, debe contener un apuntador al siguiente nombre del tipo de estructura. Lo siguiente es el código correcto.

```
/* a structure to be used in a linked list */
struct data{
    char firstname[10];
    char lastname[10];
    char middlename[10];
    struct data *next_name;
};
```

Respuestas para el Día 12 "Alcance de las variables"

Cuestionario

1. El *alcance* de una variable se refiere al alcance que tienen las diferentes partes del programa en su acceso a la variable, es decir, desde donde es visible la variable.
2. Una variable con clase de almacenamiento local es visible solamente en la función en que está definida. Una variable con clase de almacenamiento externo es visible a lo largo de todo el programa.
3. Al definir una variable en una función se le hace local, y al definirla fuera de cualquier función se le hace externa.
4. Automática (por omisión) o estática. Una variable automática es creada cada vez que es llamada la función y destruida cuando la función termina. Una variable local estática persiste y retiene su valor entre llamadas a la función.
5. Una variable automática es inicializada cada vez que es llamada la función. Una variable estática es inicializada solamente la primera vez que es llamada la función.
6. Falso. Cuando se declaran variables de registro se está haciendo una petición. No hay garantía de que el compilador satisfaga la petición.
7. Una variable global sin inicializar es inicializada automáticamente a 0. Sin embargo, es más conveniente inicializar explícitamente a las variables.
8. Una variable local sin inicializar no es inicializada automáticamente y puede contener cualquier cosa. Nunca se debe usar una variable local sin inicializar.

9. Debido a que la variable `count` es ahora local al bloque, el `printf()` ya no tiene acceso a la variable llamada `count`. El compilador indica un error.

10. Si el valor necesita ser recordado debe ser static. Si la variable fuera llamada `vari`, la declaración debiera ser:

```
static int vari;
```

11. La palabra clave `extern` se usa como modificador de clase de almacenamiento. Indica que la variable ha sido declarada en cualquier otro lugar del programa.

12. La palabra clave `static` se usa como modificador de clase de almacenamiento. Le dice al compilador que guarde el valor de la variable o función mientras dure el programa. Dentro de una función la variable guarda su valor entre las llamadas a la función.

Ejercicios

1.
```
register int x = 0;
```

2.
```
/* Ilustra el alcance de variables. */
#include <stdio.h>

void print_value(int x);

main()
{
    int x = 999;

    printf("%d", x);
    print_value ( int x )
}

void print_value( int x)
{
    printf("%d", x);
}
```

3. Debido a que se está declarando a `var` como global, no se necesita pasarla como parámetro.

```
/* Ejercicio 12.3 - Uso de una variable global */
#include <stdio.h>
```

```
int var = 99;

void print_value(void);

main()
{
   print_value();
}

void print_value(void)
{
   printf( "El valor es %d", var );
}
```

4. Sí, se necesita pasar la variable `var` para imprimirla en una función diferente.

```
/* Ejercicio 12.4 - Uso de una variable local */
#include <stdio.h>

void print_value(int var);

main()
{   int var = 99
   print_value( var );}

void print_value(int var)
{
   printf( "El valor es %d", var ):
}
```

5. Sí, un programa puede tener una variable local y otra global con el mismo nombre. En estos casos la variable local activa tiene precedencia.

```
/* Ejercicio 12.5 - Uso de una global */
#include <stdio.h>

int var = 99:

void print_func(void);

main()
{
```

```
    int var = 77:
   printf( "Impresión en función con local y global :);
   printf( "\nEl valor de var es %d", var );
   print_func( );
}
void print_func( void )
{
   printf( "\nImpresión en función solamente con global:");
   printf( "\nEl valor de var es %d", var );}
```

6. Sólo hay un problema con `a_sample_function()`. Las variables pueden ser declaradas al inicio de cualquier bloque. Por lo tanto las declaraciones `ctr1` y `star` son correctas. La otra variable, `ctr2`, no es declarada al inicio del bloque, por lo que necesita ser declarada. La siguiente es la función corregida dentro de un programa completo:

```
/* Ejercicio 12.6 */     #include <stdio.h>

void_a_sample_function( );
main()
{
   a_sample_function();
   return 0;
}

void a_sample_function( void )
{
   int ctr1;

   for ( ctr1 = 0; ctr1 < 25; ctr1++ )
      printf( "*" );

   puts( "\nEste es un ejemplo de función" );
{
   char star = '*';
   int ctr2;                    /* corrección */
   puts( "Ha habido un problema" );
   for ( ctr2 = 0; ctr2 < 25; ctr2++ )
   {
       printf( "%c", star);
```

```
        }
    }
}
```

7. Este programa no tiene errores, pero podría estar mejor. Realmente no hay razón para declarar a la variable `tally` como `static`. Una variable `static` en `main()` es equivalente a una variable local en `main()`.

8. ¿Cuál es el valor de `star`? ¿Cuál es el valor de `dash`? Estas dos variables nunca han sido inicializadas. Debido a que ambas son variables locales pueden contener cualquier valor. Observe que este programa compila sin errores ni avisos, pero sí hay un problema.

   Hay un segundo punto que debe ser comentado acerca de este programa. La variable `ctr` es declarada como global, pero sólo se usa en `print_function()`. Esto no es una buena asignación. El programa estaría mejor si `ctr` fuera una variable local en la función `print_function()`.

9. Este programa imprime el siguiente patrón por siempre. Vea el ejercicio 10.

   ```
   X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==...
   ```

10. Este programa plantea un problema debido al alcance global de `ctr`. Tanto la función `main()` como `print_letter2()` usan a `ctr` en ciclos al mismo tiempo. Debido a que `print_letter2()` cambia el valor, el ciclo `for` de `main()` nunca termina. Esto puede ser corregido en diferentes maneras. Una de ellas es usar dos variables de contador diferentes. Una segunda manera es cambiar el alcance de la variable de contador, `ctr`. Puede ser declarada tanto en `main()` como en `print_letter2()` como variable local.

    Un comentario adicional sobre `letter1` y `letter2`. Debido a que cada una de ellas se usan solamente en una función, debieran ser declaradas como locales. A continuación se presenta el listado corregido:

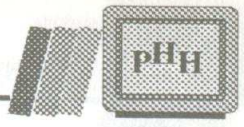
```
#include <stdio.h>
void print_letter2(void);              /* prototipo de función */

main()
{
   char letter1 = 'X';
   int ctr;

   for( ctr = 0; ctr < 10; ctr++ )
{
```

```
            printf( "%c", letter1 )
            print_letter2();
        }
    }

    void print_letter2(void)
    {
        char letter2 = '=';
        int ctr;                    /* ésta es una variable local */
                                    /* y es diferente de ctr en main() */
        for( ctr = 0; ctr < 2; ctr++ )
            printf( "%c", letter2 );
    }
```

Respuestas para el Día 13 "Más sobre el control del programa"

Cuestionario

1. Nunca. (A menos que sea *muy* cuidadoso.)
2. Cuando se llega a un enunciado `break` la ejecución sale inmediatamente del ciclo `for`, `do...while` o `while` que contiene el `break`. Cuando se llega a un `continue` se inicia inmediatamente la siguiente iteración del ciclo que lo contiene.
3. Un ciclo infinito es un ciclo que ejecuta para siempre. Se crea escribiendo un ciclo `for`, `do...while` o `while` con una condición que siempre es cierta.
4. La ejecución termina cuando el programa llega al final de `main()` o es llamada la función `exit()`.
5. La expresión en un enunciado `switch` puede evaluar a un valor `long`, `int` o `char`.



6. El enunciado `default` es un caso en un enunciado `switch`. Cuando la expresión del enunciado `switch` evalúa a un valor para el cual no hay caso, el control pasa al caso `default`.
7. La función `atexit()` registra funciones que serán ejecutadas cuando el programa termine.
8. La función `system()` ejecuta un comando a nivel del sistema operativo.

Ejercicios

1. `continue;`
2. `break;`
3.
```
atexit(f3);
atexit(f2);
atexit(f1);
```
4. `system("dir");`
5. Este fragmento de código es correcto. No se necesita un enunciado `break` después del `printf()` para `'N'`, debido a que de todas formas termina el enunciado.
6. Tal vez piense que `default` debe ir hasta el final del enunciado `switch` y esto no es cierto. El `default` puede ir en cualquier lugar. Sin embargo, hay un problema. Debe haber un enunciado `break` al final del caso `default`.
7.
```
if( choice == 1 )
   printf("You answered 1");
else if( choice == 2 )
      printf( "You answered 2");
    else
      printf( "You did not choose 1 or 2");
```
8.
```
do {
/* cualquier enunciado del C */
} while ( 1 );
```

Respuestas para el Día 14 "Trabajando con la pantalla, la impresora y el teclado"

Cuestionario

1. Un flujo es una secuencia de caracteres. Un programa en C usa flujos para toda la entrada y la salida.

2. a. Una impresora es un dispositivo de salida.

   b. Un teclado es un dispositivo de entrada.

   c. Un modem es un dispositivo tanto de entrada como de salida.

   d. Un monitor es un dispositivo de salida. (Aunque una pantalla sensible al tacto podría ser un dispositivo de entrada y salida.)

   e. Una unidad de disco puede ser un dispositivo tanto de entrada como de salida.

3. Los cinco flujos predefinidos son: `stdin` (el teclado), `stdout` (la pantalla), `stderr` (la pantalla), `stdprn` (la impresora) y `stdaux` (el puerto serie COM1:).

4. a. `printf()` y `puts()` usan el flujo `stdout`.

   b. `scanf()` y `gets()` usan el flujo `stdin`.

   c. a `fprintf()` le puede ser pasado cualquier flujo de salida. De los cinco flujos estándar puede usar a `stdout`, `stderr`, `stdprn` y `stdaux`.

5. La entrada con almacenamiento temporal es enviada al programa solamente hasta que el usuario oprime Enter. La entrada sin almacenamiento temporal es enviada un carácter a la vez, tan pronto como cada tecla es oprimida.

6. La entrada replicada envía automáticamente cada carácter a `stdout` en cuanto lo recibe, y la entrada sin replicar no lo hace.

7. Sólo se puede "desobtener" un carácter entre lecturas. El carácter `EOF` no puede ser regresado al flujo de entrada con `unget()`.

8. Con el carácter de nueva línea que se genera cuando el usuario oprime Enter.

9. a. Válido.

   b. Válido.

c. Válido.

d. Inválido. No hay identificador que sea `q`.

e. Válido.

f. Válido.

10. `stderr` no puede ser redireccionado, ya que siempre imprime a la pantalla. `stdout` puede ser redireccionado a algún otro flujo diferente a la pantalla.

Ejercicios

1. `printf( "Hello World" );`

2. ```
   fprintf( stdout, "Hello World" );
     puts( "Hello World");
   ```

3. `fprintf( stdaux, "Hello Auxiliary Port" );`

4. ```
   char buffer[31];
   scanf( "%30[^*]", buffer );
   ```

5. ```
   printf( "Juan preguntó \"¿Qué es una diagonal
   invertida\?\"\nPedro\ dijo, \"Es \'\\\'\"");
   ```

7. Pista: Use un arreglo de 26 enteros. Para llevar la cuenta de cada carácter incremente al elemento adecuado del arreglo para cada lectura de carácter.

10. Pista: Obtenga una cadena a la vez, y luego imprima un número de línea formateado seguido de un tabulador y de la cadena.

Respuestas para el Día 15 "Más sobre apuntadores"

Cuestionario

1. ```
   float x;
   float *px = &x;
   float **ppx = &px;
   ```

2. El error es que el enunciado usa un operador de indirección simple y, como resultado, asigna el valor de 100 a `px` en vez de hacerlo a `x`. El enunciado debiera ser escrito con un operador de doble indirección.

   ```
   **ppx = 100;
   ```

3. `arreglo` es un arreglo con dos elementos. Cada uno de esos elementos es a su vez un arreglo que contiene tres elementos. Cada uno de esos tres elementos es un arreglo que contiene cuatro variables tipo `int`.
4. `arreglo[0][0]` es un apuntador al primer arreglo de cuatro elementos `int`.
5. Las comparaciones primera y tercera son ciertas, y la segunda no lo es.
6. `void func1(char *p[]);`
7. No hay manera de saberlo. Este valor se le debe pasar a la función como otro argumento.
8. Un apuntador a función es una variable que guarda la dirección de memoria en donde se encuentra guardada la función.
9. `char (*ptr)(char *x[]);`
10. Si se omiten los paréntesis que rodean a `*ptr`, la línea es el prototipo de una función que regresa un apuntador a tipo `char`.

Ejercicios

1. a. `var1` es un apuntador a un entero.
   b. `var2` es un entero.
   c. `var3` es un apuntador a un apuntador a un entero.
2. a. `a` es un arreglo de 36 (3 x 12) enteros.
   b. `b` es un apuntador a un arreglo de 12 enteros.
   c. `c` es un arreglo de 12 apuntadores a enteros.
3. a. `z` es un arreglo de 10 apuntadores a caracteres.
   b. `y` es una función que toma un entero (campo) como argumento y regresa un apuntador a carácter.
   c. `x` es un apuntador a función que toma un entero (campo) como argumento y regresa un carácter.
4. `float (*func)(int campo);`
5. Un arreglo de apuntadores de función puede ser usado junto con un sistema de menús. El número de menú seleccionado puede relacionarse con el índice del arreglo para el apuntador de función. Por ejemplo, la función apuntada por el

quinto elemento del arreglo sería ejecutada si se seleccionara el concepto 5 del menú.

```
int (*opción_menu[10])(char *título);
```

6. `char *ptrs[10];`

7. `ptr` fue declarado como un arreglo de 12 apuntadores a enteros y no como un apuntador a un arreglo de 12 enteros. El código correcto debiera ser

```
int x[3][12];
int (*ptr)[12];

ptr = x;
```

Respuestas para el Día 16 "Uso de archivos de disco"

Cuestionario

1. Un flujo de modo-texto ejecuta automáticamente la traducción entre el carácter de nueva línea, `\n`, que usa el C para indicar el fin de una línea y el par de caracteres-retorno de carro avance de línea, que usa el DOS para indicar el fin de una línea. Por el contrario, un flujo de modo binario no ejecuta traducciones. Todos los caracteres son recibidos y enviados sin modificación.
2. Debe abrir el archivo usando la función de biblioteca `fopen()`.
3. Cuando se usa `fopen()` se debe especificar el nombre del archivo de disco a abrir y el modo en que se abre. La función `fopen()` regresa un apuntador a tipo `FILE`. Este apuntador se usa en las funciones subsecuentes de acceso al archivo para hacer referencia a este archivo específico.
4. Formateado, de carácter y directo.
5. Secuencial y al azar.
6. `EOF` es el indicador de fin de archivo. Es una constante simbólica igual a -1.
7. Se usa `EOF` con los archivos de texto para determinar si se ha llegado al fin de archivo.
8. En modo binario se debe usar la función `feof()`. En modo texto se puede revisar la presencia del carácter `EOF` o usar `feof()`.

9. El indicador de posición de archivo indica la posición en un archivo dado donde sucederá la siguiente operación de lectura o escritura. Se puede modificar al indicador de posición de archivo con `rewind()` y `fseek()`.
10. El indicador de posición de archivo apunta al primer carácter del archivo, es decir, desplazamiento 0.

Ejercicios

1. `fcloseall();`
2. `rewind(fp);` y `fseek(fp, 0, SEEK_SET);`
3. No se puede usar la revisión de `EOF` con un archivo binario. En vez de ello se debe usar la función `feof()`.

Respuestas para el Día 17 "Manipulación de cadenas"

Cuestionario

1. La longitud de una cadena es la cantidad de caracteres entre el inicio de la cadena y el carácter nulo terminal (sin contar al carácter nulo). Se puede determinar la longitud de la cadena con la función `strlen()`.
2. Debe asegurarse de asignar suficiente espacio de almacenamiento para la nueva cadena.
3. Concatenar significa unir dos cadenas, añadiendo una cadena al final de la otra.
4. Cuando se comparan cadenas, "mayor que" significa que los valores ASCII de una cadena son mayores que los valores ASCII de la otra cadena.
5. `strcmp()` compara dos cadenas completas. `strncpy()` sólo compara la cantidad especificada de caracteres de la cadena.
6. `strcmp()` compara dos cadenas tomando en consideración mayúsculas y minúsculas. (`'A'` y `'a'` son diferentes.) `strcmpi()` ignora a las mayúsculas y minúsculas. (`'A'` y `'a'` son lo mismo.)
7. `isascii()` revisa los valores que se le pasan para ver si son caracteres estándar ASCII entre 0 y 127. No revisa los caracteres ASCII extendidos.

8. Tanto `isascii()` como `iscntrl()` regresan `TRUE` y todas las demás regresan `FALSE`. Recuerde que estas macros revisan el valor de carácter.

9. 65 es equivalente al carácter ASCII `'A'`. Las siguientes macros regresan `TRUE`: `isalnum()`, `isalpha()`, `isascii()`, `isgraph()`, `isprint()` e `isupper()`.

10. Las funciones para revisión de caracteres determinan si un carácter en particular satisface determinada condición, como si es una letra, un signo de puntuación o alguna otra cosa.

Ejercicios

1. `TRUE` (1) o `FALSE` (0)
2. a. 65
   b. 81
   c. -34
   d. 0
   e. 12
   f. 0
3. a. 65.000000
   b. 81.230000
   c. -34.200000
   d. 0.000000
   e. 12.000000
   f. 1000.000000
4. A `cadena2` no le fue asignado espacio antes de ser usada. No hay manera de saber el lugar a donde `strcpy()` copió el valor de `cadena1`.

Respuestas para el Día 18 "Obteniendo más de las funciones"

Cuestionario

1. El pasarlos por valor significa que la función recibe una copia del valor de la variable del argumento. El pasarlos por referencia significa que la función recibe la dirección de la variable del argumento. La diferencia es que al pasarlos por referencia, se le permite a la función modificar el valor original, lo que no sucede cuando se les pasa por valor.
2. Un apuntador a tipo `void` es un apuntador que puede apuntar a cualquier tipo de objeto de dato del C (un apuntador genérico).
3. Mediante el uso de un apuntador `void`, se puede crear un apuntador "genérico" que puede apuntar a cualquier objeto. El uso más común de un apuntador `void` se da en la declaración de parámetros de función. Se puede crear una función que puede manejar argumentos de tipos diferentes.
4. La especificación de tipo proporciona información acerca del tipo del objeto de dato a que el apuntador `void` está apuntando en ese momento. Se debe dar la especificación de tipo para el apuntador `void` antes de desreferenciarlo.
5. A una función que toma una lista variable de argumentos se le debe pasar por lo menos un argumento fijo. Esto se hace para decirle a la función la cantidad de argumentos que le son pasados cada vez que se le llama.
6. `va_start()` debe ser usada para inicializar la lista de argumentos. `va_arg()` debe ser usada para recuperar los argumentos. `va_end()` debe ser usada para hacer la limpieza una vez que se han recuperado todos los argumentos.
7. ¡Vaya pregunta capciosa! Los apuntadores `void` no pueden ser incrementados, debido a que el compilador no sabe qué valor debe sumar.
8. Sí, una función puede regresar un apuntador a cualquiera de los tipos de variable del C. Una función también puede regresar un apuntador a áreas de almacenamiento como arreglos, estructuras y uniones.

Ejercicios

1. `int función( char arreglo[] );`
2. `int números( int *núm1, int *núm2, int *núm3);`

3. `int núm1 = 1, núm2 = 2, núm3 = 3;`
   `números( &núm1, &núm2, &núm3);`
4. Aunque el código puede parecer algo confuso, es perfectamente correcto. Esta función toma el valor apuntado por `núm` y lo multiplica por sí mismo.
5. Cuando se usan listas variables de parámetros se deben usar todas las herramientas de macros. Esto incluye a `va_list`, `va_start()`, `va_arg()` y `va_end()`. Vea el listado 18.3 sobre la manera correcta de usar listas variables de parámetros.

Respuestas para el Día 19 "Exploración de la biblioteca de funciones"

Cuestionario

1. Tipo `double`.
2. En la mayoría de los compiladores es equivalente a `long`, mas, sin embargo, esto no se garantiza. Revise el archivo TIME.H del compilador de que disponga, o el manual de consulta, para saber qué tipo de variable usa el compilador.
3. La función `time()` regresa la cantidad de segundos que han transcurrido desde la medianoche del primero de enero de 1970. La función `clock()` regresa la cantidad de 1/100 de segundo que ha transcurrido desde que el programa inició su ejecución.
4. No hace nada. Simplemente despliega un mensaje que describe el error.
5. Hay que ordenar el arreglo en forma ascendente.
6. 14
7. 4
8. 21
9. `0` si los valores son iguales.

   `>0` si el valor del elemento 1 es mayor que el del elemento 2.

   `<0` si el valor del elemento 1 es menor que el del elemento 2.
10. NULL

Ejercicios

1.
```
bsearch( minombre, nombres (sizeof(nombres)/sizeof(nombres[0])),
        sizeof(nombres[0]), comp_nom);
```

2. Hay tres problemas. El primero es que no se proporciona el ancho de campo en la llamada a `qsort()`. El segundo es que no se deben añadir paréntesis al final de nombre de función en la llamada a `qsort()`. El tercero es que al programa le falta la función de comparación. `qsort()` usa `función_de_comparación()` y no está definida en el programa.

3. La función de comparación regresa los valores en forma errónea. Debe regresar un número positivo si `elemento1` es mayor que `elemento2` y un número negativo si `elemento1` es menor que `elemento2`.

Respuestas para el Día 20 "Otras funciones"

Cuestionario

1. La función `malloc()` asigna una cantidad específica de bytes de memoria, en tanto que `calloc()` asigna la cantidad suficiente de memoria para una cantidad específica de objetos de dato de un determinado tamaño. `calloc()` también pone a 0 los bytes de la memoria.

2. Para conservar la parte fraccionaria de la respuesta cuando se divide un entero entre otro y se asigna el resultado a una variable de punto flotante.

3. a. `long`

   b. `int`

   c. `char`

   d. `float`

   c. `float`

4. La memoria asignada dinámicamente es memoria que puede ser asignada al momento de la ejecución. Al asignar memoria dinámicamente se asigna la memoria solamente cuando se le necesita.

5. A una cadena que contiene el nombre del programa actual incluyendo la información de ruta.

6. Definiendo un miembro de campo de bits con un tamaño de tres bits.

7. Dos bytes. Usando campos de bits se puede declarar una estructura de la manera siguiente:

```
struct fecha{
   unsigned mes : 4;
   unsigned día : 5;
   unsigned año : 7;
}
```

Esta estructura guarda a la fecha en dos bytes (16 bits). El mes puede ir de 0 a 15, el día puede ir de 0 a 31 y el año de 0 a 127. Cuando se le añade a 1900, el año puede ir de 1900 a 2027.

8. 00100000
9. 00001001
10. Las dos expresiones evalúan al mismo resultado. El usar OR exclusivo con 11111111 es virtualmente lo mismo que usar el operador de complemento.

Ejercicios

1.
```
long *ptr;
ptr = malloc( 1000 * sizeof(long));
```

2.
```
long *ptr;
ptr - calloc( 1000, sizeof(long));
```

3. No. Cuando se usan campos de bits se les debe de poner en la primera posición dentro de una estructura. Lo siguiente es correcto.

```
struct respuestas_cuestionario {
    unsigned respuesta1   : 1;
    unsigned respuesta2   : 1;
    unsigned respuesta3   : 1;
    unsigned respuesta4   : 1;
    unsigned respuesta5   : 1;
    char nombre_estudiante[15];
}
```

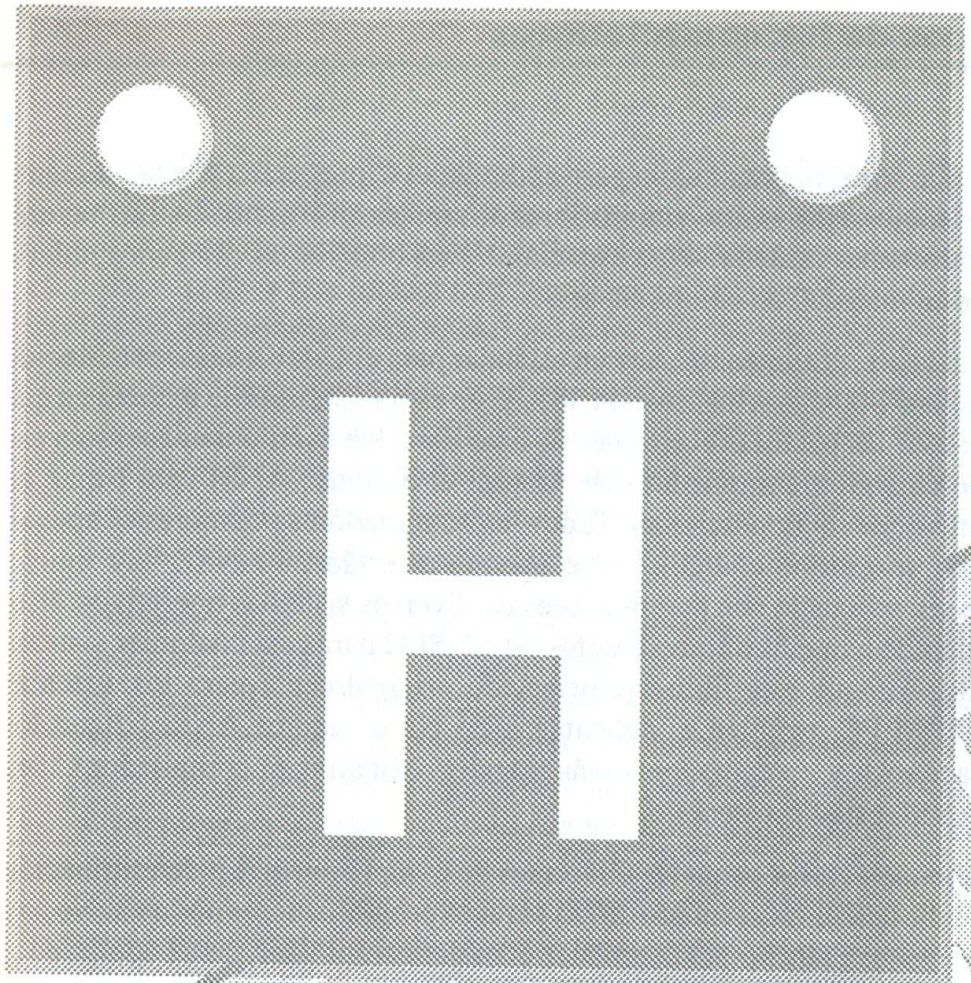
4. No hay error alguno en el ejemplo dado, sin embargo, el resultado podría no ser el idóneo. Debido a que número1 y número2 son enteros, el resultado de su división será un entero, perdiéndose, por lo tanto, la parte fraccionaria. Para tener la parte fraccionaria en la respuesta se necesita dar una especificación de tipo, para que la expresión sea de tipo float.

```
respuesta = (float) número1 / número2;
```

Respuestas para el Día 21 "Aprovechando las directivas del preprocesador y más"

Cuestionario

1. La programación modular se refiere al método de desarrollo de programas que divide a un programa en varios archivos de código fuente.
2. El módulo principal contiene la función `main()`.
3. Para evitar efectos indeseados, asegurándose que las expresiones complejas que se pasan como argumentos a la macro sean primero evaluadas por completo.
4. En comparación a una función, una macro da como resultado una ejecución más rápida del programa, pero a costa de un mayor tamaño de programa.
5. El operador `defined()` revisa si está definido un nombre en particular, regresando `CIERTO` si el nombre está definido y `FALSO` en caso contrario.
6. Usted debe usar `#endif`.
7. Los archivos fuente compilados se convierten en archivos objeto con una extensión .OBJ.
8. La directiva `#include` copia el contenido de un archivo al archivo actual.
9. Un enunciado `#include` con comillas dobles busca en el directorio actual el archivo a incluirse. Un enunciado `#include` con <> busca en el directorio estándar el archivo a incluirse.
10. `__DATE__` se usa para poner en el programa la fecha en que el programa se compiló.



H

Puntos específicos de los compiladores



Puntos específicos de los compiladores

Los programadores que se están preparando para aprender el C preguntan frecuentemente cuál compilador es el mejor. Hay varios compiladores diferentes en el mercado. La mayoría de ellos hacen gala de muchas características especiales. Esta obra fue escrita para ayudarle a seleccionar entre esta multitud de compiladores.

Tal como se dijo en el Día 1, "Comienzo", hay un estándar para el C que ha sido establecido por el American National Standards Institute (ANSI). Este estándar define la manera en que un compilador debe usar las palabras claves del C. Este libro fue escrito de acuerdo a ese estándar, por lo que cualquier compilador que soporte el estándar ANSI será capaz de compilar los programas que se proporcionan. Todos los compiladores populares se apegan al estándar ANSI. Si se escribe código que se apegue al estándar ANSI, hay mucha probabilidad de que sea utilizable con compiladores de diversos sistemas operativos. Esta probabilidad se da debido a que hay compiladores de ANSI C para casi cualquier sistema operativo importante. Aunque este libro fue orientado originalmente para los sistemas compatibles con la IBM PC, también es aplicable a los compiladores ANSI C para los sistemas UNIX, Macintosh y computadoras más grandes. Mientras el compilador C sea compatible con el ANSI, este libro será útil.

Muchas compañías venden varios paquetes diferentes de compiladores. La determinación de cuál compilador debe usarse es a menudo difícil para los programadores novatos. No hay día en que alguien pregunte cuál compañía produce el mejor compilador, o qué compilador debe ser usado para programar en C. Todo mundo tiene prejuicios que afectan a sus opiniones acerca de los diversos compiladores. La determinación de cuál compilador es el mejor depende de las opiniones propias. La mayoría de los compiladores de C disponibles soportan el estándar ANSI. Muchos de ellos también soportan C++, la programación de Windows, la programación a 32 bits y mucho más. Por lo general, cuanto más características tengan mayor es el costo. Las siguientes son recomendaciones a seguir cuando se compre un compilador de C:

- ❑ Asegúrese de que soporta el estándar ANSI.
- ❑ Si usted cree que va a necesitar soporte (ayuda) para el compilador, cómprelo a una compañía grande de buena reputación. Es más probable que las compañías grandes proporcionen soporte. Además, es más probable que proporcionen actualizaciones a nuevos estándares y nuevas características. Compañías tales como Borland, Microsoft y Symantec son compañías grandes e importantes dentro del mercado del C/C++.
- ❑ Determine qué tipo de programas quiere escribir. Muchas compañías sólo trabajan en determinadas plataformas. El ThinkC de la corporación Symantec funciona en la Macintosh. El Visual C/C++ Edición Estándar de Microsoft no crea aplicaciones para el DOS. El Turbo C++ para DOS de Borland no crea programas para Windows.

- Determine si va a querer avanzar al C++ en el futuro. La mayoría de los compiladores del C vienen ahora acompañados de compiladores C++. De hecho, actualmente la mayoría de los compiladores son anunciados como compiladores de C++ que incluyen C. Para obtener un compilador C tal vez tenga que comprar un compilador de C++.

Borland International Inc. y Microsoft Corporation son los dos mayores fabricantes de compiladores C/C++. Ambas compañías proporcionan varios paquetes diferentes de compilador. Típicamente las compañías venden versiones profesionales y no profesionales de sus compiladores. Las versiones estándar, o no profesionales, son, por lo general, más baratas; sin embargo, no tienen todas las características.

Borland vende varios paquetes de compilador. La versión profesional de su compilador incluye el nombre de la compañía, Borland C++. Las versiones estándar son etiquetadas con el nombre Turbo. Hay varios paquetes Turbo. Turbo C++ para Windows ejecuta bajo Windows. Puede crear aplicaciones de Windows pero no aplicaciones de DOS. Turbo C++ para DOS es una aplicación de DOS, que puede crear programas para el DOS pero no puede crear programas para Windows. Ambos contienen típicamente compiladores de C junto con el de C++. El paquete profesional, Borland C++, puede crear programas tanto para el DOS como para Windows. El paquete profesional es, de hecho, el mismo compilador que el compilador Turbo, combinado con una cantidad de características adicionales.

Microsoft también tiene varios paquetes de compilador. La línea principal de compiladores de Microsoft es el Visual C/C++. Este viene en Ediciones Estándar y Profesional, que requieren, ambas, a Windows para operar. En la misma forma que los compiladores de Borland, la principal diferencia es que la edición profesional contiene todo lo de la edición estándar y otras cosas adicionales. La edición estándar permite la creación de aplicaciones de Windows y de QuickWin. La edición profesional añade la capacidad de crear aplicaciones de DOS. Microsoft también tiene un compilador de nivel más bajo que no requiere Windows. El compilador QuickC estaba disponible al momento en que este libro se escribió; sin embargo, su futuro es desconocido. Este es un compilador de bajo nivel para crear solamente aplicaciones de DOS.



Nota: La siguiente es información que puede ser de ayuda cuando se instale la edición estándar del Visual C/C++ de Microsoft o el Turbo C/C++ para DOS de Borland. La información también puede ser relevante para otros compiladores.

Hay dos razones para la selección de estos compiladores. La primera es que uno de éstos trabaja bajo Windows y el otro trabaja bajo el DOS. Si está instalando un compilador diferente basado en Windows, es probable que tenga opciones

similares a las que se presentan en las secciones para la instalación del Visual C/C++ de Microsoft. Si está instalando un compilador diferente basado en el DOS, es probable que sea similar al compilador Turbo C/C++ para el DOS de Borland.

La segunda razón de la selección de estos compiladores en particular como ejemplo, se debe a su popularidad. Los compiladores de C/C++ de Borland y Microsoft son actualmente los más populares en el mercado. Tome en cuenta que la selección de estos dos compiladores no debe ser considerada como una recomendación de los autores.

Instalación de la edición estándar del Visual C/C++ de Microsoft

El Visual C/C++ de Microsoft es fácil de instalar. Debido a que el Visual C/C++ de Microsoft es un paquete de Windows, se debe ejecutar Windows antes de comenzar el proceso de instalación. Los manuales que vienen con el compilador lo llevan de la mano en la instalación. El Visual C/C++ se comporta como si contuviera varios compiladores. En realidad contiene un compilador que puede crear varios archivos finales diferentes. En la edición estándar hay tres diferentes tipos de archivos finales, uno para crear aplicaciones de Windows, otro para crear bibliotecas de enlace dinámico de Windows (DLL) y otro más para crear ejecutables de QuickWin. La edición profesional también incluye un cuarto tipo para crear aplicaciones para el DOS. Si está usando la edición estándar, tal vez quiera usar el compilador QuickWin para los ejemplos de este libro. Si está usando la edición profesional, tal vez quiera usar el compilador para QuickWin o para DOS.

El compilar aplicaciones para QuickWin es similar a la compilación de aplicaciones para el DOS. Las aplicaciones de QuickWin son programas escritos para DOS que están compilados para que trabajen como programas de Windows. Los programas creados así ejecutarán solamente bajo Windows. QuickWin se ve y trabaja exactamente como lo haría una aplicación de DOS, a excepción de que lo hace en una ventana genérica de Windows. Tenga siempre presente que con la edición estándar del Visual C/C++ de Microsoft no podrá crear programas que ejecuten bajo el DOS. Debido a esto necesitará usar el compilador QuickWin.

Instalación de lo mínimo

Los compiladores Visual C/C++ son grandes. La edición estándar, que es la más pequeña de las dos, ocupa algo más de 30 megabytes de espacio de disco duro. Si su intención inicial es solamente aprender a programar en C, no es necesaria la mayor parte de lo que se instala con el Visual C/C++. Esta obra no trata la programación de ejecutables o DLL de Windows ni trata al C++. El Visual C/C++ instala una gran cantidad de programas para esto. Mientras esté aprendiendo C no hay necesidad de tener estos "extras" ocupando espacio de disco.

Durante la instalación del Visual C/C++ verá un cuadro de diálogo parecido al que se muestra en la figura H.1.

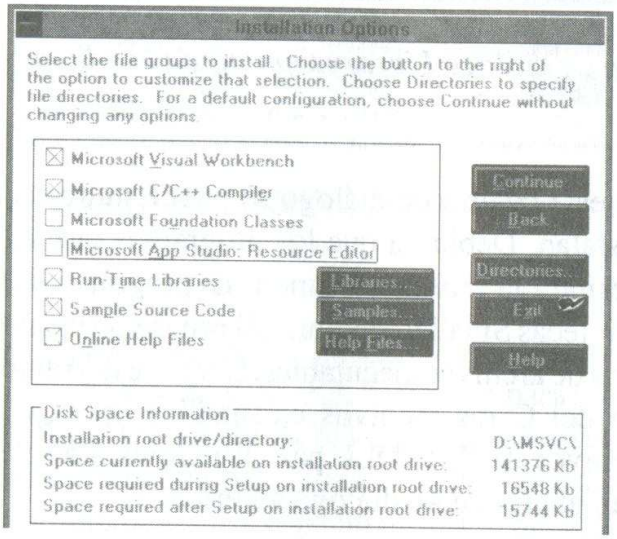


Figura H.1. *Cuadro de diálogo Installation Options de la edición estándar del Visual C/C++ de Microsoft.*

Este es considerado el cuadro de diálogo principal, debido a que determina lo que será instalado o no. Si decide no instalar una opción en este momento, podrá volver a ejecutar el programa de instalación posteriormente e instalar la opción.

Tal como se muestra en la figura H.1, hay dos opciones que no son necesarias. Son las de Microsoft Foundation Classes y Microsoft App Studio: Resource Editor. Las clases fundamentales son usadas con C++, y el Editor de Recursos de APP Studio es usado para la programación de aplicaciones de Windows. Se debe quitar la selección de estas opciones.

El Microsoft Visual Workbench, que se menciona al principio en el cuadro de diálogo, es un ambiente integrado en donde se puede teclear, corregir y ejecutar programas desde un área central. Si usted tiene su propio editor para teclear programas, no lo necesitará. Sin embargo, muchos programadores prefieren instalarlo y usar el Workbench.

El cuadro de diálogo principal contiene varios botones. Cuando se selecciona el botón Libraries se presenta el cuadro de diálogo que se muestra en la figura H.2.

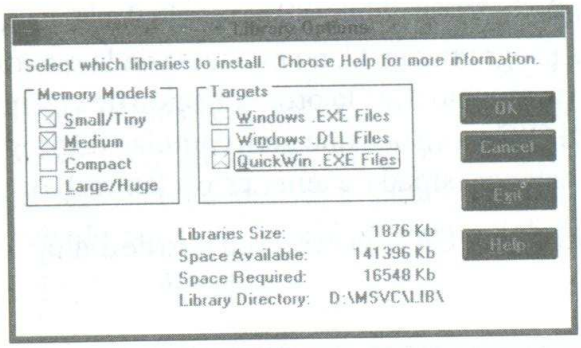


Figura H.2. *Cuadro de diálogo Library Options de la edición estándar del Visual C/C++ de Microsoft.*

Mediante el uso de este cuadro de diálogo se puede indicar qué biblioteca se crea y qué compiladores se instalan. Debido a que los programas que se presentan en este libro son pequeños sólo necesitará la biblioteca Small. El programa de instalación tiene asignadas por omisión las bibliotecas Small y Medium. El programa de instalación también indica, por omisión, los tres tipos de archivos ejecutables. Como se dijo anteriormente, sólo se necesitan para el aprendizaje del C los archivos ejecutables tipo QuickWin. Se puede quitar la selección de los archivos .EXE y .DLL para Windows. La selección de OK, o la opresión de Enter, lo regresan al cuadro de diálogo anterior.

Con la opción Samples se determina cuáles programas de ejemplo son instalados. Si no se quiere ningún programa de ejemplo, seleccione la opción Sample Source Code en el cuadro de diálogo principal (vea la figura H.1). Al quitar la selección de esta opción se protege al botón Sample. Cuando se selecciona el botón Sample se despliega el cuadro de diálogo que se muestra en la figura H.3.

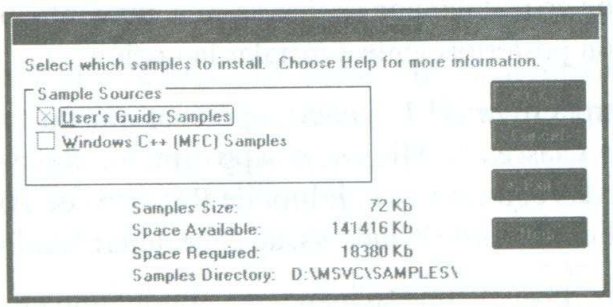


Figura H.3. *Cuadro de diálogo Source Options de la edición estándar del Visual C/C++ de Microsoft.*

Los ejemplos de Windows C++ (MFC) no son necesarios, debido a que este libro solamente trata el C. Si va a usar la guía de usuario que viene con el manual, instale los ejemplos User

guide. Incluso, aunque no vaya a usar la guía de usuario, tal vez quiera instalar el código fuente de ejemplo. Los programas de ejemplo de la guía de usuario ocupan poco espacio de disco. Seleccione OK, u oprima Enter, para regresar al cuadro de diálogo anterior.

Si cree que no va a tener problemas no necesita instalar los archivos de ayuda. Si usted es como la mayoría de la gente, los archivos de ayuda le serán muy importantes. Cuando seleccione el botón Help Files verá el cuadro de diálogo que se muestra en la figura H.4.

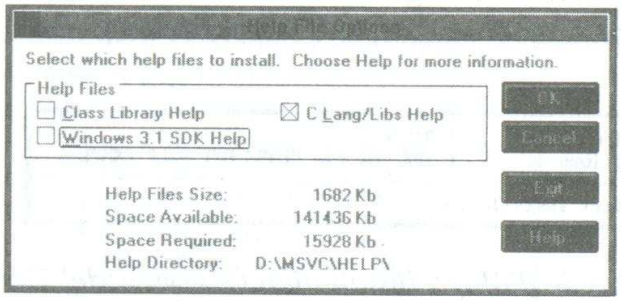


Figura H.4. *Cuadro de diálogo Help File Options de la edición estándar del Visual C/C++ de Microsoft.*

Este cuadro de diálogo le proporciona tres opciones. Los archivos de Class Library Help son para el C++. Los archivos del Windows SDK3.1 Help son para la programación en Windows. Solamente los archivos de C Lang/Libs Help le serán útiles en relación a su *aprendizaje del C en 21 días*. Aunque estos archivos de ayuda no son indispensables le conviene instalarlos, a menos de que tenga problemas con el espacio de disco. Seleccione el botón OK, u oprima Enter, para regresar al cuadro de diálogo principal.

Estas son todas las opciones modificables por usted que pueden afectar a la cantidad de espacio de disco usada. Conforme selecciona o no las opciones, las cifras que se encuentran en la parte inferior de los cuadros de diálogo le informarán el total de la cantidad necesaria de espacio de disco. Si la unidad de disco actual no tiene suficiente espacio, se le puede cambiar. Use el botón Directories para cambiar a una unidad diferente. Si la unidad actual tiene suficiente espacio, se puede oprimir Continue para completar la instalación.

Instalación del Turbo C/C++ para DOS de Borland

La instalación de un compilador basado en el DOS es relativamente simple. Parte de su simplicidad le viene del hecho de que la mayoría de los compiladores proporcionan opciones por omisión. Los compiladores de Borland no son diferentes. La instalación más fácil es la instalación predeterminada.

El Turbo C/C++ de Borland se instala poniendo el primer disco en la unidad de discos flexibles de la computadora y ejecutando el programa llamado INSTALL. Una vez que arranque este programa se verá una pantalla de saludo, que indica la cantidad de espacio de disco que se requiere para una instalación completa del compilador. Si no se dispone de esta cantidad de espacio de disco, tal vez quiera instalar una configuración mínima, tal como se dijo anteriormente. Si el espacio de disco no es de importancia, instale la opción predeterminada. Responda a todas las preguntas que se le hagan en el procedimiento de instalación. En el cuadro de diálogo Installation Overview, que es similar al de la figura H.5, seleccione la opción Start Installation. Con esto se instalará la opción predeterminada.

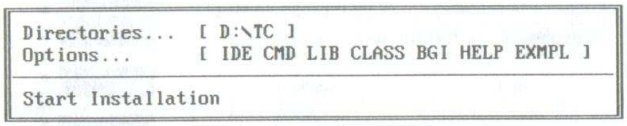


Figura H.5. *Cuadro de diálogo Installation Overview del Turbo C++ de Borland.*

Instalación de lo mínimo para el Turbo C/C++ para DOS de Borland

Mediante el uso del cuadro de diálogo que se presenta en la figura H.5 se puede determinar lo que instalará el compilador Turbo C/C++. Hay varias opciones que no necesita instalar si sólo pretende usar el compilador C y este libro. Siempre podrá instalar posteriormente opciones adicionales.

La primera opción es Directories. La selección de esta opción hace que se despliegue el cuadro de diálogo que se muestra en la figura H.6, que le permite instalar el compilador en una unidad diferente a la predeterminada.

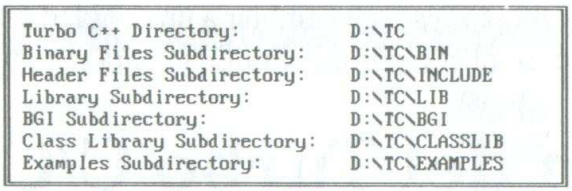


Figura H.6. *Cuadro de diálogo Directories del Turbo C++ de Borland.*

Cuando se pone el indicador resaltado en la línea Options del cuadro de diálogo Overview, se despliega un cuadro de descripción en la parte inferior de la pantalla. La figura H.6 muestra cómo puede verse el cuadro Description. Dependiendo de la versión del compilador que se use, las cantidades pueden ser mayores o menores.

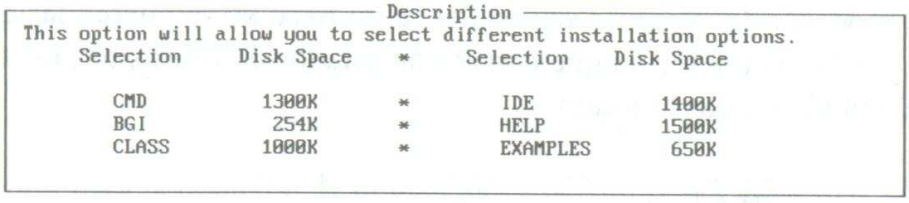


Figura H.7. *Cuadro Options Description del Turbo C++ de Borland.*

Al seleccionar la línea Options del cuadro de diálogo Overview se despliega el cuadro de diálogo que se muestra en la figura H.8.

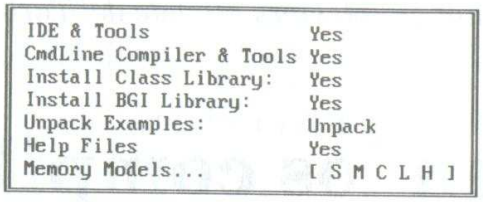


Figura H.8. *Cuadro de diálogo Options del Turbo C++ de Borland.*

Cinco de las siete opciones pueden ser puestas a Yes o No. La opción IDE & Tools es la única requerida si se va a usar el Ambiente Integrado de Desarrollo (IDE). Un ambiente integrado es el propio editor del compilador, que le permite hacer toda la programación, compilación y depuración en una sola área integrada.

Si usted tiene su propio editor probablemente no necesite el IDE ni sus herramientas. La segunda opción es el compilador de línea de comandos y sus herramientas. Si se está utilizando el IDE esta opción no es necesaria. Si usted utiliza su propio editor debe instalar esta opción.

Las opciones tercera y cuarta instalan las bibliotecas. Las bibliotecas de clase (Class Libraries) son usadas en el C++. Este libro solamente trata el C, por lo que estas bibliotecas no son necesarias. Las bibliotecas BGI son para hacer gráficos en Borland. Aunque este libro no trata estas bibliotecas específicas de Borland, es conveniente que las instale si piensa hacer cualquier programación con gráficos.

Las opciones quinta y sexta se refieren a los programas de ejemplo y los archivos de ayuda. Los archivos de ayuda tienden a ocupar una gran cantidad de espacio de disco; sin embargo, son muy útiles cuando se está comenzando a aprender a programar. Los archivos de ayuda no son indispensables, por lo que si se necesita espacio de disco se les puede dejar sin instalar. Los programas de ejemplo tampoco son necesarios, y, por lo tanto, tampoco necesitan ser instalados.

La selección de la opción final, Memory Models, despliega el cuadro de diálogo que se muestra en la figura H.9. De la misma forma que con el compilador de Microsoft, se puede

escoger qué módulos de memoria se instalen. Este libro no requiere más que los módulos Small/Tiny. De la misma forma que con el compilador de Microsoft, tal vez le convenga instalar también el modelo Medium.

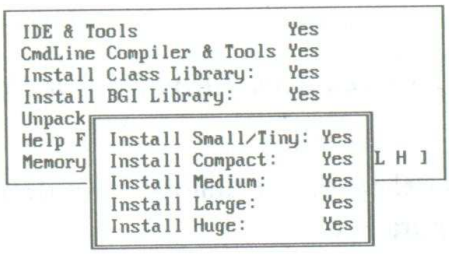


***Figura H.9.** Cuadro de diálogo Memory Module del Turbo C++ de Borland.*

¿Qué ofrecen los compiladores?

Cada compilador proporciona su propio conjunto de características, herramientas y ejemplos. A continuación se presentan unos cuantos compiladores y algunas de las características que ofrecen. Estas descripciones no son exhaustivas y no se tratan todas las herramientas. El propósito de esto es mostrar que hay diferencias entre los compiladores, incluso entre aquellos producidos por la misma compañía. Cuando esté seleccionando un compilador busque uno que satisfaga la mayoría de sus necesidades. La mayor parte de la información que se presenta aquí se refiere a las versiones que eran actuales cuando fue escrito el libro. Los compiladores principales son actualizados a veces cada año. Siendo un programador de C principiante no es necesario usar la versión actual de un compilador de C. Tal como se dijo anteriormente, mientras el compilador que se use sea compatible con ANSI C se tendrá la capacidad de usarlo con este libro.

Borland C++

El compilador Borland C++ es un compilador a nivel profesional. Como tal, contiene mucho más de lo que necesita el programador de C/C++ medio. Muchas de sus características están orientadas más hacia el programador de C++ que al de C. A continuación se presenta una lista de muchas de las características y herramientas que proporciona el compilador Borland:

- ❑ Soporte para el ANSI C (compatible al 100%).
- ❑ Soporte para la especificación del C++ de AT&T versión 3.0.
- ❑ Optimación de código (optimación global).
- ❑ Encabezados precompilados.

- Una interfaz de modo protegido de DOS (DPMI), que permite que sean compilados programas de modelos large (huge).
- Soporte para la creación de bibliotecas de enlace dinámico (DLL).
- Soporte para la creación de aplicaciones de Windows 3.0 y 3.1. El compilador de recursos le ayuda en la compilación de programas de Windows.
- Soporte para la creación de archivos de recursos con el Resource Workshop.
- Soporte para la creación de sistemas de ayuda con el compilador para Help.
- Soporte para la compilación de aplicaciones del DOS como aplicaciones EasyWin (aplicaciones de Windows).
- Un editor de archivos múltiples.
- Soporte para código de ensamblador en línea por medio de un ensamblador integrado.
- Un ambiente de desarrollo integrado (IDE) soportado por Windows.
- Soporte para la revisión de objetos creados en C++ usando el ObjectBrowser integrado.
- Soporte para la visión de mensajes de Windows usando WinSight.
- Administrador de memoria Virtual Run-time Object-Oriented Memory Manager (VROOM), que es usado para el traslapado de código.
- Ayuda de hipertexto.
- Soporte para clases contenedoras en C++.
- Compatibilidad con NMAKE para la automatización de la compilación y el enlace.
- Soporte para plantillas del C++.

Turbo C++ para DOS de Borland

Los productos Turbo de Borland son compiladores de nivel elemental. Estos compiladores proporcionan todo lo que se necesita típicamente para los usuarios caseros de C. Se quitan pocas de las características sofisticadas de los compiladores profesionales. Aun cuando el compilador Borland profesional soporta aplicaciones tanto del DOS como de Windows, Borland proporciona dos paquetes de compilador Turbo que manejan estas aplicaciones por separado. Uno es para el DOS y otro es para Windows. La mayoría de las versiones de los compiladores Turbo C++ también contienen compiladores de C. A continuación se presentan muchas de las características del compilador Turbo C++ para el DOS.

- ❑ Soporte para la especificación del ANSI C.
- ❑ Soporte para el C++ estándar de AT&T.
- ❑ Soporte para encabezados precompilados.
- ❑ Capacidad para compilar programas tipo huge con la interfaz de modo protegido del DOS (DPMI).
- ❑ Un ambiente integrado de desarrollo con todas las características (IDE).
- ❑ Soporte para overlay con el Virtual Run-time Object-Oriented Memory Manager (VROOM).
- ❑ Ayuda en línea con ejemplos que pueden ser cortados y pegados dentro del ambiente de desarrollo.
- ❑ Soporte para todos los flujos de entrada/salida estándar del C++.
- ❑ Soporte para clases contenedoras.
- ❑ Soporte para matemática compleja y matemática BCD.
- ❑ Soporte para el NMAKE de Microsoft.

Edición estándar del Visual C++ de Microsoft

Los compiladores Visual C++ de Microsoft también son compiladores de nivel profesional. El modelo de nivel elemental es llamado *Edición Estándar.* Aunque se le llame "edición estándar" todavía es un compilador a nivel profesional. Proporcionan muchas características y herramientas tales como

- ❑ Soporte para el ANSI C (compatible al cien por ciento).
- ❑ Soporte para la especificación de la versión 3.0 del C++ de AT&T.
- ❑ Un ambiente visual para el desarrollo de aplicaciones, Visual Workbench.
- ❑ El App Studio, para la creación y edición de recursos tales como cuadros de diálogo, menúes, barras de herramientas, controles y otras características del Windows.
- ❑ El App Wizard, para la creación de archivos fuente de la biblioteca de clases fundamentales del C++ de Microsoft.
- ❑ El Class Wizard, un ambiente del C++ que permite la derivación de nuevas clases, la visión de mensajes de Windows, la creación fácil de nuevas funciones para el manejo de mensajes y más.

La edición profesional también

- ❑ Soporta aplicaciones basadas en el DOS.
- ❑ Soporta aplicaciones de código P del DOS.
- ❑ Soporta aplicaciones de código P de Windows.
- ❑ Soporta overlays.
- ❑ Soporta programas COM del DOS.
- ❑ Soporta optimización.
- ❑ Proporciona un depurador (CodeView) que trabaja dentro de Windows.
- ❑ Incluye el juego para desarrollo de sistemas (SDK).

Otros compiladores

Se dispone de muchos otros compiladores. El compilador Zortech lo proporciona Symantec Corporation. La versión actual de este compilador soporta muchas de las mismas características de los compiladores de Borland y Microsoft. Symantec produce compiladores para Windows, DOS y OS/2. Además, ThinkC también es producido por Symantec. ThinkC es un compilador que trabaja en las computadoras Macintosh para el desarrollo de aplicaciones de la Macintosh.

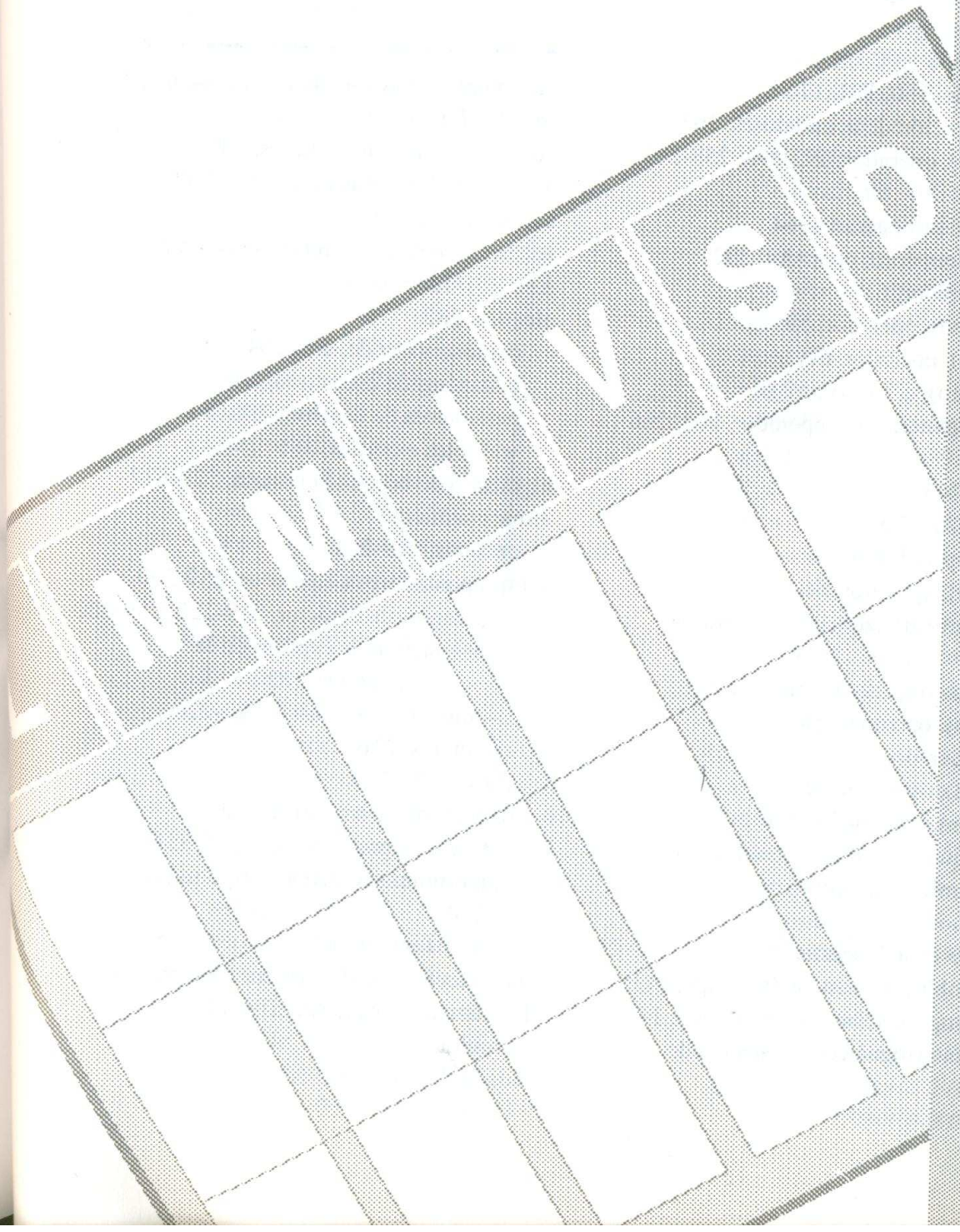
Existen otros compiladores en el mercado. Varían en precio y características. Watcom International Corporation produce un compilador profesional llamado Watcom C. Varios compiladores de alto nivel que se encuentran disponibles no son muy conocidos. Franklin Software produce un Franklin C Professional Developers Kit, Archimedes Software, Inc. produce Archimedes C-Cross Compiler y Liant Software Corporation produce el compilador Liant LPI-C++.

En el rango medio también hay varios compiladores. Archelon Inc. vende el Archelon C para el procesador i960. Byte Craft Limited ofrece dos compiladores: Byte Craft C6805 Code Development System y Byte Craft Z8C Code Development System. Metaware, Incorporated tiene el compilador Metaware High C/C++.

Hay otros compiladores que no son muy caros. Imagesoft Inc. tiene un compilador de C++ llamado Glockenspiel CommonView. Lattice, Inc. ofrece un compilador mejor conocido, Lattice C. El Lattice C está disponible para el DOS y OS/2. Clarion Software produce CLARION TopSpeed C y CLARION TopSpeed C++. Gimpel Software produce Gimpel C-Vision. Un último compilador que debe ser mencionado, probablemente el más barato de los que se han mencionado, es Power C de Mix Software.

Estos son unos cuantos de los compiladores que pueden comprarse. De hecho, hay una multitud de compañías que venden compiladores C. No importa qué marca de compilador decida usar mientras aprende, ni qué versión, siempre y cuando el compilador sea compatible con ANSI. Por ahora tome cualquier compilador y úselo.

Indice



Símbolos

! (no), operador, 75-77
!= (no igual), operador, 66
(encadenador), operador, definición de macro, 572
(concatenación), operador, macros, 573
% (módulo), operador, 60-62
&& (y), operador, 75-77
&, operadores
  AND, 555-556
  dirección de, 150, 153, 174
    inicialización de apuntadores, 193
* (indirección), operador, 192-193, 258-261, 500, 507
* (multiplicación), operador, 61
** (doble indirección), operador, 392
+ (suma), operador, 61
++ (incremento), operador, 59, 263
- (sustracción), operador, 61
— (decremento), operador, 59-61
-> (membresía indirecta), operador, 261, 266
. operadores
  de punto, 243
  miembro, 255, 261, 267
/ (división), operador, 61
< (menor que), operador, 66
<< (desplazamiento izquierdo), operador, 554-555
<= (menor que o igual a), operador, 66
= (asignación), operador, 58
== (igual), operador, 60-69
> (mayor que), operador, 66
> (redirección), símbolo, 369-371
s>= (mayor que o igual a), operador, 66
>> (desplazamiento derecho), operador, 554-555
\ (diagonal inversa), carácter, 55
\" (comilla doble), secuencia de escape, 141
\' (comilla simple), secuencia de escape, 141
\? (signo de interrogación), secuencia de escape, 141
\? (signo de interrogación), secuencia de escape, 541
\\ (diagonal inversa), secuencia de escape, 141
^ (O exclusivo), operador, 555-556
{ } llaves, 27
| (O inclusivo), operador, 555-556
|| (O), operador, 75-77
~ (complemento), operador, 556

A

\a, timbre (alerta), secuencia de escape, 141
abs(), función, 516
acceso aleatorio de archivos, 443-449
acceso de elementos de arreglo, 202
acceso directo, 193
acceso secuencial a archivos, 443-449
acos(), función, 514
alcance, 282
  demostración de, 282-284
  instancias de estructura, 296
  parámetros de función, 290
  variables externas, 285
algoritmo de búsqueda binaria, 529-537
almacenamiento
  cadenas, 219-221
    apuntadores a tipo char, 259-260
    arreglos de carácter, 219-221, 259-260
    asignación de espacio en la compilación, 222
    función malloc(), 222-226
  caracteres, 216
  clases, 293-294
  elementos de arreglo, 198-200
  espacio, asignación, 224, 545-551
  requerimientos, datos de tipo numérico, 183-184
alpha() función, 420
American Standard Code for Information Interchange, véase ASCII AND (&), operador, 555-556
anidado de, 71

directivas #include, 575
enunciados
for, 123-125
while, 128-130
estructuras, 245-248
anidados
ciclos, 134-136
comentarios, 26
enunciados if, 71-72
ANSI (American National Standards Institute), estándar, soporte, 680
apertura de archivos de disco, 427-431
argumentos de modo, 428-429
apuntadores de arreglo, 249
apuntadores de cabeza, listas encadenadas, 273
apuntadores de cadena, 221
apuntadores tipo void, 500-504
modificación, 545
apuntadores, 190-192
a apuntadores, 392-393
a apuntadores char, 559
a arreglos, 197-204
almacenamiento de elementos de arreglo, 198-200
arreglos multidimensionales, 393-401
nombres de arreglo como apuntador, 197
a estructuras, 260-262
a funciones, 411-421
control del orden, 417-421
declaración, 411-412
inicialización, 412-413
llamado de funciones con, 413-415
paso como argumentos, 416-417
versus funciones que regresan apuntadores, 507
acceso de arreglos de estructuras, 262-265
aritmética de apuntadores, 200-204
arreglos multidimensionales, 397-398
arreglos de, 249, 402-411
paso a funciones, 404-405
programa de ejemplo, 405-411
variables char, 403-404
cadenas, 221
como miembros de estructuras, 258-260
comparación, 204
creación, 191-192
declaración, 192, 392
decremento, 201-203
diferenciación, 203
errores, 231-232
funciones que regresan, 507-509
incremento, 201, 266
inicialización, 192-193
modificación, 545
paso, 510
a funciones, 496-500
arreglos a funciones, 206-210
arreglos multidimensionales, 399-401
precauciones, 204-205
tipo void, 500-504
usos, 193-195
va_list, 505
variables de varios bytes, 195-197
apuntadores, 192, 392
a funciones, 411-412
arreglos, 171, 176-184
estructuras, 242-243
variables, 42, 190
Archelon C, compilador, 691
Archimedes C, compilador cruzado, 691
archivo de encabezado de E/S estándar, véase STDIO.H, archivo de encabezado
archivos
acceso aleatorio a archivos, 443-449
acceso secuencial a archivos, 443-449
archivos de disco
apertura de, 427-431
argumentos de modo, 428-429
borrado de, 452-453
cierre de, 441-443
copia de, 454-457
escritura de datos a, 431-441
E/S de archivos directos, 438-441
E/S de archivos formateados, 431-435

E/S de caracteres, 436-437
fin de archivo, detección de, 449-452
flujos, 426
binarios, 427
de texto, 426-427
introducción de texto desde, 349-351
lectura de caracteres, 341
lectura de datos desde, 431-441
E/S de archivos directos, 438-441
E/S de archivos formateados, 431-435
E/S de caracteres, 436-437
nombres de archivo, 427
renombrado, 453-454
vaciado de, 441-443
archivos de encabezado, 23, 567
ASSERT.H, 524-526, 618
CALC.H, 565-566
con módulos secundarios, 564
CTYPE.H, 487-491, 616-617
ERRNO.H, 526-527
inclusiones múltiples, evitándolas, 578-579
MATH.H, 617-618
PROG.H, 578
prototipos de función, 620-626
rutas explícitas, 581
STDARG.H, 504, 618
STDIO.H, 148, 153-154, 229, 612-614
STDLIB.H, 182, 546, 610-611
STRING.H, 615-616
TIME.H, 518, 614
archivos de inclusión, véase archivos de encabezado
archivos de módulos secundarios, 566
archivos ejecutables, nombres de archivo, 581
archivos fuente
CALC.C, 565-566
DATABASE.C, 568
EX2-2.C, 32-33
EX2-4.C, 33
EX2-5.C, 33
EXPENSES.C, 173
GRADES.C, 176-177
HELLO.C, 12-15
con errores de compilación, 14
IF.C, 68-69
LIST_IT.C, 28-30
MULTIPLY.C, 22-23
programación con varios, véase programación modular
RANDOM.C, 180
SECONDS.C, 61-62
SIZEOF.C, 41
SQUARE.C, 565-566
UNARY.C, 59-60
week1.c, 160-164
week2.c, 380-387
week3.c, 586-594
archivos temporales, 457-458
argumentos de la línea de comandos, 551-553
argumentos, 22
de la línea de comandos, 551-553
paso de, a funciones, 105
por referencia, 496-500
por valor, 496-500
versus parámetros, 96-99
arreglo de una sola dimensión, 170-175
arreglos bidimensionales, 175
arreglos de cuatro dimensiones, 175
arreglos multidimensionales, 175, 178
apuntadores a, 393-401
aritmética de apuntadores, 397-398
inicialización, 179
paso de, a funciones, 399
arreglos tridimensionales, 175
arreglos, 170
acceso de elementos, 202
almacenamiento de elementos, 198-200
apuntadores a, 197-204
almacenamiento de elementos de arreglo, 198-200
aritmética de apuntadores, 202-204
arreglos multidimensionales, 393-401
nombres de arreglo como, 197
`char` (carácter), 235
cadenas, almacenamiento de, 219-220

inicialización, 220-221
tamaño de cadenas en, 236
de apuntadores, 402-411
a variables char, 403-404
paso de, a funciones, 404-405
programa de ejemplo, 405-411
de estructuras, 251-255
acceso, 262-265
inicialización, 256-257
de una sola dimensión, 170-175
declaración, 116-117, 171, 176-184
denominación, 176
dimensiones, 185
estructuras que contienen, 249-251
inicialización, 178-182, 185
multidimensionales, 175, 178, 393-401
aritmética de apuntadores, 397-398
paso de, a funciones, 399-401
notación de apuntadores, 205
notación de subíndices, 205-206
numeración de elementos, 170
ordenamiento, 530-537
paso de, a funciones, 206-210
arreglos de apuntadores, 404-405
arreglos multidimensionales, 399-401
tamaño de los elementos, 182, 396
tamaño máximo, 182-184
utilización, 171
versus variables individuales, 185
asctime(), función, 519-521
asignación de memoria, 545-551
asin(), función, 514
asm, palabra clave, 600
assert(), función, 524-526
ASSERT.H, archivo de encabezado, 524-526
prototipos de función, 618
asterisco (*), operador de multiplicación, 46
atan(), función, 514
atan2(), función, 514
atexit(), función, 322-324
\a, timbre (alerta), secuencia de escape, 141
atof(), función, 486-487
atoi(), función, 485-486
atol(), función, 486
auto, palabra clave, 600
average(), función, 506

B

bloques, 27, 55-56, 67
definición de variables locales dentro de, 294-295
véase también enunciados compuestos
Borland International, Inc.
C++, compilador, 681, 688-689
paquetes de compilador, 681
Turbo C/C++ para DOS, compilador, 689-690
instalación de, 685-688
borrado de archivos de disco, 452-453
break, enunciado, 302-304
break, palabra clave, 600
bsearch(), función, 529-537
búsqueda de cadenas, 476-482
bsearch(), función, 534-537
strcspn(), función, 478-479
strchr(), función, 476-477
strpbrk(), función, 480
strrchr(), función, 478
strspn(), función, 479-480
strstr(), función, 481-482
búsqueda, véase búsqueda
by_ref(), función, 499
by_value(), función, 499
Byte Craft C6805 Code Development System, compilador, 691
Byte Craft Z8C Code Development System, compilador, 691
bytes, 36
char (carácter), variables, 220

C

%c, especificador para conversión de un solo carácter, 145-148

.C, extensión, véase archivos fuente
C CLARION TopSpeed, compilador, 691
C++ CLARION TopSpeed, compilador, 691
C++, compiladores, 681, 688-689
C, lenguaje de programación
  compiladores, selección de, 680-682
  historia, 4
  ventajas, 4-5
.C, extensión, véase archivos fuente
C, preprocesador, 569
cadenas para formateo, 141
cadenas, 55, 235
  almacenamiento, 219-221
    apuntadores a tipo `char`, 259-260
    arreglos char como miembros de estructuras, 259-260
    arreglos `char`, 219-221
    asignación de espacio a la compilación, 222
    función `malloc()`, 222-226
  arreglos de apuntadores, 402
  búsqueda, 476-482
    `bsearch()`, función, 534-537
    `strchr()`, función, 476-477
    `strcspn()`, función, 478-479
    `strpbrk()`, función, 480
    `strrchr()`, función, 478
    `strspn()`, función, 479-480
    `strstr()`, función, 481-482
  comparación, 472-476
    cadenas parciales, 475-476
    ignorando mayúsculas y minúsculas, 475
  concatenación, 469-472
  conversión
    a variables numéricas, 485-487
    de mayúsculas y minúsculas, 482-483
  copia, 465-469
  desplegado, 227
    con la función `fputs()`, 361-362
    con la función `printf()`, 228
    con la función `putchar()`, 360
    con la función `puts()`, 227-228, 361-362
  formateo, 141
  funciones, 615-616
  inversión del orden de caracteres, 483
  lectura desde el teclado, 229
    `gets()`, función, 229-232
    `scanf()`, función, 232-236
  longitud, determinación de, 464-465
  ordenamiento, 534-537
  prueba de caracteres, 487-491
  reuso, 559
  tamaño en arreglos, 236
CALC.C, archivo fuente, 565-566
CALC.H, archivo de encabezado, 565-566
`calloc()`, función, 546
campos de bit, 559
  en estructuras, 556-558
campos de bits, 559
  en estructuras, 556-558
carácter diagonal inversa (\), 55
carácter subrayado (_), 38-39
caracteres
  almacenamiento de, 216
  ASCII extendidos, impresión de, 218-219
  "desobteniendo", 348
  diagonal inversa (\), 55
  lectura desde archivos de disco, 341
  no igual a (!=) en enunciados if-else, 74
  nueva línea (\n), 141
    inserción automática de, 227
  prueba de, 487-491
  subrayado (_), 38-39
caracteres para especificación de tipo, 353
`case`, palabra clave, 600
`ceil()`, función, 516
`%c`, especificador para conversión de un solo carácter, 145-148
ciclo para el desarrollo de programas, 6-11
  compilación del código fuente, 8
  creación del código fuente, 7-8
  pasos desde el código fuente hasta el archivo ejecutable, 9
ciclos
  anidados, 134-136
  `do...while`, 130-133

for, 117-123
infinitos, 309-312
terminación, 302-306
ciclos infinitos, 309-312
cierre de archivos de disco, 441-443
clase de almacenamiento automático, 293
clase de almacenamiento estético, 293
clases de almacenamiento de variables, tipos, 293-294
clases, almacenamiento, 293-294
cláusulas, else, 69
cleanup(), función, 324
clock(), función, 521
código fuente
compilación, 8, 12-15
creación, 7-8
pasos desde el código fuente hasta el archivo ejecutable, 9
modificaciones del preprocesador, 569
códigos, ASCII, 216
coma (,), operador, 80-81, 121
comandos
del sistema operativo, ejecución de, 325-326
DOS, SET, 575
comentarios, 26-27
anidados, 26
comparación
de apuntadores, 204
de cadenas, 472-476
compare(), función, 420
compilación
código fuente, 8, 12-15
de archivos múltiples, 569
de programas, errores de compilación, 13-15
compilación condicional, 576-577
compiladores, 681
Archelon C, 691
C++ (Borland), 681, 688-689
CLARION TopSpeed C++, 691
CLARION TopSpeed C, 691
Gimpel C-Vision, 691
Glockenspiel Common View, 691
Lattice C, 691
Liant LP1-C++, 691
Power C, 691
selección de, 680-682
ThinkC, 691
Turbo C/C++ para DOS, 689-690
instalación de, 685-688
Visual C++ edición estándar, 690-691
instalación de, 682-685
Watcom C, 691
Zortech, 691-692
complemento (~), operador, 556
concatenación, cadenas, 469-472
concatenación, operador ##, macros, 573
const, palabra clave, 47-50, 600
constantes de cadena, literales, 55
constantes de punto flotante, 45
constantes enteras, notación, 45-46
constantes hexadecimales, 45
constantes literales, 44-46
cadenas, 55
de carácter, creación, 217
constantes manifiestas, véase constantes simbólicas
constantes simbólicas, 46-50, 575
creación, 217, 570
constantes, 44
creación de, 178
de punto flotante, 45
enteras, notación, 45-46
hexadecimales, 45
literales, 44-46
cadenas, 55
de carácter, creación de, 217
peso/año de nacimiento, programa, 48
simbólicas, 46-50, 575
creación de, 217, 570
contadores, incremento/decremento, 120-121
continue, enunciado, 304-306
continue, palabra clave, 600
continue_function(), función, 164
conversión
cadenas a variables numéricas, 485-487
de representaciones de tiempo, 519

por asignación, 543
por mayúsculas/minúsculas, 482-483
conversión automática de tipo, 542-543
conversión explícita de tipo
de tipo, 543-545
en expresiones aritméticas, 544-545
conversiones de tipo, 542
automáticas, 542-543
en expresiones, 542-543
conversiones de tipo, 542
automáticas, 542-543
en expresiones, 542-543
copia de
archivos de disco, 454-457
cadenas, 465-469
copy_file, función, 455-457
cos(), función, 515
cosh(), función, 515
ctime(), función, 519-521
CTYPE.H, archivo de encabezado, 487-491
prototipos de función, 616-617
cube(), función, 90
cubos, cálculo de, 89
cuerpo de la función, 90
componentes, 99-103
cuerpo, de función, 90, 99
char (carácter), arreglos, 235
cadenas, almacenamiento de, 219-220
inicialización, 220-221
tamaño de cadenas en, 236
char (carácter), tipo de dato, 216-217
char (carácter), variables, 40, 216-219
arreglos de apuntadores a, 403-404
naturaleza numérica, demostración, 217-218
char, palabra clave, 600

D

%d, especificador para conversión de entero decimal con signo, 145-148
DATABASE.C, archivo fuente, 568
__DATE__, macro predefinida, 579-580
datos numéricos, entrada de, 149-154
declaración de
decremento (—), operador, 59
decremento de apuntadores, 201-203
default, palabra clave, 600
#define, directiva, 47, 50
marcos de función, creación de, 570-573
macros de sustitución, 569-570
usos, 569
#define, enunciado, 178
defined(), operador, 577-578
definición de estructuras, 242-243
definiciones de
función, 90-92
ubicación de, 109-110
variables, 24
definiciones de función, 90-92
ubicación de, 109-110
delay(), función, 312
designación de programas, 93-95
desobteniendo caracteres, 348
%d, especificador para conversión de entero decimal con signo, 145-148
desplazamiento derecho (>>), 554-555
desplazamiento izquierdo (<<), operador, 554-555
desplegado de
cadenas
con la función fputs(), 361-362
con la función putchar(), 360
con la función puts(), 361-362
de la hora, 519-521
entrada de teclado, 405-411
derreferenciado de apuntadores vacíos, 501
diferenciación de apuntadores, 203
difftime(), función, 521-522
dimensiones, arreglos, 185
direcciones, 190
almacenamiento, 191
operador, olvido de, 155
directivas
#define, 47, 50
macros de función, creación de, 570-573

macros de sustitución, 569-570
usos, 569
#elif, 576-577
#else, 576-577
#endif, 576-577
depuración con, 577-578
#if, 576-577
depuración con, 577-578
#include, 23-24
#include, 575
#undef, 579
directivas de preprocesador, véase directivas
directorio estándar, 575
directorios, estándar, 575
display_instructions(), función, 164
display_report, función, 165
display_usage, función, 592
dispositivos, 332
división (/), operador, 61
do, palabra clave, 600
do...while, ciclo, 130-133
do...while, enunciado, sintaxis, 133-134
doble indirección (**), operador, 392
DOS, comandos, SET, 575
double, palabra clave, 600
draw_box(), función, 124

E

E/S de archivos directos, 438-441
E/S de archivos estándar, 334
E/S de archivos formateados, 351-359, 362-369, 431-435
E/S de caracteres, 336-351, 359-361, 436-437
E/S, 332-333
de archivos directos, 438-441
de archivos formateados, 351-359, 362-369, 431-435
de caracteres, 336-351, 359-361, 436-437
f nciones de, 612-614
ınea, 349-351
redirección de, 369-371
editores, 7-8
ejecución de programas, 10, 13, 27
ejecución del programa
control, 117-134
ramificación, 306-309
ejecución, de programa (control de), 117-134
#elif, directiva, 576-577
#endif, directiva, 576-577
depuración con, 577-578
else, cláusula, 69
else, palabra clave, 600
#else, directiva, 576-577
encabezado de función, 90
componentes, 96-99
encabezados, de función, 90
componentes, 96-99
encadenadores, 9
errores, 15
entrada de
datos numéricos, función scanf(), 149-154
texto desde archivos de disco, 349-351
entrada de línea, 349-351
entrada, 332
archivos directos, 438-441
archivos formateados, 351-359, 434-435
caracteres, 336-351, 436-437
dispositivos, 332
línea, 349-351
redirección, 369-371
teclado
aceptación, 336-359, 405-411
desplegado, 405-411
ordenamiento, 405-411
teclas especiales, 341-351
véase también E/S
enum, palabra clave, 600
enunciados compuestos, 55-56
enunciados nulos, 55
a continuación de enunciados for, 121, 124
enunciados para el control de programa, 67
enunciados, 24-25, 54
asignación, 54, 231
en enunciados if, 74

bloques, 27
break, 302-304
varios enunciados break, 304
compuestas, 55-56
continue, 304-306
control de programa, 67
decisión de cuándo usarlo, 136
#define, 178
do...while, 133-134
for, 117-123
anidado, 123-125
función, 101-102
goto, 306-309
evitándolo, 327
if, 67-72, 82
múltiples return en funciones, 102-104
nula, 55
a continuación de enunciados for, 121, 124
return, 90
switch, 312-320
versus ciclos anidados, 327
while, 125-127
anidado, 128-130
sintaxis, 127-128
y el espacio en blanco, 54-55
ERRNO.H, archivo de encabezado, 526-527
errores de
apuntador, 231-232
compilación, 13-15
encadenamiento, 15
escritura
a archivos de disco, 431-441
E/S de archivo directo, 438-441
E/S de archivo formateado, 431-435
E/S de carácter, 436-437
funciones de, 96
encabezado de función, 96-99
espacio
cadenas, asignándolas al momento de compilación, 222
de almacenamiento, asignándolo conforme se necesita, 224
memoria, 37
espacio en blanco, 54-55, 82
función scanf(), 150
espacios, 82
especificadores de conversión, 141
función fprintf(), 364
función printf(), 145-147, 364
función scanf(), 153
estándares, ANSI, 680
estructuras
anidamiento, 245-248
apuntadores a, 260-262
arreglos de, 251-255
inicialización, 256-257
asignación a cada una, 277
campos de bit en, 556-558
de arreglos, acceso de, 262-265
declaración, 277
inicialización, 255-257
pasándolas como argumentos a funciones, 265-266
que contienen
arreglos, 249-251
otras estructuras, 245-248
simples, 242
declaración, 244-245
definición/declaración, 242-243
miembros de estructuras, acceso de, 243-245
typedef, palabra clave, 275-276
versus uniones, 267
estructuras complejas, 245-251
estructuras simples, 242
declaración, 244-245
definición/declaración, 242-243
miembros de estructuras, acceso, 243-245
etiqueta, estructura, 255
etiquetas de estructura, 255
EX2-2.C, archivo fuente, 32-33
EX2-5.C, archivo fuente, 33
exit(), función, 318, 321-324
exp(), función, 515
expansión de macros (vista de), 574-575

expansión de macros, vista de, 574-575
EXPENSES.C, archivo fuente, 173
expresiones
  aritméticas, conversión explícita de tipo, 544-545
  complejas, 56-57
  conversiones de tipo en, 542-543
  creación de, con el operador de coma, 121
  relacionales
    combinación de, 75-81
    evaluación de, 72-75
  simples, 56
  sobrecarga de, 65
expresiones aritméticas, conversión explícita de tipo, 544-545
expresiones complejas, 56-57
expresiones relacionales
  combinación de, véase operadores lógicos
  evaluación, 72-75
expresiones simples, 56
ext_key(), función, 346-348
extern, palabra clave, 286-287, 567, 600
external, clase de almacenamiento, 293

F

%f, especificador de conversión para números decimales de punto flotante, 145-148
factorial(), función, 109
  uso de la recursión, 111
fclose(), función, 441-443
fcloseall(), función, 442
feof(), función, 449-452
%f, especificador de conversión para números decimales de punto flotante, 145-148
fflush(), función, 442-443
fgetc(), función, 341, 436
fgets(), función, 349-351, 436-437
__FILE__, macro predefinida, 579-580
fin de archivo, detección de, 449-452
float, palabra clave, 600
floor(), función, 516
flujos binarios, 334, 427
flujos predefinidos, 334
flujos, 332-334
  archivos de disco, 426
  binarios, 334, 427
  de texto, 334, 426-427
  equivalencia, 336
  funciones, 335-336
  predefinidos, 334
flushall(), función, 442-443
fmod(), función, 516
fopen(), función, 427-431
for, enunciado, 117-123
  anidado de, 123-125
  sintaxis, 122-123
for, palabra clave, 600
fprintf(), función, 362-369, 432-434
  cuándo usarlo, 371-373
  especificadores de conversión, 364
fputc(), función, 361
fputs(), función, 361-362, 437
Franklin C Professional Developers Kit, compilador, 691
fread(), función, 439-441
free(), función, 549-554
frexp(), función, 515
fscanf(), función, 434-435
fseek(), función, 446-449
ftell(), función, 444-446
fuera de rango, inicializaciones, 44
func(), función, 507
func1(), función, 416
funciones de biblioteca, 22, 25
funciones de búsqueda, 529-537
funciones de carácter, 615-616
funciones de diagnóstico, 618
funciones de fecha, 614
funciones de tiempo, 518-524, 614
  programa de ejemplo, 522-524
funciones definidas por el usuario, 22, 25
  ubicación, 109
funciones exponenciales, 515
funciones hiperbólicas, 515-516

funciones matemáticas, 514-517, 617-618
exponenciales, 515
hiperbólicas, 515-516
programa de ejemplo, 517
trigonométricas, 514-515
funciones para el manejo de errores, 524-529
funciones para la asignación de memoria, 235
malloc(), 222-226
listas encadenadas, 275
funciones trigonométricas, 514-515
funciones, 88-89, 620-626
abs(), 516
acos(), 514
alpha(), 420
apuntadores a, 411-412
control del orden, 417-421
declaración de, 411-412
inicialización, 412-413
llamado de funciones con, 413-415
paso como argumento, 416-417
argumentos, 22
asctime(), 519-521
asignación de memoria, 235
véase también malloc()
asin(), 514
assert(), 524-526
atan(), 514
atan2(), 514
atexit(), 322-324
atof(), 486-487
atoi(), 485-486
atol(), 486
average(), 506
bsearch(), 529-537
sby_ref(), 499
by_value(), 499
calloc(), 546
ceil(), 516
cleanup(), 324
clock(), 521
compare(), 420
con número variable de argumentos, 504-507
continue_function(), 164
copy_file(), 455-457
cos(), 515
cosh(), 515
ctime(), 519-521
cube(), 90
definidas por el usuario, 22, 25
delay(), 312
difftime(), 521-522
display_instructions(), 164
display_report(), 165
display_usage(), 592
draw_box(), 124
escritura de, 96-99
exit(), 318, 321-324
exp(), 515
ext_key(), 346-348
factorial(), 109
recursión, uso de, 111
fclose(), 441-443
fcloseall(), 442
feof(), 449-452
fflush(), 442-443
fgetc(), 341, 436
fgets(), 349-351, 436-437
floor(), 516
flushall(), 442-443
fmod(), 516
fopen(), 427-431
fprintf(), 362-369, 432-434
cuándo usar, 371-373
especificadores de conversión, 364
fputc(), 361
fputs(), 361-362, 437
fread(), 439-441
free(), 549-554
frexp(), 515
fscanf(), 434-435
fseek(), 446-449
ftell(), 444-446
func(), 507
func1(), 416
funciones de biblioteca, 22, 25

funciones de búsqueda, 529-537
funciones de cadenas, 615-616
funciones de carácter, 615-616
funciones de diagnóstico, 618
funciones de E/S, 612-614
funciones de fecha, 614
funciones de flujo, 335-336
funciones de tiempo, 518-524, 614
  programa de ejemplo, 522-524
funciones matemáticas, 514-517, 617-618
  exponenciales, 515
  hiperbílicas, 515-516
  programa de ejemplo, 517
  trigonométricas, 514-515
funciones para el manejo de archivos, 452-457
funciones para el manejo de errores, 524-529
fwrite(), 438-441
get_data(), 165, 593
get_int(), 488-491
get_lines(), 405-411
get_menu_choice(), 133, 144, 152
getc(), 341, 436
getch(), 179, 182, 339
getchar(), 335-338
getche(), 340
gets(), 229-232, 334-335, 349
half(), 503
labs(), 516
larger_of(), 103
largest(), 207-210
ldexp(), 515
localtime(), 519
log(), 515
log10(), 515
longitud, 101-104
look_up(), 594
llamado de, 25, 106-107
  con apuntadores a funciones, 413-415
  recursión, 107-109
main(), 23, 62
  argumentos de la línea de comandos, recuperación de, 552-553
  variables locales, 287
malloc(), 222-226, 275, 409, 546
menu(), 312
modf(), 516
nombres, 96, 111
operación de, 91-92
paso de apuntadores a, 496-500
paso de argumentos a, 105
paso de arreglos a, 206-210
  arreglos de apuntadores, 404-405
paso de arreglos multidimensionales a, 399-401
paso de estructuras como argumentos a, 265-266
pasos desde el código fuente hasta el archivo ejecutable, 9
perror(), 335, 527-529
pow(), 516
print_function(), 272
print_report(), 144
print_strings(), 405
print_value(), 282-284
printarray_1(), 401
printarray_2(), 401
printf(), 24-25, 62, 140-148, 217, 228, 282, 334-335, 362-369
  cadenas para formateo, 363
  especificadores de conversión, 145-147, 364
  secuencia de escape, 142-144
product(), 25
putc(), 361, 437
putchar(), 335, 359-361
puts(), 103, 152-154, 227-228, 334-335, 361-362
  mensajes, desplegado de, 148-155
que regresan apuntadores, 507-509
  versus apuntadores a funciones, 507
rand(), 182
realloc(), 547-549, 559
regreso de valores de, 102-103
remove(), 452-453
rename(), 453-454
reverse(), 420

rewind(), 444-446
scanf(), 25, 62, 149-155, 174, 232-234, 334-335, 351-354
  caracteres extra, 354-356
  programa de ejemplo, 356-359
sin(), 515
sinh(), 516
sleep(), 312
sort(), 405-411, 417-421
sqrt(), 516
strcat(), 469-471
strcmp(), 473-474
strcmpi(), 475
strcmpl(), 475
strcpy(), 252, 409, 465-467
strcspn(), 478-479
strchr(), 476-477
strdup(), 468-469
stricmp(), 475
_stricmp(), 475
strlen(), 409, 464-465
strlwr(), 482-483
strncat(), 471-472
strncmp(), 475-476
strncpy(), 467-468
strnset(), 484-485
strpbrk(), 480
strrchr(), 478
strrev(), 483-485
strset(), 484-485
strspn(), 479-480
strupr(), 482-483
su papel en la programación estructurada, 93-95
system(), 325-326
tan(), 515
tanh(), 516
time(), 518-519
tmpnam(), 457-458
ubicación de, 109-110
ungetc(), 348
valores, regreso de, 110
variables, declaración dentro, 99-101
  véase también variables locales
  versus macros, 503, 573-574
  vprintf(), 335
fwrite(), función, 438-441

G

get_data(), función, 165, 593
get_int(), función, 488-491
get_lines(), función, 405-411
get_menu_choice(), función, 133, 144, 152
getc(), función, 341, 436
getch(), función, 179, 182, 339
getchar(), función, 335-338
getche(), función, 340
gets(), función, 229-232, 334-335, 349
  sintaxis, 232
Gimpel-C Vision, compilador, 691
Glockenspiel Common View, compilador, 691
goto, enunciado, 306-309
  evitando, 327
  sintaxis, 306
goto, palabra clave, 600
GRADES.C, archivo fuente, 176-177

H

.H, extensiones, 581
half(), función, 503
heap, memoria, 549
HELLO.C, archivo fuente, 12-15
  con errores de compilación, 14
.H, extensiones, 581

I

IDE (interfaz integrada de programación), 574
#if, directiva, 576-577
  depuración con, 577-578
if, enunciado, 67-72

compuesto, 82
anidado, 71-72
if, palabra clave, 600
IF.C, archivo fuente, 68-69
igual (==), operador, 66, 69
igual, signo (=), operador de asignación, 43
impresión
caracteres extendidos ASCII, 218-219
de varias líneas de texto, 147
números del 0 al 99, 121
#include, directiva, 23-24, 575
incrementación de apuntadores, 201
incremento (++), operador, 59, 263
indirección (*), operador, 192-193, 258-261, 500, 507
indirección, 193
múltiple, 393
inicialización de
apuntadores, 192-193
a funciones, 412-413
arreglos, 178-182, 185
estructuras, 255-257
variables locales, estáticas versus automáticas, 289
inicializaciones, fuera de rango, 44
instalación de
Turbo C/C++ para DOS (Borland), 685-688
Visual C/C++ (Microsoft), 682-685
instancias
declaración de estructuras sin, 277
estructura, alcance, 296
instrucciones de asignación, 54, 231
en los enunciados if, 74
instrucciones de la función, 101-102
int (entero), variables, 39-40
conversión de cadenas a, 485-486
int, palabra clave, 600
interfaz integrada de programación (IDE), 574
inversin del orden de los caracteres de una cadena, 483
iteraciones, funciones recursivas, 109

J

juego de caracteres ASCII (American Standard Code for Information Interchange), 216
extendido, 236
rangos, 218
juego de caracteres extendidos ASCII, 236
impresión de, 218-219
juegos de caracteres, ASCII (American Standard Code for Information Interchange), 216
extendido, 236

K

K (kilobytes), 36
KEYBOARD.OBJ, archivo objeto, 568
kilobytes (K), 36

L

labs(), función, 516
larger_of(), función, 103
largest(), función, 207-210
Lattice C, compilador, 691
ldexp(), función, 515
lectura
de archivos de disco, 431-441
E/S de archivos directos, 438-441
E/S de archivos formateados, 431-435
E/S de carácter, 436-437
de caracteres desde archivos de disco, 341
de líneas desde stdin, 349
teclas extendidas, 342
lenguaje de máquina, 8
Liant LPI-C++, compilador, 691
liberación de memoria, 559
__LINE__, macro predefinida, 579-580
líneas en blanco, 82
en la entrada, prueba de, 230, 231
líneas, múltiples (impresión), 147

LIST_IT.C, archivo fuente, 28-30
lista variable de argumentos, herramientas, 504-505
listados
1.1. Archivo fuente HELLO.C, 12
1.2. HELLO.C con un error, 14
2.1. Archivo fuente MULTIPLY.C, 22-23
2.2. Archivo fuente LIST_IT.C, 28-30
3.1. Archivo fuente SIZEOF.C, 41
3.2. Programa peso/año de nacimiento, 48
4.1. Archivo fuente UNARY.C, 59-60
4.2. Archivo fuente SECONDS.C, 61-62
4.3. Archivo fuente IF.C, 68-69
4.4. Enunciado `if` con una cláusula else, 70
4.5. Evaluación de expresiones relacionales, 72
4.6. Precedencia de operadores lógicos, 78
5.1. Cálculo de un cubo (funciones definidas por el usuario), 89
5.2. Argumentos versus parámetros, 97-98
5.3. Demostración de variables locales, 100
5.4. Varios enunciados return en funciones, 102-103
5.5. Funciones recursivas, 108
6.1. Demostración del enunciado `for`, 119
6.2. Enunciados for anidados, 123
6.3. Demostración del enunciado while, 126
6.4. Enunciados while anidados, 128-129
6.5. Demostración del enunciado `Do...while`, 132
7.1. Secuencias de escape de `printf()`, 142-143
7.2. Uso de `printf()` para desplegar valores numéricos, 146
7.3. Uso de `scanf()` para obtención de valores numéricos, 150-151
8.1. Archivo fuente EXPENSES.C, 173
8.2. Archivo fuente GRADES.C, 176-177
8.3. Archivo fuente RANDOM.C, 180
8.4. Determinación de los requerimientos de almacenamiento con el operador `sizeof()`, 183-184
9.1. Ilustración del uso básico de apuntadores, 194
9.2. Desplegado de las direcciones de elementos sucesivos de un arreglo, 199
9.3. Uso de la aritmética de apuntadores y de la notación de apuntadores para accesar elementos de arreglo, 202
9.4. Paso de arreglos a funciones, 207-208
9.5. Forma alternativa para pasar arreglos a funciones, 209
10.1. Demostración de la naturaleza numérica de las variables char, 217-218
10.2. Impresión de caracteres extendidos ASCII, 218
10.3. Uso de la función `malloc()` para ubicar espacio de almacenamiento para datos de cadenas, 224-225
10.4. Uso de la función `puts()` para desplegar texto en pantalla, 227
10.5. Entrada de datos de cadena con la función `gets()`, 229
10.6. Prueba de la entrada de líneas en blanco con la función `gets()`, 230
10.7. Entrada de datos numéricos/texto con la función `scanf()`, 234
11.1. Una demostración de estructuras que contienen otras estructuras, 247-248
11.2. Una demostración de una estructura que contiene miembros de arreglo, 250
11.3. Demostración de un arreglo de estructuras, 253
11.4. Acceso de elementos sucesivos de un arreglo incrementando un apuntador, 263-264
11.5. Paso de una estructura como argumento de una función, 265-266
11.6. Un ejemplo del uso equivocado de las uniones, 268
11.7. Un uso práctico de una unión, 270-271
12.1. La variable `x` es accesible dentro de la función `print_value`, 282-283

12.2. La variable x no es accesible dentro de la función print_value, 283
12.3. La variable externa x es declarada como extern, 286-287
12.4. Ilustra la diferencia entre variables automáticas y locales estéticas, 288
12.5. Muestra las variables locales dentro de bloques, 294
13.1. Uso del enunciado break, 303
13.2. Demostración del enunciado continue, 305
13.3. Demostración del enunciado goto, 307-309
13.4. Uso de un ciclo infinito para implementar un sistema de menú, 310-311
13.5. Demostración del enunciado switch, 313-314
13.6. Uso correcto de switch, incluyendo enunciados break en el lugar adecuado, 314-315
13.7. Uso del enunciado switch para ejecutar un sistema de menú, 316-317
13.8. Otra manera de usar el enunciado switch, 318-319
13.9. Uso de las funciones exit() y atexit(), 323-324
13.10. Uso de la función system() para ejecutar comandos del sistema, 325-326
14.1. La equivalencia de los flujos, 336
14.2. Demostración de la función getchar(), 337
14.3. Uso de la función getchar() para la entrada de una línea completa de texto, 338
14.4. Uso de la función getch(), 339
14.5. Uso de la función getch() para la entrada de una línea completa, 340
14.6. Aceptación de la entrada de teclas extendidas, 342
14.7. Un sistema de menú que responde a la entrada de teclas de función, 346-348
14.8. Uso de la función fgets() para la entrada de teclado, 350
14.9. Limpieza de caracteres adicionales en stdin para evitar errores, 355-356
14.10. Algunas formas de usar a scanf() para la entrada de teclado, 357-359
14.11. La función putchar(), 360
14.12. Desplegado de una cadena con putchar(), 360
14.13. Uso de la función puts() para el desplegado de cadenas, 362
14.14. Algunas maneras de usar la función printf(), 366-368
14.15. Programa para demostrar la redirección de la entrada y la salida, 370
14.16. Envío de salida a la impresora, 372-373
15.1. Relación entre arreglos multidimensionales y apuntadores, 396
15.2. Determinación del tamaño de elementos de arreglo, 396
15.3. Aritmética de apuntadores con arreglos multidimensionales, 397-398
15.4. Paso de arreglos multidimensionales a funciones usando un apuntador, 399-401
15.5. Inicialización y uso de un arreglo de apuntadores de tipo char, 404
15.6. Paso de un arreglo de apuntadores a una función, 404-405
15.7. Programa que lee, ordena y despliega texto desde el teclado, 406-411
15.8. Uso de apuntadores a funciones para llamar funciones, 413
15.9. Uso de un apuntador a funciones para llamar funciones, 414-415
15.10. Paso de un apuntador a una función como un argumento, 416-417
16.1. Uso de fopen() para abrir archivos de disco, 429-430

16.2. Demostración de la equivalencia de la salida formateada de `fprintf()`, 432-433
16.3. Uso de `scanf` para leer datos formateados de un archivo de disco, 435
16.4. Uso de `fwrite()` y `fread()` para el acceso de archivos directos, 439-441
16.5. Uso de `ftell()` y `rewind()`, 444-446
16.6. Acceso aleatorio de archivos con `fseek()`, 447-449
16.7. Uso de `feof()` para detectar el final de un archivo, 450-452
16.8. Uso de la función `remove()` para borrar un archivo de disco, 452-453
16.9. Uso de `rename()` para cambiar el nombre de un archivo de disco, 454
16.10. Una función que copia un archivo, 455-457
16.11. Uso de `tmpnam()` para crear nombres de archivos temporales, 457-458
17.1. Uso de la función `strlen()` para determinar la longitud de una cadena, 464-465
17.2. Antes de usar `strcpy()` se debe asignar espacio de almacenamiento para la cadena de destino, 466
17.3. La función `strncpy()`, 467-468
17.4. Uso de `strdup()` para copiar cadenas con asignación automática de memoria, 468-469
17.5. Uso de `strcat()` para concatenar cadenas, 470-471
17.6. Uso de la función `strncat()` para concatenar cadenas, 471-472
17.7. Uso de `strcmp()` para comparar cadenas, 473-474
17.8. Comparación de partes de cadenas con `strncmp()`, 475-476
17.9. Uso de `strchr()` para buscar un solo carácter en una cadena, 477
17.10. Búsqueda de un conjunto de caracteres con la función `strcspn()`, 478-479
17.11. Búsqueda del primer carácter que no concuerda con `strspn()`, 479-480
17.12. Uso de `strstr()` para buscar una cadena dentro de otra, 481-482
17.13. Conversión a mayúsculas o a minúsculas de una cadena con `strlwr()` y `strupr()`, 482-483
17.14. Una demostración de `strrev()`, `strnset()` y `strset()`, 485-485
17.15. Uso de `atof()` para convertir cadenas a variables de tipo numérico doble, 486-487
17.16. Uso de las macros `isxxxx()` para implementar una función que da entrada a un entero, 489-491
18.1. Paso por valor y paso por referencia, 498-499
18.2. Uso de apuntadores tipo `void`, 501-503
18.3. Uso de una lista variable de argumentos, 505-506
18.4. Regreso de un apuntador desde una función, 508
19.1. Uso de las funciones de la biblioteca matemática de C, 517
19.2. Uso de las funciones de la biblioteca de tiempo de C, 522-524
19.3. Uso de la macro `assert()`, 525-526
19.4. Uso de `perror()` y `errno` para manejar errores del momento de ejecución, 527-529
19.5. Uso de las funciones `qsort()` y `bsearch()` con valores, 531-532
19.6. Uso de `qsort()` y `bsearch()` con cadenas, 534-537
20.1. Una división entera pierde la parte fraccional de la respuesta, 544
20.2. Uso de la función `calloc()` para asignar memoria dinámicamente, 546-547

20.3. Uso de realloc() para incrementar el tamaño de bloque de memoria asignada dinámicamente, 548-549
20.4. Uso de free() para liberar memoria asignada en forma dinámica anteriormente, 550-551
20.5. Paso de argumentos de la línea de comandos a main(), 525
20.6. Uso de los operadores de desplazamiento, 555
21.1. Archivo fuente SQUARE.C, , 565
21.2. Archivo fuente CALC.C, 565
21.3. El archivo de encabezado CALC.H para CALC.C, 565
21.4. Uso del operador # en expansión de macros, 573
21.5. Uso de las directivas del preprocesador con archivos de encabezado, 578
R1.1 Archivo fuente week1.c, 160-164
R2.1 Archivo fuente week2.c, 380-387
R3.2 Archivo fuente week3.c, 586-594
listas de argumentos, variable
funciones que las usan, 504-507
herramientas, 504-505
listas de parámetros, 96-99
listas encadenadas
malloc(), función, 275
organización de, 273-274
tipos, 272
localtime(), función, 519
log(), función, 515
log10(), función, 515
long (entero largo), variables, 40, 50
conversión de cadenas a, 486
long, palabra clave, 600
look_up(), función, 594
llamado de funciones, 25, 106-107
con apuntadores a funciones, 413-415
recursión, 107-109
llaves ({ }), 27

M

macros
de función, creación, 570-473
NDEBUG, 525-526
parámetros, 571
predefinidas, 579-580
sustitución, directiva #define, 569-570
va_arg(), 505
va_end(), 505-507
va_start(), 505-507
versus funciones, 503, 573-574
macros de función
creación de, directiva #define, 570-573
versus funciones, 573-574
macros de sustitución, directiva #define, 569-570
macros predefinidas, 579-580
main(), función, 23, 62
argumentos de la línea de comandos, recuperación de, 552-553
variables locales, 287, 292-293
malloc(), función, 222-226, 409, 546
listas encadenadas, 275
MATH.H, archivo de encabezado, prototipos de función, 617-618
mayor que (>), operador, 66
mayor que o igual a (>=), operador, 66
mayúsculas-minúsculas
conversión, 482-483
sensitividad, 38
membresía indirecta (->), operador, 261-266
memoria de la computadora, 36-37
memoria, 36-37, 190-191
espacio de almacenamiento, asignación, 545-551
liberación de, 559
menor que (<), operador, 66
menor que o igual a (<=), operador, 66
mensajes
desplegado de, función puts(), 148-155
de texto, impresión, 140

mensajes de error, list1202.c(17):Error: undefined identifier 'x'., 284
menu(), función, 312
Metaware High C/C++, compilador, 691
Microsoft
paquetes de compilador, 681
Visual C++, compilador, edición estándar, 690-691
Visual C/C++, compilador, edición estándar, 682-685
miembros de estructura, acceso, 243-245, 262
miembros, acceso de
en estructuras, 243-245, 262
en uniones, 267-269
modf(), función, 516
modificadores de tipo, 543-545
modo de posfijo, 59-60
modo de prefijo, 59-60
modos, prefijo versus posfijo, 59-60
módulo (%), operador, 60-62
módulo principal, 564
módulos
compilación, encadenamiento, 566
componentes, 566-567
módulos secundarios, 564
múltiples
archivos, compilación de, 569
enunciados return en funciones, 102-104
indirección, 393
líneas, impresión de, 147
multiplicación (*), operador de, 46, 61
MULTIPLY.C, archivo fuente, 22-23

N

\n secuencia de escape de nueva línea, 141
NDEBUG, macro, 525-526
no (!), operador, 75-77
no igual a (!=), operador, 66
en enunciados if-else, 74
nombres
de arreglo, 176
acceso de cadenas mediante, 221
como apuntadores, 197
de función, 88, 96, 111
de variable, 37-39
nombres de archivo, 427
archivos de encabezado, 581
archivos ejecutables, 581
de archivos temporales, 457-458
especificación de, directivas #include, 575
nombres de arreglo, acceso de cadenas mediante, 221
notación
científica, 45
constantes enteras, 45-46
de camello, 38
notación binaria, 606-607
notación científica, 45
notación de camello, 38
notación hexadecimal, 606-607
nueva línea (\n), carácter, 141
inserción automática de, 227
\n secuencia de escape de nueva línea, 141
NULL, valor de apuntador, 273
numeración de elementos de arreglo, 170
números
cubo, cálculo de, 89
decimales, asignación a enteros, 50
impresión de, 121
negativos, 83
en variables sin signo, 50
números decimales, asignación a enteros, 50
números negativos, 83
en variables sin signo, 50

O

.OBJ, archivos, 568-569
O (||), operador, 75-77
O exclusivo (^), operador, 555-556
O inclusivo (|), operador, 555-556
operador condicional, 80
operador de adición (+), 61
operador de dirección de (&), 150, 153, 174

inicialización de apuntadores, 193
operador de encadenamiento #, definiciones de macro, 572
operador de miembro (.), 255, 261, 267
operador ternario, 80
operadores
a nivel bit, 554
complemento (~), 556
de desplazamiento, 545-555
lógicos, 555-556
AND (&), 555-556
asignación
compuesta, 79-80
promoción de tipo, 543
signo de igual (=), 43, 58
coma (,), 80-81, 121
concatenación (##), macros, 573
condicionales, 80
de adición (+), 61
decremento (—), 59
defined(), 577-578
desplazamiento derecho (>>), 554-555
desplazamiento izquierdo (<<), 554-555
dirección de (&), 150, 153, 174
inicialización de apuntadores, 193
direcciones, olvido de, 155
división (/), 61
doble indirección (**), 392
encadenamiento #, definición de macro, 572
igual (==), 66, 69
O exclusivo (^), 555-556
incremento (++), 59, 263
indirección (*), 192-193, 258-261, 500, 507
lógicos, 75-76
precedencia, 77-78
matemáticos, 58
binarios, 60-62
unitarios, 58-60
mayor que (>), 66
mayor que o igual a (>=), 66
O inclusivo (|), 555-556
membresía indirecta (->), 261, 266
menor que (<), 66
menor que o igual a (<=), 66
miembro (.), 255, 261, 267
módulo (%), 60-62
multiplicación (*), 46, 61
no (!), 75-77
no igual (!=), 66
O (||), 75-77
precedencia, 63-64, 73-75, 604
punto (.), 243
relacional, 65-66, 73-75
creación de enunciados para el control de programa, 67
resta (-), 61
sizeof(), 42, 396
determinación del espacio de almacenamiento, 183-184
unitario, 58-60
versus binario, 82
y (&&), 75-77
operadores a nivel bit, 554
complemento (~), 556
de desplazamiento, 554-555
lógicos, 555-556
operadores binarios
matemáticos, 60-62
versus unitarios, 82
operadores de asignación
compuesto, 79-80
promoción de tipo, conversión de, 543
signo de igual (=), 43, 58
operadores de asignación compuestos, 79-80
operadores de desplazamiento, 554
operadores lógicos, 75-76
a nivel bit, 555-556
precedencia, 77-78
operadores matemáticos unitarios, 58
versus binarios, 82
operadores matemáticos, 58
binarios, 60-62
precedencia, 63
unitarios, 58-60
operadores relacionales, 65-66
creación de instrucciones para el control de programa, 67
precedencia, 73-75

orden de evaluación de subexpresiones, 65
ordenamiento
  cadenas, 534-537
  con la función qsort(), 530-537
  control del orden, 417-421
  entrada de teclado, 405-411

P

palabra clave
  asm, 600
  auto, 600
  break, 600
  case, 600
  char, 600
  const, 47-50, 600
  continue, 600
  default, 600
  do, 600
  double, 600
  else, 600
  enum, 600
  extern, 286-287, 567, 600
  float, 600
  for, 600
  goto, 600
  if, 600
  int, 600
  long, 600
  register, 291-292, 600
  return, 102, 600
  short, 601
  signed, 601
  sizeof, 601
  static, 601
  struct, 244-245, 255, 601
  switch, 601
  typedef, 43, 275-276, 601
    versus etiquetas de estructura, 277
  union, 269-270, 601
  unsigned, 601
  void, 601
  volatile, 601
  while, 601
palabras reservadas, 600-601
pantallas
  desplegado de datos de cadena en, 227
    printf(), función, 228
    puts(), función, 227-228
  desplegado de información
    printf(), función, 140-148
parámetros
  función
    alcance, 290
    declaración de, 500
  macro, 571
  versus argumentos, 96-99
parámetros de función
  alcance, 290
  declaración de, apuntadores tipo void, 500
paréntesis, precedencia de operadores, 64
paso de
  apuntadores, 510
    a funciones como argumentos, 416-417
  argumentos a funciones, 105
  arreglos
    a funciones, 206-210
    de apuntadores, 404-405
    multidimensionales, 399-401
  estructuras como argumentos de funciones, 265-266
pasos desde el código fuente hasta el archivo ejecutable, 9
perror(), función, 335, 527-529
portabilidad, 5, 680
pow(), función, 516
Power C, compilador, 691
precedencia (operadores), 63-64, 604
  lógica, 77-78
  relacional, 73-75
precisión, de variables, 40
preprocesador, 569
print_function(), 272
print_report(), 144
print_strings(), 405
print_value(), 282-284

printarray_1(), 401
printarray_2(), 401
printf(), 24-25, 62, 140-148, 217, 228, 282, 334-335, 362-369
  cadenas para formateo, 363
  especificadores de conversión, 145-147, 364
  secuencia de escape, 142-144
  sintaxis, 147-148
printf(), función, 335
product(), función, 25
PROG.H, archivo de encabezado, 578
programa de lista de teléfonos (demostrador de arreglo de estructuras), 251-255
programación
  con varios archivos fuente, véase
programación modular
  estructurada, 93
    árbol, 95
    importancia del alcance, 284
    planeación, 93-95
    ventajas, 93
  instrucciones de control, 136
  jerárquica, 94-95
  modular, 564
    técnicas, 564-566
    variables externas, 567-568
    ventajas, 564
  módulos, componentes, 566-567
  preparación, 5-6
programación estructurada, 93
  estructura de árbol, 95
  importancia del alcance, 284
  planeación, 93-95
  ventajas, 93
programación jerárquica, 94-95
programación modular, 564
  técnicas, 564-566
  variables externas, 567-568
  ventajas, 564
programas
  compilación, errores de compilación, 13-15
  componentes, 23-28
  ejecución, 10, 13, 27
  peso/año de nacimiento, 48
  salida de, 321-324
promoción de tipo, conversión por asignación, 543
prototipos (de función), 24, 90-92, 104, 620-626
  ubicación, 110
prototipos de función, 24, 90-92, 104, 620-626
  ASSERT.H, archivo de encabezado, 618
  CTYPE.H, archivo de encabezado, 616-617
  MATH.H, archivo de encabezado, 617-618
  STDARG.H, archivo de encabezado, 618
  STDIO.H, archivo de encabezado, 612-614
  STDLIB.H, archivo de encabezado, 610-611
  STRING.H, archivo de encabezado, 615-616
  TIME.H, archivo de encabezado, 614
  ubicación de, 110
prueba de caracteres, 487-491
punto (.), operador, 243
punto flotante de precisión sencilla, véase variables flotantes
punto y coma
  al final de enunciados if, 67
  en enunciados, 54-55
putc(), función, 361, 437
putchar(), función, 335, 359-361
puts(), función, 103, 152, 227-228, 334-335, 361-362
  mensajes, desplegado de, 148
  sintaxis, 148-149
  versus la función printf(), 148, 154

Q

qsort(), función, 530-537
QuickC, compilador (Microsoft), 681
quicksort, algoritmo, 530-537

R

RAM (memoria de acceso aleatorio), 36-37, 190-191
ramificación de la ejecución del programa, 306-309
rand(), función, 182
RANDOM.C, archivo fuente, 180
rango aproximado, variables numéricas, 40
rangos
  aproximados de variables numéricas, 40
  código ASCII, 218
  variables char (carácter), 218
realloc(), función, 547-549, 559
rectángulos, 245-248
recursión, 107-109
  en funciones de factorial, 111
redirección, 369-371
referencia, paso por, 496-500
register, palabra clave, 291-292, 600
registro, clase de almacenamiento, 293
remove(), función, 452-453
rename(), función, 453-454
renombrado de archivos de disco, 453-454
resta (-), operador, 61
return, enunciado, 90
  múltiple, 102-104
return, palabra clave, 102, 600
reverse(), función, 420
rewind(), función, 444-446
rutas explícitas, archivos de encabezado, 581
rutas, explícitas, 581

S

%s, especificador de conversión de cadenas de caracteres, 145-148
salida a pantalla, 359-369
salida, 332
  archivos
    directos, 438-441
    formateados, 362-369, 432-434
    de caracteres, 359-361, 437
    dispositivos, 332
    redirección, 369-371
    véase también E/S
scanf(), función, 25, 62, 149-153, 174, 232-236, 334-335, 351-354
  caracteres extra, 354-356
  operador de direcciones, 155
  programa de ejemplo, 356-359
  sintaxis, 153-154
SECONDS.C, archivo fuente, 61-62
secuencia de escape para retroceso, 141
secuencias de escape, 141
  función printf(), 142-144
secuencias de escape, 141
  función printf(), 142-144
%s, especificador de conversión de cadenas de caracteres, 145-148
SET, comando (DOS), 575
short, palabra clave, 601
signed, palabra clave, 601
sin(), función, 515
sinh(), función, 516
sintaxis
  do...while, enunciado, 133-134
  for, enunciado, 122-123
  gets(), función, 232
  goto, enunciado, 306
  malloc(), función, 223-224
  printf(), función, 147-148
  puts(), función, 148-149
  while, enunciado, 127-128
sistemas operativos, comandos, 325-326
sizeof(), operador, 42, 396
  espacio de almacenamiento, determinación de, 183-184
sizeof, palabra clave, 601
SIZEOF.C, archivo fuente, 41
sleep(), función, 312
sobrecarga de expresiones, 65
sort(), función, 405-411, 417-421
sqrt(), función, 516
SQUARE.C, archivo fuente, 565-566

static, palabra clave, 601
STDARG.H, archivo de encabezado, 504
  prototipos de función, 618
stdaux (auxiliar estándar), flujo, 334
stderr (error estándar), flujo, 334, 371-372
stdin (entrada estándar), flujo, 334
  caracteres extra, 354-356
  líneas, lectura, 349
  redirección, 369-371
STDIO.H, archivo de encabezado, 153-154
  prototipos de función, 612-614
  puts(), función, 148
STDIO.H, archivo de encabezado, 229
STDLIB.H, archivo de encabezado, 182, 546
  prototipos de función, 610-611
stdout (salida estándar), flujo, 334
  redirección, 369-371
  replicado de caracteres a, 340
stdprn (impresora estándar), flujo, 334, 372-373
strcat(), función, 469-471
strcmp(), función, 473-474
strcmpi(), función, 475
strcmpl(), función, 475
strcpy(), función, 252, 409, 465-467
strcspn(), función, 478-479
strchr(), función, 476-477
strdup(), función, 468-469
stricmp(), función, 475
_stricmp(), función, 475
STRING.H, archivo de encabezado, prototipo de función, 615-616
strlen(), función, 409, 464-465
strlwr(), función, 482-483
strncat(), función, 471-472
strncmp(), función, 475-476
strncpy(), función, 467-468
strnset(), función, 484-485
strpbrk(), función, 480
strrchr(), función, 478
strrev(), función, 483-485
strset(), función, 484-485
strspn(), función, 479-480
strstr(), función, 481-482
struct, palabra clave, 244-245, 255, 601
strupr(), función, 482-483
subexpresiones, orden de evaluación, 65
subíndices, 170
  fuera de rango, 185
  permitidos, 172
switch, enunciado, 312-320
  versus ciclos anidados, 327
switch, palabra clave, 601
Symantec de Zortech, compilador, 691-692
system(), función, 325-326

T

\t secuencia de escape de tabulador horizontal, 141
tablero de ajedrez, creación de con arreglos bi-dimensionales, 175
tamaño
  arreglos, 182-184
  elementos de arreglo, 182, 396
  uniones, 277
tan(), función, 515
tanh(), función, 516
teclado
  entrada
    aceptación, 336-359, 405-411
    desplegado, 405-411
    ordenamiento, 405-411
    teclas especiales, 341-351
  lectura de cadenas desde, 229
    función gets(), 229-232
    función scanf(), 232-234
teclas extendidas
  constantes simbólicas, 343-345
  lectura de, 342
terminación de
  ciclos, 302-306
  programas, 321-324
terminación de programas, 321-324
texto
  entrada desde archivos de disco, 349-351

flujos, 334, 426-427
mensajes, impresión, 140
varias líneas, impresión, 147
ThinkC, compilador (Symantec), 680, 691
tiempo
cálculo de diferencias, 521-522
desplegado, 519-521
obtención de, 518-519
representaciones, 518
conversión entre, 519
__TIME__, macro predefinida, 579-580
time(), función, 518-519
TIME.H, archivo de encabezado, 518
prototipos de función, 614
tipo de regreso de la función, 6
tipos de datos
char (carácter), 216-217
numérico, 40
entrada de, 149-154
requerimientos de espacio de almacenamiento, 183-184
tipos de datos numéricos, determinación de los requerimientos de almacenamiento, 183-184
tipos de retorno de función, 96
tmpnam(), función, 457-458
\t secuencia de escape de tabulador horizontal, 141
Turbo C++ para Windows (Borland), 681
Turbo C/C++ para DOS, compilador (Borland), 680-681, 689-690
instalación, 685-688
typedef, palabra clave, 43, 275-276, 601
versus etiquetas de estructura, 277

U

%u entero decimal sin signo, 145-148
ubicación dinámica de memoria, 545-551
%u entero decimal sin signo, 145-148
UNARY.C, archivo fuente, 59-60
#undef, directiva, 579
ungetc(), función, 348
union, palabra clave, 269-270, 601
uniones
acceso de los miembros, 267-269
definición/declaración, 267
inicialización de miembros, 272
tamaño, 277
versus estructuras, 267
unsigned char (carácter sin signo), variables, 40
unsigned int (entero sin signo), variables, 40
unsigned long (entero largo sin signo), variables, 40
unsigned short (entero corto sin signo), variables, 40
unsigned, palabra clave, 601

V

va_arg(), macro, 505
va_end(), macro, 505-507
va_list, apuntador, 505
va_start(), macro, 505-507
vaciado de archivos de disco, 441-443
valor, pasado por argumentos a funciones, 496-500
valores
cierto/falso, 76-77
números negativos, 83
numéricos, desplegado, 146
regresos de funciones, 102-103, 110
valores cierto/falso, 76-77
números negativos, 83
variables cortas (enteros cortos), 40
variables de punto flotante, 39
variables de registro, 291-292
variables dobles (punto flotante de doble precisión), 40
conversión de cadenas a, 486-487
variables enteras con signo, 41
variables estáticas
externas, 291
locales versus automáticas, 287-290

variables externas, 284-285
alcance, 285
cuándo usar, 285-286
estáticas, 291
extern, palabra clave, 286-287
programación modular, 567-568
variables flotantes (de punto flotante precisión sencilla)
variables globales
inicialización, 292
uso de memoria, 296
véase también variables externas
versus locales, 296
variables locales automáticas
pérdida de valor, 290
versus estáticas, 287-290
variables locales, 99-101 ,287
cuándo se usa, 286
definición dentro de bloques, 294-295
estáticas versus automáticas, 287-290
función main(), 292-293
inicialización, 292
versus globales, 296
variables numéricas, 39-42
conversión de cadenas a, 485-487
desplegado de, 146
inicialización, 43-44
variables, 37
alcance, 282
demostración de, 282-284
almacenamiento de direcciones, 191
char, (carácter), 40, 216-219
arreglos de apuntadores a, 403-404
rangos, 218
de registro, 291-292
declaración, 42-43, 190
definiciones, 24
double (punto flotante de doble precisión), 40
conversión de cadenas a, 486-487
externas, 284-285
alcance, 285
cuándo usarlas, 285-286
estáticas, 291
extern, palabra clave, 286-287
programación modular, 567-568
float (punto flotante de precisión sencilla), 39
globales, uso de memoria, 296
int (enteras), 39-40
conversión de cadenas a, 485-486
locales, 99-101, 287
cuándo usarlas, 286
definición dentro de bloques, 294-295
estáticas versus automáticas, 287-290
función main(), 292-293
long (entero largo), 40, 50
conversión de cadenas a, 486
nombres, 37-39
numéricas, 39-42
conversión de cadenas a, 485-487
inicialización, 43-44
short (entero corto), 40
unsigned char (carácter sin signo), 40
unsigned int (entero sin signo), 40
unsigned long (entero largo sin signo), 40
unsigned short (entero corto sin signo), 40
uso en funciones, reglas, 101
Visual C++, compilador (Microsoft), edición estándar, 690-691
Visual C/C++ (Microsoft)
edición estándar, 680-681
edición profesional, 681
instalación, 682-685
void, apuntadores, 500-504
conversión explícita de tipo, 545
void, palabra clave, 601
volatile, palabra clave, 601

W

Watcom C, compilador, 691
week1.c, archivo fuente, 160-164
week2.c, archivo fuente, 380-387

week3.c, archivo fuente, 586-594
`while`, enunciado, 125-127
  anidado, 128-130
  sintaxis, 127-128
`while`, palabra clave, 601

X -Y

y (&&), operador, 75-77

Z

Zortech, compilador (Symantec), 691

Consulta rápida de C

Los siguientes identificadores son palabras clave reservadas del C. No deben usarse para ningún otro objeto en un programa C. Sin embargo, están permitidas cuando estén entre comillas.

Palabra clave	Descripción
`asm`	Palabra clave del C que indica código de lenguaje ensamblador en línea.
`auto`	La clase de almacenamiento por omisión.
`break`	Comando del C que ocasiona salida incondicional de los enunciados `for`, `while`, `switch` y `do...while`.
`case`	Comando del C usado con el enunciado `switch`.
`char`	El tipo de dato de C más simple.
`const`	Modificador de datos del C que impide que una variable sea cambiada. Véase `volatile`.
`continue`	Comando del C que ocasiona la siguiente iteración de un enunciado `for`, `while` o `do...while`.
`default`	Comando del C usado dentro del enunciado `switch` para atrapar cualquier instancia no especificada dentro de un enunciado `case`.
`do`	Comando del C para hacer ciclos, usado junto con el enunciado `while`. El ciclo siempre se ejecutará por lo menos una vez.
`double`	Tipo de datos del C que puede guardar valores de punto flotante de doble precisión.
`else`	Enunciado que indica enunciados alternos que deberán ser ejecutados cuando un enunciado `if` evalúe a falso.
`enum`	Tipo de datos del C que permite que sean declaradas variables que aceptan solamente determinados valores.
`extern`	Modificador de datos del C que indica que una variable será declarada en otra área del programa.
`float`	Tipo de datos del C, para los números de punto flotante.
`for`	Comando del C para hacer ciclos, que contiene secciones de *inicialización,* de *incremento* y de *condición.*
`goto`	Ocasiona un salto a una etiqueta predefinida.
`if`	Comando del C para cambiar el flujo del programa basándose en una decisión de CIERTO/FALSO.
`int`	Tipo de datos del C para guardar valores enteros.
`long`	Tipo de datos del C para guardar valores enteros más grandes que `int`.
`register`	Modificador de almacenamiento que especifica que una variable debe ser guardada en un registro, en caso de ser posible.
`return`	Comando del C que hace que el flujo del programa salga de la función actual y regrese a la función llamadora. También se usa para regresar un solo valor.
`short`	Tipo de datos del C para guardar enteros. No se usa comúnmente, y es del mismo tamaño que un `int` en la mayoría de las computadoras.
`signed`	Modificador del C para indicar que una variable tiene valores tanto positivos como negativos.
`sizeof`	Operador del C que regresa el tamaño (cantidad de octetos) de un concepto.
`static`	Modificador del C para indicar que el compilador debe guardar el valor de la variable.
`struct`	Palabra clave del C para combinar variables del C de cualquier tipo de dato en un grupo.
`switch`	Comando del C para cambiar el flujo del programa en varias direcciones. Se emplea junto con el enunciado `case`.
`typedef`	Modificador del C para crear nuevos nombres para tipos existentes de variables y funciones.
`union`	Palabra clave del C para permitir que varias variables compartan el mismo espacio de memoria.
`unsigned`	Modificador del C para indicar que una variable contendrá solamente valores positivos. Véase `signed`.
`void`	Palabra clave del C para indicar que una función no regresa nada o que un apuntador en uso es considerado genérico (capaz de apuntar a cualquier tipo de dato).
`volatile`	Modificador del C que indica que una variable puede ser cambiada. Véase `const`.
`while`	Enunciado del C para hacer ciclos, que ejecuta una sección de código mientras una condición se mantenga CIERTA.

Consulta rápida de C

Los tipos de datos numéricos del C

Tipo de variable	Palabra clave	Bytes requeridos	Rango
carácter	char	1	-128 a 127
entero	int	2	-32,768 a 32,767
entero corto	short	2	-32,768 a 32,767
entero largo	long	4	-2,147,483,648 a 2,147,483,647
carácter sin signo	unsigned char	1	0 a 255
entero sin signo	unsigned int	2	0 a 65,535
entero corto sin signo	unsigned short	2	0 a 65,535
entero largo sin signo	unsigned long	4	0 a 4,294,967,295
punto flotante de precisión sencilla	float	4	1.2E-38 a 3.4E38[1]
punto flotante de doble precisión	double	8	2.2E-308 a 1.8E308[2]

[1] Rango aproximado; precisión = 7 dígitos. *[2] Rango aproximado; precisión = 19 dígitos.*

Los operadores del C

Matemáticos

+ Suma	Suma sus dos operandos	x + y
- Resta	Resta el segundo operando del primero	x - y
* Multiplicación	Multiplica sus dos operandos	x * y
/ División	Divide el primer operando por el segundo operando	x / y
% Módulo	Da el residuo cuando el primer operando es dividido por el segundo operando	x % y
++ Incremento	Suma 1	x++, ++x
-- Decremento	Resta 1	x--,--x

Relacionales

(Falso es igual a 0, Cierto es igual a 1)

== Igual	¿Es el operando 1 igual al operando 2?	x == y
> Mayor que	¿Es el operando 1 mayor que el operando 2?	x > y
< Menor que	¿Es el operando 1 menor que el operando 2?	x < y
>= Mayor que o igual a	¿Es el operando 1 mayor que o igual al operando 2?	x >= y
<= Menor que o igual a	¿Es el operando 1 menor que o igual al operando 2?	x <= y
!= Diferente de	¿Es el operando 1 diferente al operando 2?	x != y

Lógicos

&&	Y	Es cierto solamente si la expresión 1 y la expresión 2 son ciertas; en caso contrario es falso	(exp1 && exp2)
||	O	Es cierto si la expresión 1 o la expresión 2 son ciertas y falso solamente cuando ambas son falsas	(exp1 || exp2)
!	No	Es falso (0) si la expresión 1 es cierta, y es cierta si la expresión 1 es falsa	(!exp1)

Condicionales

?:	Condicional (if/then/else)	Si la expresión 1 es cierta, la expresión completa evalúa a la expresión 2 y, en caso contrario, la expresión completa evalúa a la expresión 3	exp1 ? exp2 : exp3

Compuestos

+=	Suma compuesta	x = x + y es equivalente a x += y
-=	Resta compuesta	x = x - y es equivalente a x -= y
*=	Multiplicación compuesta	x = x * y es equivalente a x *= y
/=	División compuesta	x = x / y es equivalente a x /= y
%=	Módulo compuesto	x = x % y es equivalente a x %= y

JUL

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

2001

Cuadros de Sintaxis

A lo largo del libro los cuadros de sintaxis explican conceptos de C. A continuación se presenta una lista de algunos de los cuadros y el lugar donde se encuentran.

El enunciado break	304	La función malloc()	223
El ciclo de desarrollo del C	10	La función printf()	147
El enunciado continue	306	La función puts()	148
El enunciado do...while	133	La función scanf()	153
El enunciado for	122	La palabra clave struct	244
Funciones	91	El enunciado switch	319
La función gets()	232	La palabra clave union	269
El enunciado if	71	El enunciado while	127

A continuación se muestra un cuadro de sintaxis de ejemplo:

Sintaxis

La función *printf()*

```
#include <stdio.h>
printf( cadena de formato[,argumentos,...]);
```

`printf()` es una función que acepta una serie de *`argumentos`*, donde a cada uno se le aplica un *especificador* de conversión en la cadena de formateo dada. `printf()` imprime la información formateada en el dispositivo estándar de salida, que, por lo general, es la pantalla. Cuando se usa `printf()` se necesita incluir el archivo de encabezado de la entrada/salida estándar, STDIO.H.

La cadena de formato es imprescindible. Sin embargo, los argumentos son opcionales. Para cada argumento debe haber un especificador de conversión. La tabla 7.2 lista los más comunes.

La cadena de formato también puede contener secuencias de escape. La tabla 7.1 lista las más usadas.

A continuación se presentan ejemplos de llamadas a `printf()` y su salida:

Ejemplo 1

```
#include <stdio.h>
main()
{
    printf( "¡Este es un ejemplo de algo impreso!");
}
```

Despliega

```
¡Este es un ejemplo de algo impreso!
```

Ejemplo 2

```
printf( "Esto imprime un carácter, %c\n un número, %d\n un punto flotante, %f",
'z', 123, 456.789 );
```

Despliega

```
Esto imprime un carácter, z
un número, 123
un punto flotante, 456.789
```

Aprendiendo C en 21 días

Revisado por la Indianapolis Computer Society

- **Las secciones de preguntas y respuestas, de talleres y de "Revisión de la semana" proporcionan repasos integrales de todo lo que usted va aprendiendo.**
- **Trata todos los temas fundamentales del C: funciones, estructuras, apuntadores, manejo de memoria, E/S de archivos, bibliotecas de funciones, etcétera.**
- **El formato autodidáctico le facilita el aprendizaje aun de los conceptos más difíciles.**

GARANTIZADO ACTUAL Y PRECISO
Precisión técnica aprobada por Scott A. Parker y Tim Moore

C es la norma. Ahora puede usted aprender esta programación esencial más rápidamente que nunca con esta edición de Aprendiendo C en 21 días.

Veintiún lecciones fáciles de estudiar le enseñan, día a día, los conceptos vitales para la escritura de programas en C. Comenzará con una explicación básica de los componentes del C, y en unos cuantos días continuará con la escritura y la depuración de sus propios programas.

Se sentirá a gusto con lo básico mediante una variedad de métodos efectivos. Los cuadros de sintaxis proporcionan útiles referencias, con ejemplos cortos sobre el uso de los enunciados. Las secciones de Debe y No Debe ofrecen sugerencias e indican peligros que se deben evitar. La salida y el análisis línea por línea que se dan a continuación de cada listado le ayudan a comprender los temas vitales para la correcta programación. Además, cada capítulo concluye con una serie de ejercicios para reforzar y probar lo aprendido.

Aprenda a programar en C en poco tiempo con esta útil obra.

CENTRO DE INFORMACION
Principiantes/Intermedios
Enseñanza práctica
Compatible con IBM
Programación
Compatible con todos los compiladores C ANSI

Visítenos en:
www.pearsonedlatino.com

PEARSON
PRENTICE HALL

ISBN-968-880-444-4
9 789688 804445 90000