Tony Zhang

Aprendiendo® C en 24 horas

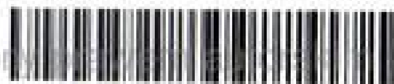
PEARSON EDUCACION DE MEXICO
EJEMPLAR PARA EVALUACION
PROHIBIDA SU VENTA

TRADUCCIÓN:
Sergio Kourchenko Barrena

REVISIÓN TÉCNICA:
Maricela Quintana López
Maestra en Ciencias Computacionales

Pearson Educación

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


Material chroniony
6KQ5-8HJ-AA3E

Resumen de contenido

Introducción 1

Parte I Los fundamentos de C 9

Hora 1 El primer paso 11
2 Su primer programa de C 27
3 La estructura de un programa de C 41
4 Tipos de datos y palabras reservadas 55
5 Manejo de la entrada y salida estándar 71

Parte II Operadores e instrucciones de control de flujo 89

Hora 6 Manejo de datos 91
7 Ciclos 105
8 Uso de operadores condicionales 121
9 Modificadores de datos y funciones matemáticas 141
10 Control de flujo del programa 155

Parte III Apuntadores y arreglos 173

Hora 11 Apuntadores 175
12 Arreglos 189
13 Cadenas 207
14 Alcance y clases de almacenamiento 223

Parte IV Funciones y asignación dinámica de memoria 241

Hora 15 Funciones 243
16 Uso de apuntadores 259
17 Asignación de memoria 279
18 Tipos de datos y funciones especiales 295

Materiał chroniony prawem autorskim

Parte V Estructuras, uniones, E/S de archivos y más 311

Hora 19 Estructuras de datos 313
20 Uniones 333
21 Lectura y escritura de archivos 355
22 Funciones de archivo especiales 373
23 Compilación: el preprocesador de C 391
24 ¿Qué sigue después? 409

Parte VI Apéndices 437

Apéndice A Archivos de encabezado del estándar ANSI 439
B Respuestas a los cuestionarios y ejercicios 441
Índice 503

Materiał chroniony prawem autorskim

Contenido

Introducción 1

¿Quiénes deben leer este libro? ... 1
Características especiales de este libro ... 1
Ejemplos de programación ... 2
Preguntas, respuestas y taller ... 4
Convenciones utilizadas en este libro ... 4
Lo que aprenderá en 24 horas ... 4

Part I Los fundamentos de C 9

Hora 1 El primer paso 11

Qué es C ... 12
El estándar ANSI de C ... 15
Suposiciones acerca del lector ... 16
Configuración de su sistema ... 16
Hardware ... 16
Software ... 16
Muestra de un sistema configurado para programar en C ... 17
Uso del compilador de Microsoft ... 18
Uso del compilador de Borland ... 21
Resumen ... 24
Preguntas y respuestas ... 25
Taller ... 25
Cuestionario ... 25

Hora 2 Su primer programa de C 27

Un programa sencillo de C ... 28
Comentarios ... 29
La directiva #include ... 31
Archivos de encabezado ... 32
Paréntesis angulares (< >) y comillas dobles (" ") ... 32
La función `main()` ... 33
El carácter de nueva línea (`\n`) ... 33
La instrucción `return` ... 34
La función `exit()` ... 34
Compilación y enlace ... 34
¿Qué es lo que está mal en mi programa? ... 36
Depuración de su programa ... 37

Materiał chroniony prawem autorskim

Resumen 37
Preguntas y respuestas 38
Taller 38
Cuestionario 38
Ejercicios 39

Hora 3 La estructura de un programa de C 41

Los elementos básicos de un programa de C 42
Constantes y variables 42
Expresiones 42
Operadores 43
Bloques de instrucciones 45
Anatomía de una función de C 46
Determinación del tipo de una función 46
Cómo asignar un nombre válido a una función 47
Paso de argumentos a las funciones de C 47
El principio y el final de una función 48
El cuerpo de la función 48
Cómo hacer llamadas a funciones 49
Resumen 51
Preguntas y respuestas 52
Taller 52
Cuestionario 52
Ejercicios 53

Hora 4 Tipos de datos y palabras reservadas 55

Palabras reservadas de C 56
El tipo de datos `char` 57
Variables de tipo carácter 58
Constantes de carácter 58
El carácter de escape (\) 59
Impresión de caracteres 60
El tipo de datos `int` 62
Declaración de variables enteras 62
Cómo mostrar los valores numéricos de los caracteres 63
El tipo de datos `float` 64
Declaración de variables de punto flotante 64
El especificador de formato de punto flotante (`%f`) 65
El tipo de datos double 67
Uso de la notación científica 67
Denominación de una variable 68
Resumen 68
Preguntas y respuestas 68

Materiał chroniony prawem autorskim

Taller 69
Cuestionario 69
Ejercicios 70

Hora 5 Manejo de la entrada y salida estándar 71

La entrada y salida estándar 72
Obtención de entrada del usuario 72
Uso de la función getc() 72
Uso de la función getchar() 74
Impresión de salida en la pantalla 75
Uso de la función putc() 75
Otra función para la escritura: putchar() 77
Nueva visita a la función printf() 78
Conversión a números hexadecimales 79
Especificación del ancho mínimo del campo 81
Alineación de la salida 83
Uso del especificador de precisión 84
Resumen 85
Preguntas y respuestas 86
Taller 86
Cuestionario 87
Ejercicios 87

Parte II Operadores e instrucciones de control de flujo 89

Hora 6 Manejo de datos 91

Los operadores de asignación aritmética 92
El operador de asignación (=) 92
Combinación de operadores aritméticos con = ...92
Obtención de negaciones de valores numéricos 95
Incremento o decremento en una unidad 96
Mayor que o menor que 98
Uso del operador de conversión explícita 101
Resumen 102
Preguntas y respuestas 102
Taller 103
Cuestionario 103
Ejercicios 103

Hora 7 Ciclos 105

El ciclo while 106
El ciclo do-while 107
Ciclos bajo la instrucción for 109

Materiał chroniony prawem autorskim

La instrucción nula 112
Uso de expresiones complejas en una instrucción `for` 113
Uso de ciclos anidados 116
Resumen 118
Preguntas y respuestas 118
Taller 119
Cuestionario 119
Ejercicios 120

Hora 8 Uso de operadores condicionales 121

Cómo medir el tamaño de los datos 122
Todo es lógico 124
El operador lógico AND (`&&`) 124
El operador lógico OR (`||`) 126
El operador lógico NOT (`!`) 128
Manejo de bits 129
Conversión de un número decimal a hexadecimal o binario 129
Uso de operadores a nivel de bits 130
Uso de operadores de desplazamiento 133
¿Qué significa `x?y:z`? 135
Resumen 137
Preguntas y respuestas 137
Taller 138
Cuestionario 138
Ejercicios 138

Hora 9 Modificadores de datos y funciones matemáticas 141

Cómo habilitar o inhabilitar el bit de signo 142
El modificador `signed` 142
El modificador `unsigned` 143
Modificación del tamaño de los datos 145
El modificador `short` 145
El modificador `long` 145
Cómo agregar `h`, `l` o `L` a los especificadores de formato `printf` y `fprintf` 147
Funciones matemáticas en C 148
Llamadas a `sin()`, `cos()` y `tan()` 149
Llamadas a `pow()` y `sqrt()` 150
Resumen 152
Preguntas y respuestas 153
Taller 154
Cuestionario 154
Ejercicios 154

Materiał chroniony prawem autorskim

Hora 10 Control de flujo del programa 155

Cómo decir siempre "si…" 156
La instrucción if-else 158
Instrucciones if anidadas 160
La instrucción switch 161
La instrucción break 164
Interrupción de un ciclo infinito 166
La instrucción continue 167
La instrucción goto 168
Resumen 170
Preguntas y respuestas 170
Taller 171
Cuestionario 171
Ejercicios 172

Parte III Apuntadores y arreglos 173

Hora 11 Apuntadores 175

Qué es un apuntador 176
Dirección (valor izquierdo) contra contenido (valor derecho) 176
El operador de dirección (&) 177
Declaración de apuntadores 179
El operador de indirección (*) 182
Apuntadores nulos 183
Actualización de variables por medio de apuntadores 183
Cómo apuntar a la misma ubicación de memoria 184
Resumen 186
Preguntas y respuestas 187
Taller 188
Cuestionario 188
Ejercicios 188

Hora 12 Arreglos 189

Qué es un arreglo 190
Declaración de arreglos 190
Indexación de arreglos 190
Inicialización de arreglos 191
El tamaño de un arreglo 192
Arreglos y apuntadores 194
Cómo desplegar arreglos de caracteres 196
El carácter nulo ('\0') 198
Arreglos multidimensionales 199

Materiał chroniony prawem autorskim

Arreglos sin especificación de tamaño 201
Resumen 203
Preguntas y respuestas 203
Taller 204
Cuestionario 204
Ejercicios 205

Hora 13 Cadenas 207

Declaración de cadenas 208
Qué es una cadena 208
Inicialización de cadenas 208
Constantes de cadena en comparación con constantes de carácter 209
¿Cuánto mide una cadena? 212
La función `strlen()` 212
Cómo copiar cadenas con `strcpy()` 213
Lectura y escritura de cadenas 215
Las funciones `gets()` y `puts()` 215
Uso de `%s` con la función `printf()` 217
La función `scanf()` 217
Resumen 219
Preguntas y respuestas 220
Taller 221
Cuestionario 221
Ejercicios 221

Hora 14 Alcance y clases de almacenamiento 223

Cómo ocultar datos 224
Alcance de bloque 224
Alcance de bloque anidado 225
Alcance de función 226
Alcance de programa 227
Los especificadores de clases de almacenamiento 229
El especificador `auto` 229
El especificador `static` 230
El alcance de archivo y la jerarquía de los alcances 232
El especificador `register` 233
El especificador `extern` 233
Los modificadores de clase de almacenamiento 234
El modificador `const` 234
El modificador `volatile` 235
Resumen 236
Preguntas y respuestas 237

Materiał chroniony prawem autorskim

Taller 238
Cuestionario 238
Ejercicios 239

Parte IV Funciones y asignación dinámica de memoria 241

Hora 15 Funciones 243

Declaración de funciones 244
Declaración en comparación con definición 244
Especificación de los tipos de retorno 244
Uso de prototipos 245
Cómo hacer llamadas a las funciones 245
Prototipos de funciones 247
Funciones sin argumentos 248
Uso de `time()`, `localtime()` y `asctime()` 249
Funciones con un número fijo de argumentos 251
Prototipo de un número variable de argumentos 251
Procesamiento de argumentos variables 252
Comprensión de la programación estructurada 255
Resumen 255
Preguntas y respuestas 256
Taller 257
Cuestionario 257
Ejercicios 257

Hora 16 Uso de apuntadores 259

Aritmética de los apuntadores 259
El tamaño escalar de los apuntadores 260
Resta de apuntadores 263
Apuntadores y arreglos 264
Acceso de arreglos mediante apuntadores 264
Apuntadores y funciones 266
Paso de arreglos a funciones 266
Paso de apuntadores a funciones 268
Paso de arreglos multidimensionales como argumentos 270
Arreglos de apuntadores 272
Apuntadores a funciones 274
Resumen 276
Preguntas y respuestas 276
Taller 277
Cuestionario 277
Ejercicios 278

Materiał chroniony prawem autorskim

Hora 17 Asignación de memoria 279

Asignación de memoria en tiempo de ejecución 280
La función `malloc()` 280
Uso de `free()` para liberar memoria asignada 283
La función `calloc()` 286
La función `realloc()` 288
Resumen 291
Preguntas y respuestas 292
Taller 293
  Cuestionario 293
  Ejercicios 294

Hora 18 Tipos de datos y funciones especiales 295

El tipo de datos `enum` 296
  Declaración del tipo de datos `enum` 296
  Asignación de valores a nombres `enum` 296
Cómo hacer definiciones `typedef` 300
Por qué usar `typedef` 300
Funciones recursivas 303
Otro vistazo a la función `main()` 305
  Argumentos de la línea de comandos 305
  Recepción de argumentos de la línea de comandos 306
Resumen 308
Preguntas y respuestas 308
Taller 309
  Cuestionario 309
  Ejercicios 310

Parte V Estructuras, uniones, E/S de archivos y más 311

Hora 19 Estructuras de datos 313

Qué es una estructura 314
  Declaración de estructuras 314
  Definición de variables de estructuras 315
Cómo hacer referencia a miembros de estructuras con el operador de punto 315
Inicialización de estructuras 317
Estructuras y llamadas a funciones 319
  Cómo hacer referencia a estructuras mediante apuntadores 322
  Cómo hacer referencia a un miembro de una estructura mediante -> 324
Arreglos de estructuras 324
  Estructuras anidadas 327
Resumen 330

Materiał chroniony prawem autorskim

Preguntas y respuestas 330
Taller 331
Cuestionario 331
Ejercicios 332

Hora 20 Uniones 333

Qué es una unión 334
Declaración de uniones 334
Definición de variables de unión 334
Cómo hacer referencia a una unión con . o -> 335
Uniones en comparación con estructuras 337
Inicialización de una unión 337
El tamaño de una unión 339
Uso de uniones 341
Cómo hacer referencia a la misma ubicación de memoria con diferentes miembros 341
Cómo hacer flexibles las estructuras 343
Definición de campos de bits mediante `struct` 347
Resumen 350
Preguntas y respuestas 351
Taller 352
Cuestionario 352
Ejercicios 353

Hora 21 Lectura y escritura de archivos 355

Archivos en comparación con flujos 356
Qué es un archivo 356
Qué es un flujo 356
E/S en búferes 356
Los aspectos básicos de la E/S de archivos en disco 357
Apuntadores de FILE 357
Cómo abrir archivos 357
Cómo cerrar archivos 358
Lectura y escritura de archivos en disco 360
Un carácter a la vez 360
Una línea a la vez 363
Un bloque a la vez 366
Resumen 370
Preguntas y respuestas 370
Taller 371
Cuestionario 371
Ejercicios 372

Materiał chroniony prawem autorskim

Hora 22 Funciones de archivo especiales 373
Acceso aleatorio a archivos en disco 374
Las funciones fseek() y ftell() 374
La función rewind() 378
Más ejemplos de E/S de archivos en disco 378
Lectura y escritura de datos binarios 378
Las funciones fscanf() y fprintf() 381
Redirección de los flujos estándar mediante freopen() 384
Resumen 387
Preguntas y respuestas 387
Taller 388
Cuestionario 388
Ejercicios 389

Hora 23 Compilación: el preprocesador de C 391
Qué es el preprocesador de C 392
El preprocesador de C en comparación con el compilador 392
Las directivas #define y ##undef 393
Definición de macros tipo función con #define 394
Definición de macros anidadas 396
Compilación de su código bajo condiciones 397
Las directivas #ifdef y #endif 397
La directiva #ifndef 397
Las directivas #if, #elif y #else 399
Compilación condicional anidada 402
Resumen 405
Preguntas y respuestas 405
Taller 406
Cuestionario 406
Ejercicios 407

Hora 24 ¿Qué sigue después? 409
Creación de una lista enlazada 410
Estilo de programación 418
Programación modular 419
Depuración 420
Lo que ha aprendido 420
Palabras reservadas de C 420
Operadores 421
Constantes 422
Tipos de datos 423
Expresiones e instrucciones 426
Instrucciones de control de flujo 426

Materiał chroniony prawem autorskim

Apuntadores 430
Funciones 432
Entrada y salida (E/S) 433
El preprocesador de C 434
El camino a seguir… 434
Resumen 435

Parte VI Apéndices 437

Apéndice A Archivos de encabezado del estándar ANSI 439

Apéndice B Respuestas a los cuestionarios y ejercicios 441

Hora 1, "El primer paso" 441
Cuestionario 441
Hora 2, "Su primer programa de C" 442
Cuestionario 442
Ejercicios 442
Hora 3, "La estructura de un programa de C" 443
Cuestionario 443
Ejercicios 444
Hora 4, "Tipos de datos y palabras reservadas" 445
Cuestionario 445
Ejercicios 445
Hora 5, "Manejo de la entrada y salida estándar" 447
Cuestionario 447
Ejercicios 447
Hora 6, "Manejo de datos" 449
Cuestionario 449
Ejercicios 449
Hora 7, "Ciclos" 451
Cuestionario 451
Ejercicios 451
Hora 8, "Uso de operadores condicionales" 453
Cuestionario 453
Ejercicios 453
Hora 9, "Modificadores de datos y funciones matemáticas" 455
Cuestionario 455
Ejercicios 456
Hora 10, "Control de flujo del programa" 458
Cuestionario 458
Ejercicios 458

Materiał chroniony prawem autorskim

Hora 11, "Apuntadores" 460
Cuestionario 460
Ejercicios 461
Hora 12, "Arreglos" 462
Cuestionario 462
Ejercicios 463
Hora 13, "Cadenas" 465
Cuestionario 465
Ejercicios 466
Hora 14, "Alcance y clases de almacenamiento" 467
Cuestionario 467
Ejercicios 468
Hora 15, "Funciones" 470
Cuestionario 470
Ejercicios 470
Hora 16, "Uso de apuntadores" 473
Cuestionario 473
Ejercicios 474
Hora 17, "Asignación de memoria" 476
Cuestionario 476
Ejercicios 476
Hora 18, "Tipos de datos y funciones especiales" 480
Cuestionario 480
Ejercicios 480
Hora 19, "Estructuras de datos" 482
Cuestionario 482
Ejercicios 482
Hora 20, "Uniones" 486
Cuestionario 486
Ejercicios 486
Hora 21, "Lectura y escritura de archivos" 490
Cuestionario 490
Ejercicios 490
Hora 22, "Funciones de archivo especiales" 494
Cuestionario 494
Ejercicios 494
Hora 23, "Compilación: el preprocesador de C" 499
Cuestionario 499
Ejercicios 500

Índice 503

Materiał chroniony prawem autorskim

Acerca del autor

Tony Zhang tiene más de 15 años de experiencia en la programación de computadoras y en el diseño de sistemas de información para empresas. En la actualidad trabaja para una de las cinco grandes firmas consultoras enfocadas al diseño, desarrollo e implementación de infraestructura relacionada con e-business.

Poseedor de un título de maestría en física, ha publicado diversos artículos de investigación sobre láseres y programación de computadoras. Entre sus principales intereses están la pintura al óleo y la fotografía, que son las dos actividades que más disfruta.

Puede establecer contacto con Tony a través de Sams Publishing, o escribiéndole al correo electrónico `tyc24h@hotmail.com`.

Acerca del autor colaborador

John Southmayd es ingeniero de diseño de software y tiene experiencia en áreas que van desde la programación de sistemas y controladores de dispositivos, hasta el desarrollo de Windows y tecnologías de Internet. Actualmente trabaja como consultor en Excell Data Corporation y vive con su esposa en Kirkland, Washington.

Materiał chroniony prawem autorskim

Dedicatoria

A mi esposa, Ellen, y a mis padres, Zhi-ying y Bing-rong, por su amor e inspiración.

—Tony Zhang

Materiał chroniony prawem autorskim

Reconocimientos

Quisiera agradecer primero a los lectores de la primera edición en inglés de este libro por sus estímulos, paciencia, comentarios y, en especial, por sus críticas, lo cual contribuyó a que esta segunda edición fuera mejorada para aquellas personas que desean emprender un viaje a través del mundo de la programación en C.

Es un gran placer para mí trabajar por segunda ocasión con la editora Sharon Cox. Asimismo, deseo agradecer a los editores Carol Ackerman y Gus Miklos, y al autor colaborador John Southmayd por su excelente trabajo que hizo la segunda edición de este libro más comprensible y en gran medida, si no es que por completo, libre de errores. Además quiero expresar mi aprecio al gran trabajo de los demás miembros del equipo. Todos ellos hicieron posible esta segunda edición.

Aprecio mucho el amor y el apoyo de mi esposa, Ellen, quien me inspira a ver el mundo de la tecnología desde una perspectiva diferente. Siempre es un gran placer comentar con ella temas de filosofía y literatura. Mis padres, a quienes *nunca* podré agradecer lo suficiente, no sólo me dieron amor y cariño, sino también la oportunidad de recibir la mejor educación que pude tener cuando estaba en China.

Materiał chroniony prawem autorskim

Pearson Educación Latinoamérica

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

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

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

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

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

Si lo prefiere, puede mandar un fax a Pearson Educación Latinoamérica al (525) 5387-0811.

También puede ponerse en contacto con Pearson Educación Latinoamérica a través de nuestra página Web: `http://www.pearson.com.mx`.

Materiał chroniony prawem autorskim

Introducción

Si alguien aprende de otros pero no razona, estará desconcertado. Si alguien razona pero no aprende de otros, estará en peligro.

—Confucio

¡Bienvenido a *Aprendiendo C en 24 horas*!

Con base en el éxito de la primera edición en inglés de este libro y la retroalimentación de los lectores, hemos reescrito o modificado cada uno de los capítulos de la primera edición para hacer esta segunda edición más adecuada para principiantes como usted que desean comenzar tan pronto como sea posible con el lenguaje de programación C.

Desde luego, es muy normal dedicar más de 24 horas a entender cabalmente los conceptos y habilidades de programación que se presentan en el libro. Sin embargo, la buena noticia es que el libro ofrece muchos programas de muestra y ejercicios con explicaciones y respuestas claras, lo que facilita la comprensión de los conceptos del lenguaje C.

De hecho, *Aprendiendo C en 24 horas* le ofrece un buen punto de partida en la programación en C, ya que cubre los temas importantes de esta programación y establece una base sólida para un principiante serio como usted. Después de leer el libro podrá escribir por su cuenta programas sencillos de C .

Se beneficiará de la lectura de este libro cuando comience a aplicar programas de C a problemas reales o cuando decida aprender otros lenguajes de programación como Perl, C++ y Java.

¿Quiénes deben leer este libro?

Si ésta es la primera vez que estudia C, este libro está escrito para usted. De hecho, al escribir el libro di por hecho que los lectores no tendrían experiencia previa en programación. Por supuesto, siempre es una gran ventaja si usted tiene algún conocimiento acerca de las computadoras.

Características especiales de este libro

Este libro contiene los siguientes elementos especiales que hacen más claro y sencillo para usted asimilar las características y conceptos de C al momento de presentarlos:

- Cuadros de sintaxis
- Notas
- Precauciones
- Tips

Materiał chroniony prawem autorskim

Los *cuadros de sintaxis* explican algunas de las características más complicadas de C, como las estructuras de control. Cada cuadro consiste en una definición formal de la característica, seguida de una explicación. El siguiente es un ejemplo de un cuadro de sintaxis:

SINTAXIS

La sintaxis de la función `malloc()` es

```
#include <stdlib.h>
void *malloc(size_t tamaño);
```

Aquí, *`tamaño`* especifica el número de bytes de almacenamiento por asignar. El archivo de encabezado, `stdlib.h`, se tiene que incluir antes de que se pueda llamar a la función `malloc()`. Debido a que la función `malloc()` devuelve un apuntador `void`, su tipo se convierte automáticamente al tipo del apuntador que está a la izquierda del operador de asignación.

(Posteriormente aprenderá más acerca de la función `malloc()`).

Las *notas* son explicaciones sobre propiedades interesantes de alguna característica de un programa de C. Veamos los siguientes ejemplos de una nota:

En la salida alineada a la izquierda, el valor desplegado aparece en el extremo izquierdo del campo de valor. En la salida alineada a la derecha, el valor desplegado aparece en el extremo derecho del campo de valor.

Las *precauciones* le previenen sobre errores de programación que debe evitar. Aquí tenemos una precaución típica:

Nunca use en su programa las palabras reservadas de C, ni los nombres de las funciones de biblioteca de C, como nombres de variables.

Los *tips* son sugerencias sobre cómo escribir mejor sus programas en C. A continuación tenemos un ejemplo de tip:

Si tiene un proyecto de programación complejo, divídalo en partes pequeñas, y procure hacer que una función haga una cosa y que la haga muy bien.

Ejemplos de programación

Como mencioné anteriormente, este libro contiene muchos ejemplos útiles de programación con explicaciones. Estos ejemplos pretenden mostrarle cómo usar diferentes tipos de datos y funciones que proporciona C.

Materiał chroniony prawem autorskim

Cada ejemplo tiene un listado del programa; después se presentan los resultados generados a partir de ese listado. El ejemplo ofrece también un análisis del funcionamiento del programa. Se emplean iconos especiales para señalar cada parte del ejemplo: Teclear, Entrada/Salida y Análisis.

En el ejemplo que se muestra en el listado IN.1, hay algunas convenciones tipográficas especiales. La entrada que usted introduce se muestra en negritas en espacio sencillo, y la salida que genera el programa ejecutable del listado IN.1 se muestra en tipo normal en espacio sencillo.

TECLEE **LISTADO IN.1** Lee un carácter introducido por el usuario

```
1:  /* INL01.c: Lee la entrada llamando a getc() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int ch;
7:
8:     printf("Escriba, por favor, un carácter:\n");
9:     ch = getc(stdin);
10:    printf("El carácter que acaba de introducir es: %c\n", ch);
11:    return 0;
12: }
```

Después de crear y ejecutar el archivo ejecutable `INL01.exe`, se despliegan los siguientes resultados. El usuario introduce el carácter `H` y el programa despliega lo que introdujo el usuario.

```
Escriba, por favor, un carácter:
H
El carácter que acaba de introducir es: H
```

ANÁLISIS En la línea 2 del listado IN.1, se incluye el archivo de encabezado `stdio.h` para las dos funciones que se utilizan en el programa: `getc()` y `printf()`. Las líneas 4 a 12 proporcionan el nombre y cuerpo de la función `main()`.

En la línea 6 se declara una variable entera `ch`, la cual se asigna posteriormente al valor de retorno de la función `getc()` de la línea 9. La línea 8 imprime un mensaje que pide al usuario que introduzca un carácter con el teclado. La función `printf()` de la línea 8 utiliza la salida estándar predeterminada `stdout` para desplegar mensajes en la pantalla.

En la línea 9, la entrada estándar `stdin` se pasa a la función `getc()`, lo que indica que el flujo de archivo se recibirá desde el teclado. Después de que el usuario escribe un carácter, la función `getc()` devuelve el valor numérico (es decir, un entero) del carácter. Observe que en la línea 9 se asigna el valor numérico a la variable entera `ch`.

Materiał chroniony prawem autorskim

En la línea 10 se despliega en la pantalla, con ayuda de `printf()`, el carácter introducido. Observe que dentro de la función `printf()` de la línea 10 se usa el especificador de formato de carácter `%c`.

Preguntas, respuestas y taller

Cada hora (es decir, cada capítulo) termina con una sección de preguntas y respuestas que contiene las respuestas a preguntas comunes relacionadas con la lección del capítulo. Después de esta sección hay un taller que consiste en un cuestionario y ejercicios de programación. Las respuesta a estos cuestionarios y las soluciones para los ejercicios se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Para ayudarle a consolidar la comprensión de cada lección, le recomiendo responder las preguntas de los cuestionarios y terminar los ejercicios que se incluyen en el taller.

Convenciones utilizadas en este libro

Por cuestiones gramaticales, en este libro, los mensajes del código y su respectiva salida incluyen caracteres propios del idioma español, como caracteres acentuados, eñes y signos de interrogación y exclamación. Debido a que el lenguaje C fue desarrollado considerando el idioma inglés, aun cuando podrá compilar y ejecutar los programas sin problemas, tal vez estos caracteres no aparezcan como usted espera. Por lo tanto, le recomendamos que no los utilice al seguir los ejemplos del libro. Posteriormente, conforme avance en su aprendizaje del lenguaje C, encontrará que puede colocar los acentos o algún otro carácter empleando el carácter ASCII correspondiente.

Este libro emplea tipos especiales de letra para ayudarle a diferenciar entre el código de C y el lenguaje normal y para identificar conceptos importantes.

- El código de C está tipografiado en una fuente especial `monoespaciada`. Verá que esta fuente se utiliza en listados, ejemplos de entrada/salida y en fragmentos de código. La explicación de las características de C, los comandos, nombres de archivo, instrucciones, variables y todo texto que vea en la pantalla también aparecerán con esta fuente.
- La entrada de un comando y todo lo que se supone que usted introducirá aparecerá en una fuente **`monoespaciada en negritas`**. Esto lo verá principalmente en las secciones de entrada/salida de los ejemplos.
- Los indicadores de posición en las descripciones de sintaxis aparecerán en una fuente *`monoespaciada en cursivas`*. Reemplace el indicador de posición por el nombre de archivo real, parámetro o por cualquier elemento que represente.
- Las *cursivas* resaltan términos técnicos cuando aparecen por primera vez en el texto y en ocasiones se utilizan para destacar aspectos importantes.

Materiał chroniony prawem autorskim

Lo que aprenderá en 24 horas

Aprendiendo C en 24 horas consta de cinco partes. En la parte I, "Los fundamentos de C", aprenderá los aspectos básicos de este lenguaje. A continuación le presentaré un resumen de lo que aprenderá en esta parte:

La hora 1, "El primer paso", le presenta el lenguaje C, el estándar ANSI y los requerimientos básicos de software y hardware para la programación en C.

La hora 2, "Su primer programa de C", muestra todo el procedimiento para escribir, compilar, enlazar y ejecutar un programa de C.

La hora 3, "La estructura de un programa de C", le enseña varios conceptos importantes, como constantes, variables, expresiones e instrucciones. En esta hora también se presenta la anatomía de una función.

La hora 4, "Tipos de datos y palabras reservadas", lista todas las palabras reservadas de C. Se presentan detalladamente cuatro tipos de datos, `char`, `int`, `float` y `double`. Además, se explican las reglas para nombrar una variable.

La hora 5, "Manejo de la entrada y salida estándar", le enseña a recibir la entrada del teclado y a imprimir la salida en la pantalla con la ayuda de un conjunto de funciones de C, como `getc()`, `getchar()`, `putc()`, `putchar()` y `printf()`.

La parte II, "Operadores e instrucciones de control de flujo", hace énfasis en los operadores e instrucciones de control de flujo de C. A continuación le presentaré un resumen de lo que aprenderá en esta parte:

La hora 6, "Manejo de datos", le enseña cómo utilizar los operadores de asignación aritmética, el operador de negación o unario menos, los operadores de incremento/decremento, los operadores relacionales y el operador de conversión explícita.

La hora 7, "Ciclos", presenta los ciclos (es decir, la iteración) con las instrucciones `for`, `while` o `do-while`.

La hora 8, "Uso de operadores condicionales", le habla de otros operadores, como los operadores lógicos, los operadores de bits, el operador `sizeof` y el operador `?:`, los cuales se utilizan con frecuencia en C.

La hora 9, "Modificadores de datos y funciones matemáticas", describe cómo usar modificadores de datos para habilitar o deshabilitar el bit de signo, o para cambiar el tamaño de un tipo de datos. Además, se presentan varias funciones matemáticas que proporciona C.

La hora 10, "Control de flujo del programa", presenta todas las instrucciones de control de flujo que se utilizan en C. Éstas son: `if`, `if-else`, `switch`, `break`, `continue` y `goto`.

Materiał chroniony prawem autorskim

En la parte III, "Apuntadores y arreglos", se exponen los apuntadores y arreglos. A continuación le presentaré un resumen de lo que aprenderá en esta parte:

La hora 11, "Apuntadores", le enseña cómo relacionar variables con apuntadores. También se presentan conceptos como valor izquierdo y valor derecho.

La hora 12, "Arreglos", explica cómo declarar e inicializar arreglos. También se expone la relación que hay en C entre el arreglo y el apuntador.

La hora 13, "Cadenas", se enfoca en la lectura y escritura de cadenas. También se presentan varias funciones de la biblioteca de C para manipular las cadenas, como `strlen()`, `strcpy()`, `gets()`, `puts()` y `scanf()`.

La hora 14, "Alcance y clases de almacenamiento", presenta el alcance de bloque, el alcance de función, el alcance de programa y el alcance de archivo. Además, se explican los especificadores o modificadores de clases de almacenamiento, como `auto`, `static`, `register`, `extern`, `const` y `volatile`.

La parte IV, "Funciones y asignación dinámica de memoria", se enfoca en las funciones y en la asignación de memoria dinámica en C. A continuación le presentaré un resumen de lo que aprenderá en esta parte:

La hora 15, "Funciones", describe la declaración y definición de funciones en C. Se explica la creación de prototipos de funciones, así como la especificación del tipo de retorno de las mismas.

La hora 16, "Uso de apuntadores", le enseña cómo realizar operaciones aritméticas de apuntadores, cómo acceder a los elementos en arreglos utilizando apuntadores y cómo pasar apuntadores a funciones.

La hora 17, "Asignación de memoria", explica el concepto de asignación de memoria de manera dinámica. Se presentan funciones de C utilizadas para la asignación de memoria dinámica, como `malloc()`, `calloc()`, `realloc()` y `free()`.

La hora 18, "Tipos de datos y funciones especiales", presenta el tipo de datos `enum` y el uso de `typedef`. En esta hora también se enseñan la recursión de funciones y los argumentos de línea de comandos para la función `main()`.

La parte V, "Estructuras, uniones, E/S de archivos y más", expone las estructuras, uniones y la E/S de archivos en disco en C. A continuación le presentaré un resumen de lo que aprenderá en esta parte:

La hora 19, "Estructuras de datos", presenta el tipo de datos `structure`. Aprenderá a tener acceso a los miembros de una estructura y a pasar estructuras a funciones con la ayuda de apuntadores. En esta hora también se exponen las estructuras anidadas.

La hora 20, "Uniones", describe el tipo de datos `union`, y la diferencia entre `union` y `structure`. Las aplicaciones de uniones se muestran en varios ejemplos.

Materiał chroniony prawem autorskim

La hora 21, "Lectura y escritura de archivos", explica los conceptos de archivo y flujo en C. En esta primera parte se presentan los aspectos básicos de la entrada y salida de archivos en disco. También se presentan, junto con varios ejemplos, las siguientes funciones de C: `fopen()`, `fclose()`, `fgetc()`, `fputc()`, `fgets()`, `fputs()`, `fread()`, `fwrite()` y `feof()`.

La hora 22, "Funciones de archivo especiales", es la segunda parte de la E/S de archivos en disco, en la que se presentan las funciones `fseek()`, `ftell()` y `rewind()` para mostrar cómo pueden ayudarle a tener un acceso aleatorio a archivos en disco. Además, se enseñan y se invocan en programas de muestra las funciones `fscanf()`, `fprintf()` y `freopen()`.

La hora 23, "Compilación: el preprocesador de C", describe el papel que juega el preprocesador de C. A través de los ejemplos mostrados en esta hora, puede aprender directivas de preprocesador como `#define`, `#undef`, `#ifdef`, `#endif`, `#ifndef`, `#if`, `#elis` y `#else`.

La hora 24, "¿Qué sigue después?", resume las características y conceptos importantes presentados en este libro. Además se explican brevemente el estilo de programación, la programación modular y la depuración. Para posteriores lecturas, se incluye una lista de libros recomendables de C.

Ahora, al pasar el mundo a un nuevo milenio, está usted listo para iniciar el viaje a través del aprendizaje del lenguaje C. Diviértase leyendo este libro y disfrute al programar en C.

Tony Zhang

Downingtown, Pennsylvania

Enero, 2000

Materiał chroniony prawem autorskim

PARTE I

Los fundamentos de C

Hora

1. El primer paso
2. Su primer programa de C
3. La estructura de un programa de C
4. Tipos de datos y palabras reservadas
5. Manejo de la entrada y salida estándar

Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

HORA 1

El primer paso

Un viaje de mil kilómetros comienza dando el primer paso.

—Proverbio chino

Los pensamientos elevados deben tener un lenguaje elevado.

—Aristófanes

Bienvenido a *Aprendiendo C en 24 horas*. En esta primera lección aprenderá lo siguiente:

- Qué es C
- Por qué necesita aprender C
- El estándar ANSI
- Hardware y software requeridos para ejecutar programas de C

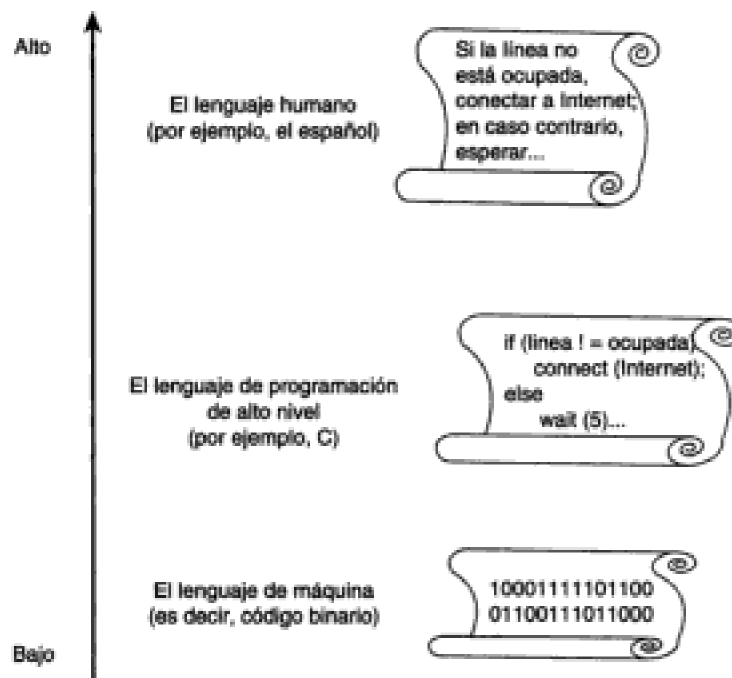
Qué es C

C es un lenguaje de programación que fue desarrollado en 1972 por Dennis Ritchie en AT&T Bell Labs. Ritchie lo denominó C simplemente porque ya existía un lenguaje de programación *B*. (En realidad, el lenguaje B condujo al desarrollo de C.)

C es un lenguaje de programación de alto nivel. De hecho, es uno de los lenguajes de programación de propósito general más populares.

En el mundo de la computación, entre más alejado esté un lenguaje de programación de la arquitectura de la computadora, su nivel será más alto. Los lenguajes de nivel más bajo son los lenguajes de máquina que las computadoras entienden y ejecutan directamente. Por otra parte, los lenguajes de programación de alto nivel se asemejan más al lenguaje humano (vea la figura 1.1).

FIGURA 1.1
El espectro del lenguaje.



Los lenguajes de programación de alto nivel, incluyendo a C, tienen las siguientes ventajas:

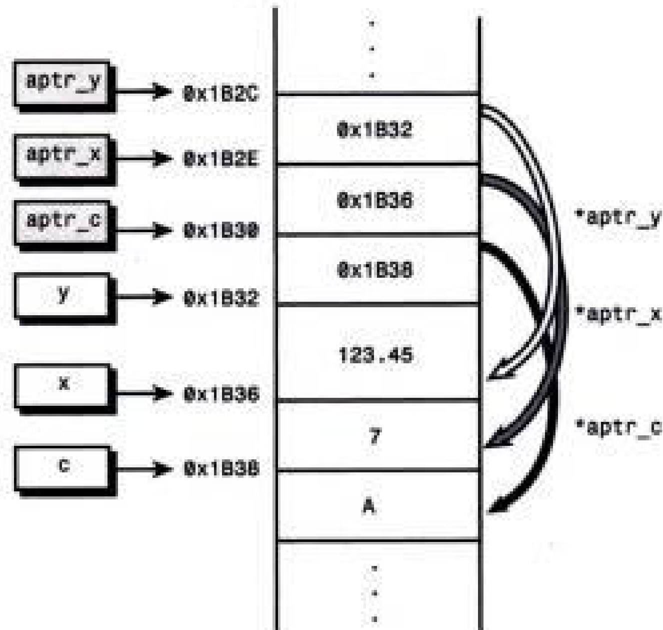
- *Legibilidad*: Los programas son fáciles de leer.
- *Facilidad de mantenimiento*: Es fácil dar mantenimiento a los programas.
- *Portabilidad*: Es fácil portar los programas a través de diferentes plataformas de cómputo.

Materiał chroniony prawem autorskim

La legibilidad y facilidad de mantenimiento del lenguaje C se deben precisamente a su semejanza con el lenguaje humano, en especial con el inglés.

Cada lenguaje de alto nivel necesita de un *compilador* o un *intérprete* para traducir las instrucciones escritas en el lenguaje de programación de alto nivel a un lenguaje de máquina que la computadora pueda entender y ejecutar. Cada máquina podría necesitar un compilador o intérprete distinto para el mismo lenguaje de programación. Por ejemplo, yo utilizo el compilador C de Microsoft para compilar en mi PC los programas de este libro. Si tuviera que ejecutar estos programas en una estación de trabajo basada en UNIX, tendría que compilarlos usando otro tipo de compilador de C. Por lo tanto, la portabilidad de los programas escritos en C se logra recompilando estos programas con diferentes compiladores para distintas máquinas (vea la figura 1.2).

FIGURA 1.2
Portación de programas escritos en C a diferentes tipos de computadoras.



Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

Además, el lenguaje C tiene otras ventajas. Los programas escritos en este lenguaje pueden ser reutilizados. Usted puede guardar partes de sus programas de C en un archivo de biblioteca e invocarlas en su siguiente proyecto de programación simplemente incluyendo dicho archivo. Muchas tareas de programación comunes y útiles ya están implementadas en bibliotecas que vienen incluidas con los compiladores. Además, las bibliotecas le permiten desencadenar con facilidad el poder y la funcionalidad del sistema operativo que esté empleando. En el resto de este libro trataremos más detalles sobre el uso de funciones de biblioteca de C.

C es un lenguaje de programación relativamente pequeño, lo que lo hace más fácil para usted. No tiene que memorizar muchas palabras clave o comandos antes de empezar a escribir programas de C que resuelvan problemas reales.

Para quienes buscan velocidad conservando la conveniencia y la elegancia de un lenguaje de alto nivel, probablemente C sea la mejor elección. De hecho, C le permite tener el control del hardware y de los periféricos de la computadora. Es por ello que en ocasiones a este lenguaje se le llama el lenguaje de programación de alto nivel más bajo.

Varios lenguajes de alto nivel han sido desarrollados con base en C. Por ejemplo, Perl es un conocido lenguaje de programación en el diseño de World Wide Web (WWW) a través de Internet. En realidad, Perl tiene muchas características de C. Si usted entiende C, aprender Perl le será muy fácil. Otro ejemplo es el lenguaje C++, el cual es simplemente una versión ampliada de C, aunque C++ facilita la programación orientada a objetos. Incluso aprender Java es mucho más fácil si usted ya conoce C.

En general, existen dos tipos de lenguajes de programación: lenguajes *compilados* y lenguajes *interpretados*.

Antes de poder ejecutar el programa en su máquina, necesita un compilador para traducir un programa escrito en algún lenguaje compilado a un código que la máquina entienda (es decir, a código binario). Una vez hecha la traducción, puede guardar el código binario en un archivo de aplicación. Puede

mantener operando el archivo de aplicación sin el compilador, a menos que el programa (código fuente) sea actualizado y tenga que recompilarlo. Al código binario o archivo de aplicación también se le conoce como código ejecutable (o archivo ejecutable).

Por otra parte, usted puede ejecutar un programa escrito en un lenguaje interpretado inmediatamente después de terminar de escribirlo, o lo que es igual, mientras lo escribe. Pero, en tiempo de ejecución, dicho programa siempre necesita un intérprete para traducir las instrucciones de alto nivel a instrucciones que entienda la máquina (código binario). No puede ejecutar el programa en una máquina, a menos que tenga el intérprete adecuado.

Puede considerar a C como un lenguaje compilado, debido a que la mayoría de los fabricantes de este lenguaje sólo hacen compiladores para manejar programas escritos en C.

Sin embargo, no hay nada inherente a un lenguaje compilado que impida que alguien proporcione un intérprete para dicho lenguaje; asimismo, hay quienes escriben compiladores para lenguajes interpretados. De hecho, es común mezclar las dos modalidades de lenguajes; un ejemplo de esto es cuando un programador compila código fuente en un pequeño archivo binario, el cual es ejecutado posteriormente por un intérprete en tiempo de ejecución.

El estándar ANSI de C

Durante muchos años, el estándar *de facto* para C fue el libro *El lenguaje de programación C*, escrito por Brian Kernighan y Dennis Ritchie en 1978. Este libro se conoce en la comunidad de programación simplemente como K&R (en referencia a las iniciales de los autores), y a la fecha tiene un lugar en los libreros de muchos programadores. Sin embargo, el libro fue escrito como una introducción a C, no como un estándar general u oficial del lenguaje. Debido a que distintos fabricantes ofrecían diversas implementaciones del lenguaje C, las diferencias entre dichas implementaciones comenzaron a aparecer.

Temiendo que C perdiera su portabilidad, un grupo de fabricantes de compiladores y desarrolladores de software solicitó en 1983 al ANSI (Instituto Estadounidense de Estándares Nacionales) que creara un estándar del lenguaje C. El instituto aprobó la solicitud y formó el Comité Técnico X3J11 para que trabajara en el estándar de C. A finales de 1989, el comité aprobó el estándar ANSI del lenguaje de programación C.

El estándar ANSI de C mejora el estándar original K&R y define un grupo de funciones de uso común de C que se conoce como la biblioteca estándar ANSI de C. En la mayoría de los casos, los compiladores de C incluyen la biblioteca estándar, junto con otras bibliotecas para proporcionar algunas otras funciones específicas del compilador.

Este libro se centra en las funciones de C definidas en el estándar ANSI, el cual manejan todos los fabricantes de compiladores. Todos los programas de este libro pueden ser

compilados por cualquier compilador que se apegue al estándar ANSI. Si está interesado en un compilador específico, puede aprender sus funciones específicas en su respectivo manual de referencia.

Suposiciones acerca del lector

No necesita ninguna experiencia previa en programación para aprender el lenguaje C con este libro, aunque sería mejor si tuviera algún conocimiento acerca de las computadoras. Además, depende de usted determinar qué tan rápido quiere recorrer las 24 horas de este libro: podría sentarse con un gran tarro de café y acabarlo en una sola sesión, o podría tomar una hora al día durante 24 días.

Después de terminar este libro, habiendo realizado todos los ejercicios indicados en él, deberá manejar apropiadamente la sintaxis y las características del lenguaje C. Además, ya tendrá cierta experiencia en muchas de las tareas que se encuentran al programar en C. Cuando esté listo para emprender sus propios proyectos de programación, será capaz de usar C como una herramienta para escribir programas útiles y poderosos. Conforme vaya avanzando, encontrará que siempre hay más que aprender, no sólo acerca de C y de cómo aprovechar su poder, sino también acerca de nuevas tecnologías e ideas de programación en general. Con esfuerzo y mucha práctica podrá apoyarse rápidamente en las habilidades y tecnologías que aprenda.

Configuración de su sistema

Básicamente, todo lo que necesita para compilar y ejecutar sus propios programas en C o los de este libro, es una computadora y un compilador de C. En las siguientes secciones se describen el hardware y software recomendados.

Hardware

Cualquier computadora que tenga o que pueda acceder a un compilador de C está bien. El compilador de C debe ser compatible con el ANSI C. Es muy probable que tenga una PC sobre su escritorio. Una PC 286 con un disco duro de 50 MB y 1 MB de memoria (RAM) es quizás el mínimo requerido para ejecutar un compilador de C basado en DOS. Para un compilador de C basado en Windows, su computadora debe tener un disco duro más grande y más memoria. Para más detalles sobre requerimientos de hardware, consulte al fabricante de su compilador.

Software

Si utiliza una estación de trabajo basada en UNIX, tal vez ya tenga un compilador de C cargado en su máquina, o por lo menos puede acceder a uno en un servidor. Consulte a su administrador de sistemas para saber cómo acceder a un compilador compatible con el

Materiał chroniony prawem autorskim

La figura 1.4 muestra el IDE con el texto que acaba de escribir. No se preocupe por el significado del texto. En el siguiente capítulo, "Su primer programa de C", se lo explicaré.

FIGURA 1.4
Código escrito en el IDE de Visual C++ 5.0.

Obraz chroniony prawem autorskim

A continuación necesita guardar el texto como un archivo. Llamaremos a este archivo `MiPrimerPrograma.c`. También es una buena idea crear un directorio en su disco duro para almacenar sus proyectos de programación, y guardar ahí el archivo. Primero haga clic en el botón Save de la barra de herramientas. Haga clic en el botón New Folder del cuadro de diálogo Save As. Después haga doble clic en esa carpeta para abrirla, escriba MiPrimerPrograma.c en el cuadro File Name, y haga clic en Save. Observe que se utiliza la extensión `.c` para indicar que el archivo que acaba de guardar es un programa de C.

Ahora necesita hacer clic en el menú Build y seleccionar la opción Compile MiPrimerPrograma.c. Al hacerlo, solicite al compilador que compile el texto que acaba de escribir y guardar (vea la figura 1.5). En este punto, Visual C++ podría pedirle que cree un nuevo espacio de trabajo; sólo haga clic en Yes y éste se creará en forma automática. No debe haber errores o advertencias en la ventana de resultados después de ejecutar el compilador.

Después haga clic de nuevo en el menú Build, y esta vez elija la opción Build MiPrimer-Programa.exe, la cual producirá finalmente un archivo ejecutable llamado `MiPrimerPrograma.exe`. La figura 1.6 muestra que no hay errores o advertencias después de generar `MiPrimerPrograma.exe`.

Material chroniony prawem autorskim

FIGURA 1.5
Compilación de un programa de C en el IDE.

```
printf("¡Hola, amigo! Éste es mi primer programa de C.\n");
return 0;
```

```
.obj - 0 error(s), 0 warning(s)
```

FIGURA 1.6
Creación del archivo ejecutable de un programa.

Obraz chroniony prawem autorskim

```
printf("¡Hola, amigo! Éste es mi primer programa de C.\n");
return 0;
```

```
exe - 0 error(s), 0 warning(s)
```

Ahora está usted listo correr el archivo ejecutable, `MiPrimerPrograma.exe`, que acaba de compilar. hacerlo mediante la siguiente ruta: Build, Execute MiPrimer-Programa.exe. Al el archivo ejecutable le aparecerá una ventana de DOS, debido a que dicho archivo aplicación de modo de consola (vea la figura 1.7).

Materiał chroniony prawem autorskim

Figura 1.7
Ejecución de un programa de C.

Ooraz chroniony prawem autorskim

En la figura 1.7 puede ver que la primera línea de la ventana de DOS es la misma que acaba de escribir: "¡Hola, amigo! Éste es mi primer programa de C." En realidad, éste es el resultado de su primer programa (observe que la segunda línea de la ventana de DOS es sólo un aviso del indicador de comandos del DOS).

Muy bien. Acabo de mostrarle cómo usar el compilador de Visual C++ para escribir y compilar un programa de C, y cómo hacerlo ejecutable. Para más detalles, necesita leer libros como *Aprendiendo Visual C++ 5 en 21 días*, el cual se enfoca en enseñarle cómo usar el compilador de Visual C++.

Uso del compilador de Borland

En esta sección voy a mostrarle cómo usar el compilador de C que viene con el paquete C++ de Borland. El procedimiento de esta sección es muy similar al de la anterior. Si necesita aprender más detalles sobre cómo instalar el C++ de Borland, siga las instrucciones que vienen con el compilador.

Daré por hecho que ya instaló una copia de Borland C++ 5.02 en su computadora. Para iniciar el compilador, puede hacer clic en el botón Inicio de su barra de tareas de Windows 95 (o Windows 98, NT o 2000) y seleccionar Programas, Borland C++ 5.02, Borland C++. O bien, puede simplemente ejecutar el archivo de aplicación `bcw.exe` directamente desde el directorio (carpeta) en el que instaló el paquete Borland C++. La figura 1.8 muestra un ejemplo del entorno de desarrollo integrado (IDE) de Borland C++ 5.02.

Después puede abrir un archivo nuevo dentro del IDE, y escribir el siguiente texto en el espacio del archivo recién abierto:

```
#include <stdio.h>

main()
{
     printf ("¡Hola, amigo! Éste es mi primer programa de C.\n");
     return 0;
}
```

Materiał chroniony prawem autorskim

FIGURA 1.8
Creación de un programa en el IDE de Borland C++.

La figura 1.9 nObraz chroniony prawem autorskim se preocupe por el significado del texto.

FIGURA 1.9
Cómo guardar texto de un programa de C en el IDE de Borland.

Materiał chroniony prawem autorskim

Ahora necesita guardar el texto como un archivo. Llamaremos a este archivo `MiPrimer-Programa.c`. Observe que se utiliza la extensión `.c` para indicar que el archivo que acaba de guardar es un programa de C.

Ahora necesita hacer clic en el menú Project y seleccionar la opción Compile. Al hacerlo, solicite al compilador que comience a compilar el texto que acaba de escribir y guardar. La figura 1.10 muestra que no hay errores o advertencias después de compilar `MiPrimer-Programa.c` y crear `MiPrimerPrograma.exe`.

FIGURA 1.10
Cómo compilar un programa de C en el IDE de Borland.

Ahora está usted listo para correr el archivo ejecutable, `MiPrimerPrograma.exe`, que acaba de compilar. Puede correrlo haciendo clic en el botón Run de la barra de herramientas, o puede ejecutarlo directamente desde el directorio en donde lo creó. Al correr el archivo ejecutable le aparecerá una ventana de DOS, debido a que dicho archivo es en realidad una aplicación de DOS (vea la figura 1.11).

FIGURA 1.11
Ejecución de un programa de C en el IDE de Borland.

Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

La figura 1.11 exhibe el resultado exactamente como usted acaba de escribirlo: "¡Hola, amigo! Éste es mi primer programa de C." En realidad, éste es el resultado de su primer programa de C.

Si desea aprender más detalles sobre cómo usar Borland C++, lea un libro como *Aprendiendo Borland C++ en 21 días.*

Resumen

En esta primera lección aprendió los siguientes elementos básicos acerca del lenguaje C:

- C es un lenguaje de programación de propósito general.
- C es un lenguaje de alto nivel que tiene las ventajas de legibilidad, facilidad de mantenimiento y portabilidad.
- C es un lenguaje muy eficiente que permite controlar el hardware y los periféricos de la computadora.
- C es un lenguaje pequeño que usted puede aprender en un tiempo relativamente breve.
- Los programas escritos en C pueden reutilizarse.
- Los programas escritos en C deben compilarse y traducirse a un código legible para la máquina antes de que la computadora pueda ejecutarlos.
- Muchos lenguajes de programación, como Perl, C++ y Java, han adoptado conceptos básicos y características útiles del lenguaje C. Una vez que aprenda C, le será más sencillo aprender estos otros lenguajes.
- Los fabricantes de compiladores de C manejan el estándar ANSI de C para mantener la portabilidad de los programas escritos en este lenguaje.
- Puede utilizar cualquier compilador compatible con el ANSI C para compilar todos los programas de C incluidos en este libro.

En la siguiente lección aprenderá a escribir su primer programa de C.

Materiał chroniony prawem autorskim

Preguntas y respuestas

P ¿Cuál es lenguaje de nivel más bajo en el mundo de la computación?

R El lenguaje de máquina de la computadora, conformado por ceros y unos, es el de más bajo nivel, debido a que es el único lenguaje que la computadora puede entender en forma directa.

P ¿Cuáles son las ventajas de los lenguajes de programación de alto nivel?

R La legibilidad, la facilidad de mantenimiento y la portabilidad son las principales ventajas de los lenguajes de alto nivel.

P ¿Qué es C, en última instancia?

R C es un lenguaje de programación de propósito general, y es un lenguaje de alto nivel que tiene ventajas como la legibilidad, facilidad de mantenimiento y portabilidad. Además, C permite descender al nivel del hardware para incrementar, si es necesario, la velocidad de rendimiento. Se necesita un compilador de C para traducir los programas escritos en C a un código que la máquina entienda. La portabilidad de estos programas se logra recompilándolos con compiladores de C específicos para cada tipo de computadora.

P ¿Puedo aprender C en poco tiempo?

R Sí. C es un lenguaje de programación pequeño. No hay muchas palabras clave o comandos que recordar. Además, es muy fácil leer y escribir los programas de C, ya que éste es un lenguaje de programación de alto nivel muy parecido al lenguaje humano, en especial al inglés. Por lo tanto, usted puede aprender C en un tiempo relativamente corto.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo que responda el cuestionario de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Cuáles son los lenguajes de nivel más bajo y más alto mencionados en este libro?
2. ¿Puede una computadora entender directamente un programa escrito en C? ¿Qué se necesita para traducir un programa escrito en C a un código que la máquina entienda (es decir, código binario)?
3. De ser necesario, ¿es posible reutilizar un programa de C en otro programa de C?
4. ¿Por qué necesitamos el estándar ANSI para el lenguaje C?

HORA 2

Su primer programa de C

Corta tu propia leña y te calentará doblemente.

—Proverbio chino

En la hora 1, "El primer paso", aprendió que C es un lenguaje de programación de alto nivel y que necesita un compilador de C para traducir sus programas de C a código binario que la computadora pueda entender y ejecutar. En esta lección escribirá su primer programa de C y aprenderá los fundamentos de dicho programa, como son:

- La directiva `#include`
- Archivos de encabezado
- Comentarios
- La función `main()`
- La instrucción `return`
- La función `exit()`

- El carácter de nueva línea (`\n`)
- Generar un archivo ejecutable a partir de un programa de C
- Depuración

Un programa sencillo de C

Veamos su primer programa de C, el cual se muestra en el listado 2.1. Más adelante en esta lección escribirá por primera vez su propio programa de C.

LISTADO 2.1 Un programa sencillo de C

```
1:  /* 02L01.c: Éste es mi primer programa de C */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     printf ("¡Hola, amigo! Éste es mi primer programa de C.\n");
7:     return 0;
8:  }
```

Éste es un programa de C muy sencillo que está guardado en un archivo de nombre `02L01.c`. Observe que el nombre de un programa de C debe tener la extensión `.c`. Si instaló un compilador de C y configuró el entorno de desarrollo correspondiente, podrá compilar este programa y convertirlo en un archivo ejecutable. Más adelante en este capítulo le diré cómo hacer un archivo ejecutable.

En la hora anterior aprendió cómo escribir un programa en su editor de texto y cómo guardarlo como un archivo de programa de C. Quizá haya observado que a diferencia del ejemplo del capítulo anterior, en este ejemplo cada línea está numerada. Sólo hago esto para tener una referencia al momento de explicar lo que hace cada línea del programa. A diferencia de otros lenguajes como BASIC, el lenguaje C no emplea números de línea. De hecho, si usted escribe los números de línea en el listado, su programa no funcionará. Así que cuando escriba estos programas recuerde no poner los números de línea que se muestran en el libro.

Dos cosas que puede notar al mirar el listado 2.1 son los caracteres de punto y coma y la sangría en las líneas 6 y 7. A diferencia de otros lenguajes, como BASIC, el final de una línea no tiene un significado especial en C. Es completamente válido (y recomendable, en muchos casos) dividir una instrucción en varias líneas para dar claridad. En general, una instrucción individual de C termina con un punto y coma; pero posteriormente veremos mucho más al respecto. La sangría sirve para identificar los distintos niveles de un programa en una especie de formato esquemático. La función `main()` es el nivel

Materiał chroniony prawem autorskim

principal del programa, de modo que va a la extrema izquierda. Las líneas 6 y 7 son parte de `main()`, así que tienen una sangría de un nivel hacia la derecha. Por lo regular se emplea la tecla Tab para sangrar un nivel antes de comenzar a escribir. Debemos señalar que al igual que con los números de línea, el compilador no realiza el sangrado; ni siquiera lo nota. El programador es libre de usar cortes de línea y sangrado, conocidos como *espacios en blanco*, para hacer que el programa luzca legible. Éste es un asunto de estilo, pero es buena idea seguir las convenciones aceptadas generalmente a fin de que otros programadores puedan entender sus programas y viceversa. Observe el uso del espacio en blanco al avanzar en este libro y siéntase en libertad de desarrollar su propio estilo.

2

Yo configuré mi entorno de desarrollo de tal manera que todos los programas de este libro se pudieran compilar y convertir en aplicaciones de consola. Por ejemplo, `02L01.exe` es el nombre de la aplicación de consola generada a partir de `02L01.c`. Observe que se incluye `.exe` como la extensión del nombre de un programa de aplicación de DOS o Windows (es decir, un archivo ejecutable).

Además, guardo todos los archivos ejecutables generados a partir de los programas de este libro en un directorio de mi computadora llamado `C:\app`. Por lo tanto, si escribo `02L01` desde el indicador de comandos de DOS y oprimo la tecla Entrar, puedo ejecutar el archivo ejecutable `02L01.exe` y exhibir en la pantalla el mensaje `¡Hola, amigo! Éste es mi primer programa de C`. El siguiente resultado es una copia de la pantalla:

SALIDA
```
¡Hola, amigo! Éste es mi primer programa de C.
```

Comentarios

Veamos ahora más de cerca el programa de C del listado 2.1.

La primera línea contiene un comentario:

```
/* 02L01.C: Éste es mi primer programa de C */
```

Puede observar que esta línea comienza con una combinación de diagonal y asterisco, `/*`, y termina con `*/`. En C, a la combinación `/*` se le denomina *marca de apertura de comentario*, y a la combinación `*/`, *marca de cierre de comentario*. El compilador de C ignora todo lo que se encuentra entre las marcas de apertura y de cierre de comentario. Esto significa que el compilador ignora por completo el comentario de la primera línea del listado 2.1, `02L01.C: Éste es mi primer programa de C`.

La única finalidad de incluir comentarios en su programa de C es ayudarle a documentar lo que hacen el programa o algunas de sus secciones específicas. Recuerde, usted y otros

programadores escriben los comentarios. Por ejemplo, cuando usted lee el código de otra persona, los comentarios le ayudan a entender lo que hace dicho código, o por lo menos lo que pretende hacer. Al hacerse más extensos y complicados sus programas, puede emplear comentarios para escribir notas para usted mismo acerca de lo que intenta hacer y por qué intenta hacerlo.

Si agrega muchos comentarios a su programa, no tiene que preocuparse por su tamaño o velocidad de rendimiento. Agregar comentarios a un programa de C no aumenta el tamaño del código binario (es decir, del archivo ejecutable), aunque sí podría agrandar el tamaño del programa mismo (es decir, del código fuente). Los comentarios del programa de C no afectan de ninguna manera la velocidad de rendimiento del archivo ejecutable generado a partir de dicho programa.

El lenguaje C le permite escribir un comentario que abarque más de una línea. Por ejemplo, en C usted puede escribir un comentario como éste:

```
/*
   Este comentario no incrementa el tamaño del
   archivo ejecutable (código binario), ni
   afecta la velocidad de rendimiento.
*/
```

que es igual a éste:

```
/* Este comentario no incrementa el tamaño del */
/* archivo ejecutable (código binario), ni */
/* afecta la velocidad de rendimiento.*/
```

Actualmente hay otra forma de poner comentarios en un programa de C. C++ comenzó utilizando dos diagonales (//) para señalar el inicio de una línea de comentario; ahora muchos compiladores de C también utilizan esta convención. El comentario termina al final de la línea. Por ejemplo, si escribo un programa de C en Borland C++ o en Visual C++, los siguientes dos comentarios serán idénticos:

```
/*
   Este comentario no incrementa el tamaño del
   archivo ejecutable (código binario), ni
   afecta la velocidad de rendimiento.
*/
// Este comentario no incrementa el tamaño del
// archivo ejecutable (código binario), ni
// afecta la velocidad de rendimiento.
```

Observe que este nuevo estilo de utilizar // como la marca de inicio de un comentario no ha sido aprobado por ANSI. Asegúrese de que su compilador de C soporte esta marca antes de utilizarla.

Materiał chroniony prawem autorskim

Algo que se debe señalar es que el estándar ANSI no soporta *comentarios anidados*, es decir, comentarios dentro de comentarios. Por ejemplo, el estándar ANSI no permite lo siguiente:

```
/* Ésta es la primera parte del primer comentario
   /* Éste es el segundo comentario */
   Ésta es la segunda parte del primer comentario */
```

Puede usar la marca de apertura de comentario /* y la marca de cierre de comentario */ para probar y corregir cualquier error que encuentre en su programa de C. A esto se le conoce comúnmente como *comentar* (o marcar como comentario) un bloque de código. Puede comentar una o más instrucciones de su programa cuando necesite concentrarse en otras instrucciones y observar minuciosamente su comportamiento. El compilador de C ignorará las instrucciones que usted marque como comentarios.

Después puede restaurar las instrucciones que marcó como comentarios simplemente quitando las marcas de apertura y cierre de comentario. De esta manera, no necesita borrar o reescribir ninguna instrucción durante las pruebas y la depuración.

Sin embargo, ya que no puede anidar comentarios hechos con /* y */, si intenta comentar código que ya contenga estos comentarios, su programa no se compilará. Una razón por la que son muy útiles los comentarios de tipo // es que se pueden anidar y, por lo tanto, se pueden comentar legalmente con /* y */.

Si su editor de texto soporta el resaltado en color de la sintaxis, puede usar esto para saber de un vistazo si las secciones de código están en realidad comentadas como usted lo quiere.

La directiva `#include`

Pasemos ahora a la línea 2 del programa de C del listado 2.1:

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

Usted puede ver que esta línea comienza con un signo de numeral, `#`, seguido por `include`. En C, `#include` forma una *directiva de preprocesador* que indica al preprocesador de C que busque un archivo y coloque el contenido de ese archivo en donde la directiva `#include` indique.

El preprocesador es un programa que hace algunos preparativos para el compilador de C antes de compilar su código. En la hora 23, "Compilación: el preprocesador de C", se exponen más detalles acerca del preprocesador.

También puede ver en esta línea que `<stdio.h>` está después de la directiva `#include`. Podría pensar que el archivo solicitado por la directiva `#include` es algo llamado `stdio.h`. Tiene razón. Aquí, la directiva `#include` solicita al preprocesador de C que busque y coloque el archivo `stdio.h` en el lugar del programa donde se encuentra la directiva.

El archivo stdio.h significa *encabezado de entrada-salida estándar*. Este archivo contiene numerosos prototipos y macros para realizar operaciones de entrada o salida (E/S) de programas de C. Verá más de la E/S de programas en la hora 5, "Manejo de la entrada y salida estándar".

Algunos sistemas operativos distinguen entre letras mayúsculas y minúsculas, pero otros no lo hacen. Por ejemplo, stdio.h y STDIO.H son nombres idénticos en una PC, pero son diferentes en UNIX.

Archivos de encabezado

Los archivos que se incluyen mediante la directiva #include, como stdio.h, se denominan *archivos de encabezado* debido a que las directivas #include se colocan casi siempre al inicio, o a la cabeza, de los programas de C. De hecho, la extensión .h significa "header" (encabezado), y se suele hacer referencia a ellos como archivos punto h.

Además de stdio.h, existen más archivos de encabezado, como stdlib.h, string.h, math.h, etcétera. En el apéndice A, "Archivos de encabezado del estándar ANSI", se proporciona una lista de todos ellos. Los archivos de encabezado específicos que necesite incluir dependerán de las funciones específicas de biblioteca que pretenda llamar. La documentación de las funciones de biblioteca le dirá qué archivo de encabezado se requiere.

Paréntesis angulares (< >) y comillas dobles (" ")

En la segunda línea del listado 2.1 hay dos paréntesis angulares, < y >, que se usan para encerrar a stdio.h. Tal vez se pregunte para qué sirven los paréntesis angulares. En C, los paréntesis angulares solicitan al preprocesador de C que busque un archivo de encabezado en un directorio distinto al actual.

Por ejemplo, en mi computadora, el directorio actual que contiene el archivo 02L01.C se llama C:\code. Por lo tanto, los paréntesis angulares que rodean a <stdio.h> le indican al preprocesador de C que busque el archivo stdio.h en un directorio diferente a C:\code.

Si desea que el preprocesador busque primero en el directorio actual antes de buscar en otra parte, puede utilizar comillas dobles para encerrar el nombre del archivo de encabezado. Por ejemplo, cuando el preprocesador de C ve "stdio.h", busca primero el archivo de encabezado stdio.h en el directorio actual, que en mi máquina es C:\code, antes de buscar en otra parte.

Los archivos de encabezado se guardan normalmente en un subdirectorio denominado include. Por ejemplo, yo instalé un compilador de C de Microsoft en el directorio MSVC de mi disco duro, el cual está etiquetado como la unidad C. Entonces, la ruta de acceso al archivo de encabezado se convierte en C:\MSVC\include.

Materiał chroniony prawem autorskim

Por lo regular, el compilador determina, al momento de instalarlo, la ruta de acceso en donde se guardarán los archivos de encabezado. A esto se le conoce comúnmente como el directorio include o la ruta de acceso include de su entorno. No tendrá que preocuparse por el directorio include, sino hasta que cree sus propios archivos de encabezado. Por ahora, sólo necesita especificar el nombre del archivo de encabezado que desea incluir.

La función `main()`

En la línea 4 del listado 2.1 se encuentra la siguiente función:

```
main ()
```

Ésta es una función muy especial en C. Todo programa de C debe tener una (y sólo una) función `main()`. En la hora 3, "La estructura de un programa de C", se proporcionan explicaciones más genéricas acerca de las funciones.

Puede colocar la función `main()` en cualquier lugar de su programa de C. Sin embargo, la ejecución de su programa siempre comienza con la función `main()`. Si crea otras funciones en su programa, `main()` siempre se ejecutará primero, incluso si está al final de su archivo de programa.

En el listado 2.1, el cuerpo de la función `main()` inicia en la línea 4 y termina en la línea 8. Debido a que éste es un programa muy sencillo, `main()` es la única función definida en el programa. Dentro del cuerpo de la función `main()` se llama a una función de biblioteca de C, `printf()`, a fin de imprimir un mensaje de saludo (vea la línea 6). En la hora 5 se abordan más detalles acerca de `printf()`.

Otra cosa importante respecto a `main()` es que la ejecución de todo programa de C termina con `main()`. Un programa concluye cuando se han ejecutado todas las instrucciones de la función `main()`.

El carácter de nueva línea (`\n`)

Algo que vale la pena mencionar acerca de la función `printf()` es el carácter de *nueva línea*, `\n`. Por lo regular agregado como sufijo al final de un mensaje, el carácter de nueva línea indica a la computadora que mueva el cursor al inicio de la siguiente línea, para que cualquier cosa que se imprima después del mensaje comience en la siguiente línea de la pantalla.

En un entorno UNIX, `\n` pasa a la siguiente línea por sí mismo, pero deja el cursor en la posición en que estaba en la línea anterior. En este caso, es necesario imprimir `\r\n` en vez de solamente `\n`. El carácter `\r` es el carácter de *retorno de carro*. Al ejecutar los programas de muestra de este libro, podrá saber de inmediato si el cursor regresa al principio de la nueva línea; si no es así, simplemente utilice `\r\n` siempre que vea `\n` en los listados de programas.

El ejercicio 3 de esta lección le da la oportunidad de utilizar el carácter de nueva línea para dividir en dos líneas un mensaje de una línea.

La instrucción `return`

En C, todas las funciones devuelven valores. Por ejemplo, al crear una función para sumar dos números, usted puede hacer que dicha función le devuelva el valor de la suma.

La función `main()` por sí misma devuelve un valor entero. En C, los enteros son números decimales sin fracciones.

Por lo tanto, en la línea 7 del listado 2.1 hay una instrucción, `return 0;`, que indica que la función `main()` devuelve un `0` y el programa termina normalmente. Hay casos en los que debe terminar sus programas debido a una condición de error. Cuando eso sucede, puede devolver valores distintos a `0` para indicar al sistema operativo (o al programa que ejecutó su programa) que hubo un error.

La función `exit()`

También existe una función de biblioteca de C, `exit()`, que se puede utilizar para terminar un programa. Debido a que la función `exit()` está definida en el archivo de encabezado `stdlib.h`, usted debe incluir el archivo de encabezado al principio de su programa para poder usar la función. La función `exit()`no devuelve un valor a su programa por sí misma.

Observe que `return` y `exit()` también se pueden usar en otras funciones. En lo que resta del libro verá más ejemplos de la palabra reservada (o palabra clave, como también se le conoce) `return`.

Compilación y enlace

Ya debe estar ansioso por saber cómo se hace un archivo ejecutable. Veamos cómo se compila un programa de C y cómo se traduce a un archivo ejecutable. Como se muestra en la figura 2.1, se necesitan por lo menos tres pasos para crear un archivo ejecutable.

Primero se hace un archivo de programa escrito en C, denominado *código fuente*. El nombre del archivo de código fuente termina con la extensión `.c`.

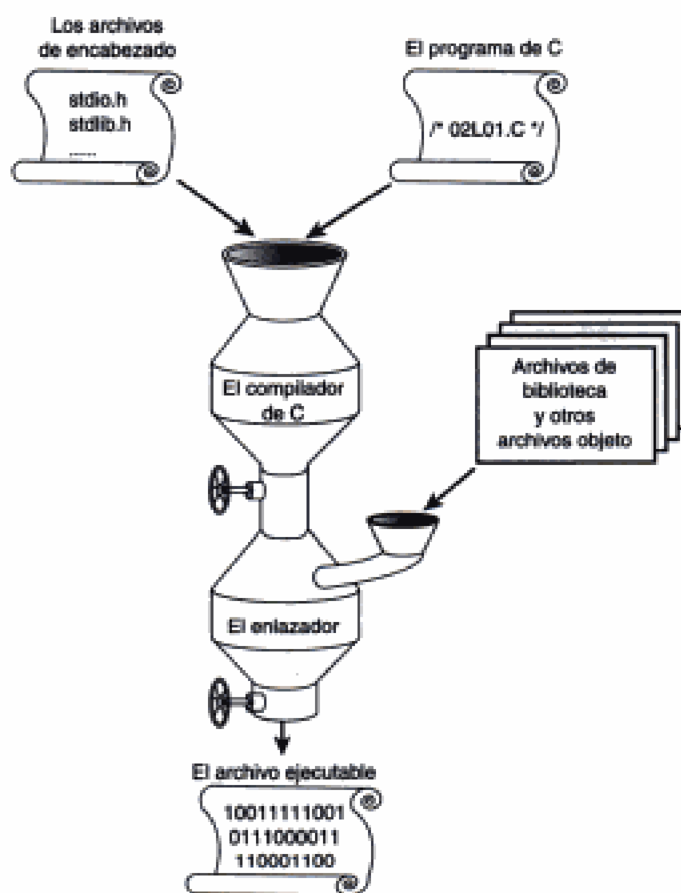
Después, un *compilador* de C compila el archivo de código fuente y crea un nuevo archivo. El nuevo archivo es un *archivo objeto*. En el sistema operativo UNIX, el nombre del archivo objeto termina con la extensión `.o`. En los sistemas operativos DOS y Windows, la extensión es `.obj`.

No es posible ejecutar el archivo objeto debido a que falta cierto código de funciones. Es necesario concluir el paso siguiente: el enlace. Éste se hace invocando a un programa especial llamado *enlazador*, el cual viene normalmente con el paquete del compilador.

Materiał chroniony prawem autorskim

Un enlazador se emplea para vincular el archivo objeto, la biblioteca estándar de C y otras bibliotecas generadas por el usuario para producir un archivo ejecutable: el código binario. En esta etapa, el código binario de las funciones de biblioteca que se llaman en el código fuente se combina con el archivo objeto; el resultado se guarda en un nuevo archivo: un archivo ejecutable. Como aprendió en el primer capítulo de este libro, el nombre de un archivo ejecutable de DOS o de Windows termina por lo regular con la extensión `.exe` (`.com` es otra extensión que se usa para un nombre de archivo ejecutable de DOS). En UNIX no es necesario incluir dicha extensión en el nombre de un archivo ejecutable.

FIGURA 2.1
Creación de un archivo ejecutable mediante el compilador y el enlazador.



Más adelante aprenderá que en muchos casos tal vez tenga que enlazar varios archivos objeto a fin de crear un programa ejecutable.

Observe que tanto el archivo objeto como el archivo ejecutable son dependientes de la máquina. Usted no puede simplemente pasar un archivo ejecutable de la plataforma de cómputo actual a otra que sea operada por un sistema operativo distinto, aunque el código fuente de ese archivo ejecutable, presumiblemente escrito en ANSI C, podría ser independiente de la máquina (es decir, portable).

La portabilidad es un concepto importante en C, ya que fue una de las metas del diseño original del lenguaje. La portabilidad del código C es la que impulsó su uso y su popularidad. En aquellos tiempos en que las aplicaciones y los sistemas operativos eran hechos a la medida de un sistema de cómputo específico, el software tenía que ser escrito desde cero cada vez que aparecía una nueva computadora. El advenimiento del lenguaje C disoció el software del hardware y, en un sentido muy concreto, ayudó al surgimiento de la industria del software como la conocemos en la actualidad. En esencia, la portabilidad se refiere al proceso de portar un programa a una computadora y/o sistema operativo diferente. El desarrollo de software es un proceso caro y tardado, y reescribir un programa existente para que funcione en una nueva computadora es una tarea desalentadora que es mejor evitar. Suponiendo que cada combinación de sistema operativo y compilador esté adaptada a la computadora específica en la que se pretende ejecutar, entonces un programa de C debe ser portable con facilidad haciéndole cambios mínimos al código. La portabilidad se menciona a lo largo de este libro, ya que es importante que los programas de C no tengan que preocuparse por el sistema de cómputo específico en el que operen. Por fortuna, el estándar ANSI de C proporciona muchas características y funciones que le permiten a su programa adaptarse básicamente a su entorno, en lugar de actuar bajo suposiciones hechas por usted, el programador. Si desde el principio se hace el hábito de mantener la portabilidad, entonces, al avanzar en C, este hábito se volverá parte de usted.

¿Qué es lo que está mal en mi programa?

Al terminar de escribir un programa de C y comenzar a compilarlo, quizás obtenga mensajes de error o advertencias. Que no le dé pánico cuando vea mensajes de error. Somos humanos. Todos cometemos errores. De hecho, debe apreciar que su compilador capte algunos errores por usted antes de que continúe.

Por lo regular, su compilador puede ayudarle a revisar la gramática, es decir, la *sintaxis* de su programa de C y a asegurarse de que siguió adecuadamente las reglas de la programación de C. Por ejemplo, si olvida poner el paréntesis de cierre en la función `main()` de la línea 8 del listado 2.1, obtendrá un mensaje de error similar a éste: `syntax error : end of file found`.

Además, el enlazador emitirá un mensaje de error si no puede encontrar en las bibliotecas el código faltante de una función necesaria. Por ejemplo, si escribió `printf()` como `pprintf()` en el programa del listado 2.1, verá el mensaje de error: `'_pprintf' : unresolved external` (o algo parecido).

Todos los errores que encuentren el compilador y el enlazador se deben corregir antes de generar un archivo ejecutable (código binario).

Material chroniony prawem autorskim

Depuración de su programa

En el mundo de la computación, a los errores de programa se les llama también *bugs*. En muchos casos, su compilador y enlazador de C no encuentran ningún error en su programa, pero el resultado de ejecutarlo no es el que usted esperaba. Quizás necesite usar un depurador para detectar esos errores "ocultos" en su programa.

Normalmente, su compilador de C ya incluye un programa de depuración. El depurador puede ejecutar su programa línea por línea de manera que usted pueda observar de cerca qué es lo que está sucediendo con el código de cada línea, o de modo que pueda pedir al depurador que detenga la ejecución de su programa en cualquier línea. Para más detalles acerca de su depurador, consulte las instrucciones que vienen con su compilador de C.

Más adelante aprenderá que depurar es un paso muy importante y necesario al escribir programas. (Este tema se aborda en la hora 24, "¿Qué sigue después?".)

Resumen

En esta lección aprendió los siguientes conceptos e instrucciones acerca del lenguaje C:

- Es necesario incluir algunos archivos de encabezado al principio de su programa de C.
- Los archivos de encabezado, como `stdio.h` y `stdlib.h`, contienen las declaraciones de funciones utilizadas en su programa de C; por ejemplo, las funciones `printf()` y `exit()`.
- Los comentarios de su programa de C son necesarios para ayudarle a documentar sus programas. Puede poner los comentarios en cualquier parte de sus programas.
- En el ANSI C, un comentario comienza con la marca de apertura de comentario, `/*`, y termina con la marca de cierre de comentario, `*/`.
- Todo programa de C debe tener una (y sólo una) función `main()`. La ejecución del programa inicia y termina con la función `main()`.
- La instrucción `return` se puede utilizar para devolver un valor con el fin de indicar al sistema operativo si ocurrió un error. La función `exit()` termina un programa; el argumento para la función también indica el estado de error.
- Compilar y enlazar son pasos consecutivos que se deben realizar antes de producir un archivo ejecutable.
- Todos, incluyéndonos usted y yo, cometemos errores al programar. Depurar es un paso muy importante en el diseño y codificación de su programa.

En la siguiente lección aprenderá más acerca de lo fundamental de los programas de C.

Preguntas y respuestas

P ¿Por qué necesita poner comentarios en sus programas?

R Los comentarios le ayudan a documentar lo que hace el programa. Cuando un programa se vuelve muy complejo, necesita escribir comentarios para explicar las diferentes partes de dicho programa.

P ¿Por qué es necesaria la función `main()` en un programa?

R La ejecución de un programa de C comienza y termina con la función `main()`. Sin ella, la computadora no sabe dónde comenzar a ejecutar un programa.

P ¿Qué hace la directiva `#include`?

R La directiva `#include` se utiliza para incluir archivos de encabezado que contienen las declaraciones para las funciones que utiliza su programa de C. En otras palabras, la directiva `#include` indica al preprocesador de C que busque en la ruta de acceso include el archivo de encabezado especificado.

P ¿Por qué necesita un enlazador?

R Después de compilar, puede faltar aún en el archivo objeto de un programa algún código de funciones. Entonces se debe usar un enlazador para vincular el archivo objeto a la biblioteca estándar de C o a otras bibliotecas generadas por el usuario, e incluir el código de función faltante de manera que se pueda crear un archivo ejecutable.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Puede un compilador de C ver los comentarios de un programa?
2. ¿Qué clase de archivos produce en realidad un compilador de C?
3. ¿Devuelve algún valor la función `exit()`?, ¿y la función `return`?
4. ¿Qué es un archivo de encabezado?

Materiał chroniony prawem autorskim

Ejercicios

1. ¿Es lo mismo `#include <stdio.h>` que `#include "stdio.h"`?
2. Es el momento de que escriba un programa por su cuenta. Tomando como referencia el programa del listado 2.1, escriba un programa de C que imprima el mensaje: `Es divertido escribir mi propio programa de C.`
3. Actualice el programa del listado 2.1 agregando un carácter más de nueva línea dentro del mensaje que imprime la función `printf()`. Después de ejecutar el archivo ejecutable actualizado, debe ver en la pantalla dos líneas del mensaje:

```
¡Hola, amigo!
Éste es mi primer programa de C.
```

4. ¿Qué mensaje de error o advertencia, si lo hay, obtendrá al compilar el siguiente programa?

```
#include <stdlib.h>
#include <stdio.h>
main()
{
   printf ("¡Hola, amigo! Éste es mi primer programa de C.\n");
   exit(0);
}
```

5. ¿Qué mensaje de error obtendrá del siguiente programa cuando intente compilarlo?

```
void main()
{
   printf ("¡Hola, amigo! Éste es mi primer programa de C.\n");
   return 0;
}
```

2

Materiał chroniony prawem autorskim

HORA 3

La estructura de un programa de C

El entero es igual a la suma de sus partes.

—Euclides

En la hora 2, "Su primer programa de C", vio y escribió algunos programas sencillos en C. También aprendió acerca de la estructura básica de un programa de C. Usted sabe que un programa escrito en C se tiene que compilar antes de que pueda ser ejecutado. En esta lección aprenderá más de lo fundamental de un programa de C, como por ejemplo:

- Constantes y variables
- Expresiones
- Instrucciones
- Bloques de instrucciones
- Tipos y nombres de funciones de C
- Argumentos para funciones
- El cuerpo de una función
- Llamadas a funciones

Los elementos básicos de un programa de C

Así como un edificio está hecho de ladrillos, un programa de C está hecho de elementos básicos, como expresiones, instrucciones, bloques de instrucciones y bloques de funciones. Estos elementos se explican en las siguientes secciones. Pero antes necesita aprender dos elementos más pequeños pero importantes, las constantes y las variables, las cuales conforman las expresiones.

Constantes y variables

Como su nombre indica, una *constante* es un valor que nunca cambia. Por otra parte, una *variable* se puede usar para presentar diferentes valores.

Puede comparar a una constante con un disco compacto de música; la música almacenada en el disco compacto nunca cambia. Una variable es más parecida a un casete de audio: usted puede actualizar el contenido del casete simplemente sustituyendo las canciones que tenía por otras nuevas.

Puede ver muchos ejemplos en los que hay constantes y variables en la misma instrucción. Considere, por ejemplo, la siguiente:

```
i = 1;
```

en donde el símbolo `1` es una constante ya que siempre tiene el mismo valor (`1`), y al símbolo `i` se le asigna la constante `1`. En otras palabras, `i` contiene el valor `1` después de ejecutar la instrucción. Luego, si hay otra instrucción,

```
i = 10;
```

después de que se ejecuta esta instrucción, a `i` se le asigna el valor `10`. Debido a que `i` puede contener diferentes valores, en el lenguaje C se le llama variable.

Expresiones

Una expresión es una combinación de constantes, variables y operadores que se emplean para denotar cálculos.

Por ejemplo, la siguiente:

```
(2 + 3) * 10
```

es una expresión que primero suma 2 y 3, y después multiplica el resultado de la suma por 10. (El resultado final de la expresión es `50`.)

En forma similar, la expresión `10 * (4 + 5)` produce `90`. La expresión `80/4` da como resultado `20`.

Aquí tenemos otros ejemplos de expresiones:

Materiał chroniony prawem autorskim

Expresión	*Descripción*
`6`	Una expresión de una constante.
`i`	Una expresión de una variable.
`6 + i`	Una expresión de una constante más una variable.
`exit(0)`	Una expresión de llamada a una función.

Operadores

Como ha visto, una expresión puede contener símbolos como `+`, `*` y `/`. En el lenguaje C, estos símbolos se denominan *operadores aritméticos*. La tabla 3.1 presenta una lista de todos los operadores aritméticos y su significado.

TABLA 3.1 Operadores aritméticos de C

Símbolo	*Significado*
`+`	Suma
`-`	Resta
`*`	Multiplicación
`/`	División
`%`	Residuo (o módulo)

Tal vez ya esté familiarizado con los operadores aritméticos, con excepción del de residuo (`%`). Este operador se usa para obtener el residuo de la división del primer operando entre el segundo. Por ejemplo, la expresión

```
6 % 4
```

produce un valor de `2` debido a que 4 cabe una vez en 6 con un residuo de 2.

El operador de residuo, `%`, también se conoce como *operador de módulo*.

Entre los operadores aritméticos, los operadores de multiplicación, división y residuo tienen una precedencia más alta que los operadores de suma y resta. Por ejemplo, la expresión

```
2 + 3 * 10
```

da como resultado `32`, no `50`, debido a que el operador de multiplicación tiene mayor precedencia que el de suma. Primero se calcula `3 * 10` y después se suma 2 al resultado de la multiplicación.

Como quizá sepa, puede colocar paréntesis alrededor de una suma (o de una resta) para forzar que la suma (o resta) se realice antes de un cálculo de multiplicación, división o módulo. Por ejemplo, la expresión

```
(2 + 3) * 10
```

realiza primero la suma de 2 y 3 antes de multiplicar el resultado por 10.

La coma y el punto y coma son operadores empleados en la sintaxis. En general, el punto y coma se utiliza para indicar el final de una instrucción, como verá más adelante. La coma se usa en ciertas instancias en las que una instrucción consta de una lista de expresiones o declaraciones.

Aprenderá más operadores del lenguaje C en la hora 6, "Manejo de datos", y en la hora 8, Uso de operadores condicionales".

Identificadores

Junto con los números (como la constante `7`) y los operadores (como el símbolo `+`), las expresiones también pueden contener palabras denominadas *identificadores*. Los nombres de funciones (como `exit`) y los nombres de variables (como `i`), así como las palabras reservadas son *identificadores* de C.

El siguiente es un conjunto de caracteres que usted puede utilizar para formar un identificador válido. Los caracteres o símbolos que no sigan estas reglas no se pueden utilizar en los identificadores.

- Los caracteres de la `A` a la `Z` y de la `a` a la `z`.
- Los dígitos del `0` al `9` (aunque éstos no se pueden usar como el primer carácter de un identificador).
- El carácter de subrayado (_).

Por ejemplo, `signo_alto`, `Ciclo3` y `_pausa` son identificadores válidos.

Los siguientes son caracteres ilegales; esto es, no satisfacen el conjunto de reglas anterior para identificadores:

- Los signos aritméticos de C (`+`, `-`, `*`, `/`).
- El punto (`.`).
- Los apóstrofos (`'`) o comillas (`"`).
- Cualquier otro símbolo especial como `*`, `@`, `#`, `?`, `á`, etcétera.

Por ejemplo, algunos identificadores inválidos son `4banderas`, `suma-resultado`, `método*4`, y `¿qué_tamaño?`

Materiał chroniony prawem autorskim

> Nunca utilice las palabras reservadas o los nombres de funciones de la biblioteca estándar de C para nombres de variables o de funciones que cree en sus propios programas de C.

En la hora anterior vio la palabra clave `return`. La hora 4, "Tipos de datos y palabras reservadas", presenta una lista de todas las palabras reservadas.

Instrucciones

En el lenguaje C, una *instrucción* completa termina con un punto y coma. En muchos casos, usted puede convertir una expresión en una instrucción, simplemente agregando un punto y coma al final de la expresión.

Por ejemplo, la siguiente

```
i = 1;
```

es una instrucción. Ya habrá notado que la instrucción consta de la expresión `i = 1` y de un punto y coma (`;`).

Aquí tenemos otros ejemplos de instrucciones:

```
i = (2 + 3) * 10;
i = 2 + 3 * 10;
j = 6 % 4;
k = i + j;
```

Además, en la primera lección de este libro aprendió instrucciones como:

```
return 0;
exit(0);
printf ("¡Hola, amigo! Éste es mi primer programa de C.\n");
```

Bloques de instrucciones

Un grupo de instrucciones puede formar un *bloque de instrucciones* que comience con una llave de apertura (`{`) y termine con una llave de cierre (`}`). El compilador de C trata al bloque de instrucciones como a una sola instrucción.

Por ejemplo, el siguiente

```
for(. . .) {
   s3 = s1 + s2;
   mul = s3 * c;
   residuo = sum % c;
}
```

es un bloque de instrucciones que inicia con { y termina con }. Aquí, `for` es una palabra reservada de C que determina el bloque de instrucciones. La palabra reservada `for` se explica en la hora 7, "Ciclos".

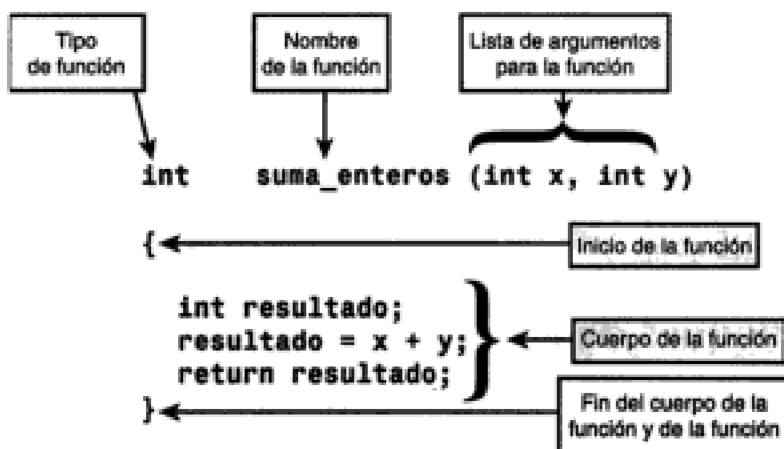
Un bloque de instrucciones proporciona una forma de agrupar una o más instrucciones como si fueran una sola. Muchas palabras reservadas de C sólo pueden controlar una instrucción. Si desea que una palabra reservada de C controle más de una instrucción, puede agregar dichas instrucciones dentro de un bloque de modo que la palabra reservada considere al bloque como una sola instrucción.

Anatomía de una función de C

Las funciones son los bloques de construcción de los programas de C. Además de las funciones de la biblioteca estándar de C, puede utilizar en su programa algunas otras funciones hechas por usted mismo o por otro programador. En la hora 2 vio la función `main()`, así como dos funciones de la biblioteca de C, `printf()` y `exit()`. A continuación veremos más de cerca las funciones.

Como se muestra en la figura 3.1, una función consta de seis partes: el tipo de función, el nombre de la función, los argumentos para la función, la llave de apertura, el cuerpo de la función y la llave de cierre.

FIGURA 3.1
La anatomía de una función del lenguaje C.



En las siguientes secciones se explican las seis partes de la función.

Determinación del tipo de una función

El *tipo de función* se usa para indicar el tipo de valor que devolverá la función después de su ejecución. Por ejemplo, en la hora 2 aprendió que el tipo predeterminado de la función `main()` es `int`.

En C, `int` se utiliza como palabra reservada para el tipo de datos entero. En la siguiente hora aprenderá más acerca de los tipos de datos.

Materiał chroniony prawem autorskim

Además del tipo `int`, una función puede ser de otros tipos, como el tipo carácter (palabra reservada: `char`), el tipo de punto flotante (`float`), etcétera. Más adelante aprenderá más acerca de estos tipos de funciones.

Cuando usted llama a una función de C, la cual devuelve un tipo de datos, el valor que devuelve (conocido como *valor de retorno o valor de devolución*) se puede utilizar en una expresión. Puede asignarlo a una variable, como por ejemplo:

```
int a = func();
```

o usarlo en una expresión como ésta:

```
a = func() + 7;
```

Cómo asignar un nombre válido a una función

El *nombre de una función* normalmente se asigna de tal forma que refleje lo que puede hacer ésta. Por ejemplo, el nombre de la función `printf()` significa "imprime datos con formato" (print formatted data).

Debido a que un nombre de función es un identificador, al crear sus propias funciones y darles nombre debe seguir las reglas para crear identificadores válidos.

Además, no puede utilizar los nombres de las funciones estándar de C, como `printf()` o `exit()`, para nombrar sus propias funciones. Las funciones estándar de C ya están definidas y es ilegal usar el mismo nombre para definir más de una función.

Paso de argumentos a las funciones de C

Las funciones son útiles debido a que usted puede llamarlas una y otra vez desde diferentes puntos de su programa. Sin embargo, es probable que cada vez que llame a una cierta función lo haga por razones ligeramente distintas.

Por ejemplo, en el listado 2.1 de la hora 2, la cadena de caracteres `"¡Hola, amigo! Éste es mi primer programa de C.\n"` se pasa a la función `printf()`, y luego `printf()` imprime la cadena en la pantalla. El único fin de `printf()` es imprimir una cadena en la pantalla, pero cada vez que usted llama a esta función, le pasa la cadena específica que desea imprimir en esa ocasión.

Las piezas de información que se pasan a las funciones se conocen como *argumentos*. Como ya vio, el argumento de una función se coloca entre los paréntesis que siguen al nombre de la función.

El número de argumentos de una función lo determina la declaración de dicha función, y la declaración la determina la tarea que va a realizar la función. Si una función necesita más de un argumento, los argumentos que se le pasen a la función deberán estar separados por comas; a estos argumentos se les considera una *lista de argumentos*.

Si no necesita pasar información a una función, debe dejar en blanco el campo de argumentos que está entre los paréntesis. Por ejemplo, la función `main()` del listado 2.1 de la hora 2 no tiene argumentos, así que el campo que está entre los paréntesis que siguen al nombre de la función está vacío. Vea esta copia del listado 2.1:

LISTADO 2.1 Un programa sencillo de C

```
1:  /* 02L01.c: Éste es mi primer programa de C */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      printf ("¡Hola, amigo! Éste es mi primer programa de C.\n");
7:      return 0;
8:  }
```

El principio y el final de una función

Como ya se habrá dado cuenta, se emplean llaves para señalar el inicio y el final de una función. La *llave de apertura* ({) indica el comienzo del cuerpo de una función, mientras que la *llave de cierre* (}) marca el fin del cuerpo de la función.

Como se mencionó antes, también se utilizan llaves para marcar el inicio y el final de un bloque de instrucciones. Puede pensar en ello como una extensión natural para usar llaves con las funciones, ya que una función consta de una o varias instrucciones.

El cuerpo de la función

En una función, el *cuerpo* de la misma es el lugar que contiene declaraciones de variables y otras instrucciones de C. La tarea de una función se lleva a cabo mediante la ejecución de las instrucciones, una a la vez, que se encuentran dentro de su cuerpo.

Es importante recordar que toda declaración de variable se debe colocar al principio del cuerpo de la función. Es ilegal poner declaraciones de variables en cualquier otra parte que no sea el inicio de un bloque de instrucciones.

Si el cuerpo de su función contiene declaraciones de variables, debe colocar dichas declaraciones antes que cualquier otra instrucción.

El listado 3.1 muestra una función que suma dos enteros especificados por sus argumentos, y devuelve el resultado de la suma.

Materiał chroniony prawem autorskim

LISTADO 3.1 Una función que suma dos enteros

```
1:  /* 03L01.c: Esta función suma dos enteros y devuelve el resultado */
2:  int suma_enteros ( int x, int y )
3:  {
4:     int resultado;
5:     resultado = x + y;
6:     return resultado;
7:  }
```

ANÁLISIS Como aprendió en la hora 2, la línea 1 del listado 3.1 es un comentario que describe lo que la función puede hacer.

Observe que en la línea 2 se pone el tipo de datos `int` como prefijo, antes del nombre de la función. Aquí, `int` se utiliza como el tipo de función, lo que significa que la función devuelve un entero. El nombre de la función que se muestra en la línea 2 es `suma_enteros`. La lista de argumentos de la misma línea contiene dos argumentos, `int x` e `int y`, y el tipo de datos `int` especifica que ambos argumentos son enteros.

La línea 3 contiene la llave de apertura (`{`) que marca el inicio de la función.

El cuerpo de la función comprende las líneas 4 a 6 del listado 3.1. La línea 4 presenta la declaración de la variable `resultado`, la cual está especificada por el tipo de datos `int` como un entero. La instrucción de la línea 5 suma los dos enteros representados por `x` y `y`, y asigna el resultado del cálculo a la variable `resultado`. La instrucción `return` de la línea 6 devuelve entonces el resultado del cálculo representado por `resultado`.

Por último, pero no por ello menos importante, se usa la llave de cierre (`}`) de la línea 7 para cerrar la función.

Al crear una función en su propio programa de C, no le asigne demasiado trabajo. Si una función tiene mucho que hacer, será muy difícil de escribir y depurar. Si tiene un proyecto de programación complejo, divídalo en piezas pequeñas. Procure asegurarse de que cada función realice solamente una tarea.

Cómo hacer llamadas a funciones

Con base en lo que hasta ahora ha aprendido, puede escribir un programa en C que llame a la función `suma_enteros()` para calcular una suma e imprimir el resultado en la pantalla. En el listado 3.2 se muestra un ejemplo de dicho programa.

Listado 3.2 Un programa de C que calcula una suma e imprime el resultado en la pantalla

```
1:  /* 03L02.c: Calcula una suma e imprime el resultado */
2:  #include <stdio.h>
3:  /* Esta función suma dos enteros y devuelve el resultado */
4:  int suma_enteros( int x, int y )
5:  {
6:     int resultado;
7:     resultado = x + y;
8:     return resultado;
9:  }
10:
11: int main()
12: {
13:    int suma;
14:
15:    suma = suma_enteros(5, 12);
16:    printf("La suma de 5 y 12 es %d.\n", suma);
17:    return 0;
18: }
```

El programa del listado 3.2 se guarda como un archivo fuente llamado `03L02.c`. Después de compilarlo y enlazarlo, se crea un archivo ejecutable para `03L02.c`. En mi máquina, el archivo ejecutable se llama `03L02.exe`. La siguiente es la salida que se imprime en la pantalla después de ejecutar dicho archivo en mi máquina:

Salida

```
La suma de 5 y 12 es 17.
```

Análisis La línea 1 del listado 3.2 es un comentario acerca del programa. Como aprendió en la hora 2, la directiva `include` de la línea 2 incluye el archivo de encabezado `stdio.h` debido a que en el programa se usa la función `printf()`.

Las líneas 3 a 9 representan a la función `suma_enteros()` que, como se explicó en la sección anterior, suma dos enteros.

La función `main()`, con el tipo de datos `int` como prefijo, inicia en la línea 11. Las líneas 12 y 18 contienen las llaves de apertura y cierre de la función `main()`, respectivamente. En la línea 13 se declara una variable entera, `suma`.

La instrucción de la línea 15 llama a la función `suma_enteros()` que usted examinó en la sección anterior. Observe que las dos constantes enteras, `5` y `12`, se pasan a la función `suma_enteros()`, y que a la variable `suma` se le asigna el resultado devuelto por la función `suma_enteros()`.

Materiał chroniony prawem autorskim

En la hora 2 vio la función `printf()` de la biblioteca estándar de C. Si piensa que encontró algo nuevo agregado a la función de la línea 16, tiene razón. Esta vez se le pasaron dos argumentos a la función `printf()`. Ellos son la cadena `"La suma de 5 y 12 es %d.\n"` y la variable `suma`.

Observe que en el primer argumento se agrega un nuevo símbolo, `%d`. El segundo argumento es la variable entera `suma`. Debido a que el valor de `suma` se imprimirá en la pantalla, usted podría pensar que `%d` tiene algo que ver con la variable entera `suma`. Otra vez tiene razón. `%d` indica a la computadora el formato en el que debe imprimir a `suma` en la pantalla.

En la hora 4 se abordan más detalles sobre `%d`. La relación entre `%d` y `suma` se explica en la hora 5, "Manejo de la entrada y salida estándar".

Pero sobre todo, es importante que se concentre en el programa del listado 3.2 y preste atención a cómo hacer una llamada, ya sea a una función generada por el usuario o a una función de la biblioteca estándar de C, desde la función `main()`.

Resumen

En esta lección aprendió los siguientes conceptos y operadores:

- En C, una constante es un valor que nunca cambia. Por otra parte, una variable puede presentar diferentes valores.
- En el lenguaje C, a una combinación de constantes, variables y operadores se le llama expresión. Una expresión se utiliza para denotar diferentes cálculos.
- Los operadores aritméticos incluyen `+`, `-`, `*`, `/` y `%`.
- Una instrucción consta de una expresión completa con un punto y coma como sufijo.
- El compilador de C trata a un bloque de instrucciones como a una sola instrucción, aunque el bloque podría contener más de una instrucción.
- El tipo de función que se especifica en la declaración de una función determina el tipo de valor que devuelve dicha función.
- Debe seguir ciertas reglas para formar un nombre de función válido.
- Un argumento contiene información que usted desea pasar a una función. Una lista de argumentos contiene dos o más argumentos separados por comas.
- Las llaves de apertura (`{`) y de cierre (`}`) se usan para marcar el inicio y el final de una función de C.
- El cuerpo de una función contiene declaraciones de variables e instrucciones.
- Por lo regular, una función debe llevar a cabo sólo una tarea.

En la siguiente lección aprenderá más acerca de los tipos de datos del lenguaje C.

Preguntas y respuestas

P ¿Cuál es la diferencia entre una constante y una variable?

R La principal diferencia es que el valor de una constante no se puede modificar, mientras que el de una variable sí. En su programa de C puede asignar diferentes valores a una variable siempre que sea necesario.

P ¿Por qué necesita un bloque de instrucciones?

R Muchas palabras reservadas de C sólo pueden controlar una instrucción. Un bloque de instrucciones proporciona una forma de poner varias instrucciones juntas, y de que una palabra reservada de C controle el bloque de instrucciones. Entonces, el bloque es tratado como una sola instrucción.

P ¿Qué operadores aritméticos tienen una precedencia más alta?

R De entre los cinco operadores aritméticos, los operadores de multiplicación, división y residuo tienen una precedencia más alta que los operadores de suma y resta.

P ¿Cuántas partes tiene normalmente una función?

R Normalmente, una función tiene seis partes: el tipo de la función, el nombre de la función, los argumentos, la llave de apertura, el cuerpo de la función y la llave de cierre.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Es `74` una constante en el lenguaje C?, ¿qué hay de `571`?
2. ¿Es `x = 570 + 1` una expresión?, ¿lo es `x = 12 + y`?
3. ¿Son válidos los siguientes nombres de función?

```
2métodos
algoritmo_m2
*función_inicio
Espacio_Cuarto
.Termina_Exe
_suma_turbo
```

4. ¿Es `2 + 5 * 2` igual a `(2 + 5) * 2`?
5. ¿Produce `7 % 2` el mismo resultado que `4 % 3`?

Materiał chroniony prawem autorskim

Ejercicios

1. Dadas las dos instrucciones, `x = 3;` y `y = 5 + x;`, ¿cómo puede generar con ambas un bloque de instrucciones?
2. ¿Qué está mal en la siguiente función?

```
int 3suma_enteros( int x, int y, int z)
{
   int suma;
   suma = x + y + z;
   return suma;
}
```

3. ¿Qué está mal en la siguiente función?

```
int suma_enteros( int x, int y, int z)
{
   int suma;
   suma = x + y + z
   return suma;
}
```

4. Escriba una función de C que pueda multiplicar dos enteros y devolver el resultado.
5. Escriba un programa en C que llame a la función de C que acaba de escribir en el ejercicio 4, para calcular la multiplicación de 3 por 5 e imprimir después en la pantalla el valor de retorno de la función.

HORA 4

Tipos de datos y palabras reservadas

¿Qué es un nombre? Aquello que llamamos rosa olería dulce con cualquier otro nombre.

—W. Shakespeare

En la hora 3, "La estructura de un programa de C", aprendió cómo formar un nombre válido para una función de C. Ahora aprenderá más acerca de cómo nombrar una variable y de las palabras reservadas del compilador de C.

Además, en esta hora aprenderá detalladamente los cuatro tipos de datos del lenguaje C:

- el tipo de datos `char`
- el tipo de datos `int`
- el tipo de datos `float`
- el tipo de datos `double`

Palabras reservadas de C

El lenguaje C reserva ciertas palabras que tienen un significado especial para el lenguaje. Estas palabras reservadas también se conocen como *palabras clave de C*. No debe utilizar en sus programas palabras reservadas de C para nombrar sus propias variables, constantes o funciones. La tabla 4.1 presenta una lista de 32 palabras reservadas de C.

TABLA 4.1 Palabras reservadas de C

Palabra reservada	*Descripción*
`auto`	Especificador de clase de almacenamiento
`break`	Instrucción
`case`	Instrucción
`char`	Especificador de tipo
`const`	Modificador de clase de almacenamiento
`continue`	Instrucción
`default`	Etiqueta
`do`	Instrucción
`double`	Especificador de tipo
`else`	Instrucción
`enum`	Especificador de tipo
`extern`	Especificador de clase de almacenamiento
`float`	Especificador de tipo
`for`	Instrucción
`goto`	Instrucción
`if`	Instrucción
`int`	Especificador de tipo
`long`	Especificador de tipo
`register`	Especificador de clase de almacenamiento
`return`	Instrucción
`short`	Especificador de tipo
`signed`	Especificador de tipo
`sizeof`	Operador
`static`	Especificador de clase de almacenamiento
`struct`	Especificador de tipo
`switch`	Instrucción

Materiał chroniony prawem autorskim

Palabra reservada	*Descripción*
`typedef`	Instrucción
`union`	Especificador de tipo
`unsigned`	Especificador de tipo
`void`	Especificador de tipo
`volatile`	Modificador de clase de almacenamiento
`while`	Instrucción

No se preocupe si no puede recordar a la primera todas las palabras reservadas de C. En el resto del libro se familiarizará más con ellas y comenzará a usarlas en los ejemplos y ejercicios.

Observe que todas las palabras reservadas de C están escritas en minúsculas. Como ya mencioné, C es un lenguaje sensible al uso de mayúsculas y minúsculas. Por lo tanto, como se muestra en la lista, `int` es una palabra reservada de C, pero `INT` no lo es.

El tipo de datos char

Un objeto del tipo de datos `char` representa un solo carácter del conjunto de caracteres que utiliza su computadora. Por ejemplo, `A` es un carácter, y `a` también lo es. Pero `7` es un número.

Sin embargo, una computadora sólo puede almacenar código numérico. Por lo tanto, los caracteres como `A`, `a`, `B`, `b`, etcétera, tienen un código numérico único que utilizan las computadoras para representar los caracteres. Por lo regular, un carácter ocupa 8 bits (es decir, 1 byte) para almacenar su código numérico.

Para muchas computadoras, los códigos ASCII son los códigos estándar para representar un conjunto de caracteres. (Sólo para su información, ASCII significa Código Estándar Estadounidense para el Intercambio de Información.) El conjunto de caracteres ASCII original tiene solamente 128 caracteres debido a que utiliza los 7 bits menos significativos con los que se pueden representar 2^7 caracteres (es decir, 128).

Sin embargo, en las PCs compatibles con IBM, el conjunto de caracteres se amplió para contener un total de 256 caracteres (esto es, 2^8).

Algo que quisiera mencionar aquí es que el estándar ANSI de C especifica sólo el valor del carácter nulo, el cual siempre es cero (es decir, un byte con ceros en todos los bits). Los demás valores numéricos de los caracteres los determinan los tipos de computadoras, sistemas operativos y compiladores de C. Le recomiendo que explore el conjunto de caracteres de su computadora. Puede hacerlo con el programa del listado 4.1.

En el mundo de la computación, un bit es la unidad más pequeña de almacenamiento de datos, y sólo puede tener uno de estos dos valores: 0 o 1. Estos valores representan los dos estados de los interruptores electrónicos que se usan en la memoria y en la CPU de la computadora. Un byte es una unidad más grande que un bit. De hecho, ocho bits equivalen a un byte.

Variables de tipo carácter

Una variable que puede representar diferentes caracteres se llama *variable de tipo carácter*.

Puede asignar el tipo de datos `char` a una variable mediante el siguiente formato de declaración:

```
char  nombrevariable;
```

en donde `nombrevariable` es el nombre que usted proporciona, y en el que se almacenarán valores de este tipo.

Si tiene más de una variable que declarar, puede utilizar el siguiente formato:

```
char  nombrevariable1;
char  nombrevariable2;
char  nombrevariable3;
```

o bien, este otro:

```
char  nombrevariable1, nombrevariable2, nombrevariable3;
```

Por ejemplo, la siguiente instrucción declara `MiCaracter` y le asigna el valor de `'A'`:

```
char MiCaracter = 'A';
```

Del mismo modo, las siguientes instrucciones declaran `x`, `y` y `z` como variables de tipo carácter y después les asignan valores:

```
char x, y, z;
x = 'A';
y = 'f';
z = '7';
```

Observe que la última instrucción, `z = '7'`, asigna a `z` el valor numérico que representa el carácter `'7'` en el conjunto de caracteres, no el número 7 real.

Más adelante aprenderá más acerca de las variables de tipo carácter y cómo usarlas en sus programas de C.

Constantes de carácter

Un carácter encerrado entre comillas sencillas (`'`) se llama *constante de carácter*. Por ejemplo, `'A'`, `'a'`, `'B'` y `'b'`, son constantes de carácter que tienen sus valores numéri-

Materiał chroniony prawem autorskim

cos únicos en un conjunto de caracteres dado. Por ejemplo, podría ver los valores numéricos únicos del conjunto de caracteres ASCII.

Es importante recordar que las constantes de carácter siempre están encerradas entre comillas sencillas ('), mientras que una cadena de más de un carácter utiliza comillas dobles ("). Si esto le parece confuso, sólo recuerde que las comillas sencillas van con un solo carácter. En la hora anterior vio un ejemplo de comillas dobles y cadenas de caracteres en las llamadas a la función `printf()`.

En el conjunto de caracteres ASCII encontrará que los valores numéricos (decimales) únicos de `'A'`, `'a'`, `'B'` y `'b'` son 65, 97, 66 y 98, respectivamente. Por lo tanto, dada x como una variable de tipo carácter, y dado el conjunto de caracteres ASCII, las siguientes instrucciones de asignación son equivalentes:

```
x = 'A';
x = 65;
```

También lo son las dos siguientes:

```
x = 'a';
x = 97;
```

Más adelante, en el listado 4.2, verá un programa que convierte los valores numéricos en sus caracteres correspondientes.

No confunda `x = 'a';` con `x = a;`. La primera instrucción le asigna a la variable x el valor numérico del carácter a, esto es, x contendrá el valor 97 después de la asignación (el valor ASCII de la letra 'a'). Sin embargo, la instrucción `x = a;` le asigna a la variable x cualquier valor contenido en la variable a. Posteriormente aprenderá más acerca de esta diferencia.

El carácter de escape (\)

En realidad, usted vio el carácter de escape (\) en la hora 2, "Su primer programa de C", cuando aprendió a usar el carácter de nueva línea (\n) para dividir un mensaje en dos partes. Por lo tanto, en el conjunto de caracteres ASCII, la diagonal invertida (\) se llama *carácter de escape*. Este carácter se emplea en el lenguaje C para indicar a la computadora que sigue un carácter especial.

Por ejemplo, cuando la computadora ve \ en el carácter de nueva línea \n, sabe que el siguiente carácter, n, genera una secuencia de retorno de carro y un salto de línea.

Además del carácter de nueva línea, los siguientes son algunos de los caracteres especiales del lenguaje C:

Carácter	*Descripción*
`\b`	El carácter de retroceso; mueve el cursor un carácter hacia la izquierda.
`\f`	El carácter de salto de página; pasa a la parte superior de la página siguiente.
`\r`	El carácter de retorno; regresa al principio de la línea actual.
`\t`	El carácter tabulador; avanza al siguiente paro de tabulador.

Impresión de caracteres

Usted ya sabe que la función `printf()`, definida en el archivo de encabezado `stdio.h` de C, se puede utilizar para imprimir mensajes en la pantalla (consulte el listado 2.1 de la hora 2). En esta sección aprenderá a usar el especificador de formato de carácter, %c, el cual indica a la función `printf()` que el argumento por imprimir es un carácter. (Aprenderá más acerca del especificador de formato en la hora 5, "Manejo de la entrada y salida estándar". Aquí sólo tendrá una probadita.) Por ahora, lo importante es que sepa que cada especificador de formato que pase a `printf()` corresponderá a una de las variables que pase a la función. Veamos primero el programa del listado 4.1, el cual imprime caracteres en la pantalla.

TECLEE **LISTADO 4.1** Impresión de caracteres en la pantalla

```
1:  /* 04L01.c: Impresión de caracteres */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char c1;
7:     char c2;
8:
9:     c1 = 'A';
10:    c2 = 'a';
11:    printf("Convierte el valor de c1 a carácter: %c.\n", c1);
12:    printf("Convierte el valor de c2 a carácter: %c.\n", c2);
13:    return 0;
14: }
```

Después de crear el archivo ejecutable de `04L01.c` del listado 4.1, puede ejecutarlo para ver qué se imprimirá en la pantalla. En mi máquina, el archivo ejecutable se llama `04L01.exe`. El siguiente es el resultado que se imprimió en la pantalla de mi computadora después de ejecutar el programa:

Materiał chroniony prawem autorskim

SALIDA

```
Convierte el valor de c1 a carácter: A.
Convierte el valor de c2 a carácter: a.
```

ANÁLISIS Como usted sabe, la línea 2 incluye el archivo de encabezado, `stdio.h`, para la función `printf()`. Las líneas 5 a 14 conforman el cuerpo de la función `main()`.

Las líneas 6 y 7 declaran dos variables de tipo carácter, `c1` y `c2`, mientras que las líneas 9 y 10 asignan a `c1` y a `c2` las constantes correspondientes a los caracteres `'A'` y `'a'`, respectivamente.

Observe que en las líneas 11 y 12 se usa el especificador de formato `%c` en la función `printf()`, lo cual indica a la computadora que el contenido de `c1` y `c2` se debe imprimir como carácter. Al ejecutar las dos instrucciones de las líneas 11 y 12, se da formato a dos caracteres y se muestran en la pantalla, con base en los valores numéricos contenidos por `c1` y `c2`, respectivamente.

Ahora observe el programa que se muestra en el listado 4.2. Esta vez, `%c` se usa para convertir los valores numéricos en sus caracteres correspondientes.

TECLEE

LISTADO 4.2 Conversión de valores numéricos en caracteres

```
1:  /* 04L02.c: Convierte valores numéricos en caracteres */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char c1;
7:     char c2;
8:
9:     c1 =65;
10:    c2 =97;
11:    printf("El carácter que tiene el valor numérico de 65 es: %c.\n", c1);
12:    printf("El carácter que tiene el valor numérico de 97 es: %c.\n", c2);
13:    return 0;
14: }
```

El siguiente es el resultado que se imprimió en la pantalla de mi computadora después de ejecutar el archivo `04L02.exe`. (Usted podría obtener un resultado diferente en su computadora; depende de la implementación. Esto es, depende del tipo de su computadora, del sistema operativo y del compilador C que esté utilizando):

SALIDA

```
El carácter que tiene el valor numérico de 65 es: A.
El carácter que tiene el valor numérico de 97 es: a
```

ANÁLISIS El programa del listado 4.2 es similar al del listado 4.1, con excepción de las dos instrucciones de las líneas 9 y 10. Observe que en dichas líneas del listado 4.2, a las variables de tipo carácter `c1` y `c2` se les asignan los valores `65` y `97`, respectivamente.

Como usted sabe, en el conjunto de caracteres ASCII, `65` es el valor numérico (decimal) del carácter `A`; `97` es el valor numérico de `a`. En las líneas 11 y 12, el especificador de formato `%c` convierte los valores numéricos, `65` y `97`, en `A` y `a`, respectivamente. Entonces, los caracteres `A` y `a` se imprimen en la pantalla.

El tipo de datos `int`

En la hora 3 vio el tipo de datos entero. La palabra reservada `int` se utiliza para especificar que una variable es de tipo entero. Los números enteros, llamados simplemente *enteros*, no tienen fracciones ni punto decimal. Por lo tanto, el resultado de una división entera se trunca, simplemente porque se ignora cualquier parte de fracción.

La longitud de un entero varía dependiendo del sistema operativo y del compilador de C que utilice. Por ejemplo, en la mayoría de las estaciones de trabajo UNIX, un entero tiene una longitud de 32 bits, lo que significa que el rango de un entero va de `2147483647` (esto es, $2^{31}-1$) a `-2147483648`. Para un entero de 16 bits, el rango va de `32767` (esto es, $2^{15}-1$) a `-32768`. Como lo dije anteriormente, esto puede variar entre sistemas diferentes, de modo que para estar seguro puede revisar los materiales de referencia de su compilador.

Algunos compiladores de C, como el Visual C++ 1.5, sólo proporcionan enteros de 16 bits, mientras que otros compiladores de C de 32 bits, como el Visual C++ 5.0, soportan enteros de 32 bits.

Declaración de variables enteras

También vio la declaración de un entero en la hora 3. El siguiente es el formato básico de declaración:

```
int  nombrevariable;
```

Al igual que en la declaración de las variables de tipo carácter, si tiene más de una variable que declarar, puede utilizar un formato como éste:

```
int  nombrevariable1;
int  nombrevariable2;
int  nombrevariable3;
```

o bien, uno como éste:

```
int  nombrevariable1, nombrevariable2, nombrevariable3;
```

Aquí, *`nombrevariable1`*, *`nombrevariable2`* y *`nombrevariable3`* indican las posiciones en las que usted coloca los nombres de variables de tipo `int`.

Por ejemplo, la siguiente instrucción declara `MiEntero` como una variable entera y le asigna un valor:

```
int MiEntero = 2314;
```

Materiał chroniony prawem autorskim

Del mismo modo, la siguiente instrucción declara `A, a, B` y `b` como variables enteras:

```
int A, a, B, b;
A = 37;
a = -37;
B = -2418;
b = 12 ;
```

Más adelante aprenderá más acerca del tipo de datos entero.

Cómo mostrar los valores numéricos de los caracteres

Al igual que el especificador de formato de carácter (`%c`) que se usa para dar formato a un solo carácter, el *especificador de formato entero*, `%d`, se usa para dar formato a un entero. Tal vez recuerde que en la línea 16 del listado 3.2, se usó `%d` para dar formato de entero al segundo argumento de la función `printf()`.

En esta sección estudiará el programa que se muestra en el listado 4.3, el cual puede imprimir los valores numéricos de los caracteres utilizando el especificador de formato entero `%d` en la función `printf()`.

TECLEE

LISTADO 4.3 Cómo mostrar los valores numéricos de los caracteres

```
1:  /* 04L03.c: Cómo mostrar el valor numérico de los caracteres */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char c1;
7:     char c2;
8:
9:     c1 = 'A';
10:    c2 = 'a';
11:    printf("El valor numérico de A es: %d.\n", c1);
12:    printf("El valor numérico de a es: %d.\n", c2);
13:    return 0;
14: }
```

Obtuve el siguiente resultado en la pantalla de mi computadora, después de ejecutar el archivo `04L03.exe`. (Usted podría obtener una salida diferente, si su máquina no emplea el conjunto de caracteres ASCII.)

SALIDA

```
El valor numérico de A es: 65.
El valor numérico de a es: 97.
```

ANÁLISIS

Tal vez encuentre que el programa del listado 4.3 es muy similar al del listado 4.1. De hecho, simplemente copié el código fuente del listado 4.1 al listado 4.3 e hice cambios en las líneas 11 y 12. El principal cambio que realice fue reemplazar el especificador de formato de carácter (`%c`) por el especificador de formato entero (`%d`).

Ambos especificadores de formato hacen básicamente lo mismo: insertar ciertos datos en la cadena que usted pasa a `printf()`, pero la diferencia está en la forma en que `printf()` muestra los datos. El especificador `%c` siempre imprime un carácter; el especificador `%d` siempre imprime un número. Incluso cuando se refieran exactamente a los mismos datos, éstos se imprimirán en la forma que se indique en el especificador de formato sin importar el tipo de datos real.

Las dos instrucciones de las líneas 11 y 12 dan formato a las dos variables de tipo carácter (`c1` y `c2`) utilizando el especificador de formato entero `%d`, y después imprimen dos mensajes que muestran los valores numéricos `65` y `97` que representan, respectivamente, a los caracteres `A` y `a` del conjunto de caracteres ASCII.

El tipo de datos `float`

El *número de punto flotante* es otro tipo de datos del lenguaje C. A diferencia de un número entero, un número de punto flotante contiene un punto decimal. Por ejemplo, 7.01 es un número de punto flotante, así como también 5.71 y –3.14. Los números de punto flotante también se conocen como *números reales.*

En el lenguaje C, la palabra reservada `float` especifica un número de punto flotante. Las constantes de punto flotante pueden tener el sufijo `f` o `F` para especificar `float`. De manera predeterminada, un número de punto flotante sin un sufijo es de tipo `double`. El tipo de datos `double` se presenta más adelante en esta lección.

Al igual que un número entero, un número de punto flotante tiene un rango limitado. El estándar ANSI requiere que el rango sea de -1.0×10^{37} a 1.0×10^{37}. En la mayoría de los casos, un número de punto flotante es de 32 bits. Por lo tanto, en C, un número de punto flotante es de por lo menos seis dígitos de precisión. Esto es, un número de punto flotante tiene por lo menos seis dígitos (o posiciones decimales) después del punto decimal.

A diferencia de la división entera, en la que se trunca el resultado y se descarta la fracción, una división de punto flotante produce otro número de punto flotante. Una división de punto flotante se lleva a cabo si el divisor y el dividendo, o sólo uno de ellos, son números de punto flotante.

Por ejemplo, `571.2 / 10.0` produce otro número de punto flotante, `57.12`. Lo mismo sucede con `571.2 / 10` y `5712 / 10.0`.

Declaración de variables de punto flotante

El siguiente es el formato de declaración de una variable de punto flotante:

```
float  nombrevariable;
```

Materiał chroniony prawem autorskim

Igual que en la declaración de una variable de tipo entero o de carácter, si tiene más de una variable que declarar, puede emplear un formato como éste:

```
float  nombrevariable1;
float  nombrevariable2;
float  nombrevariable3;
```

o bien, uno como el siguiente:

```
float  nombrevariable1, nombrevariable2, nombrevariable3;
```

Por ejemplo, la siguiente instrucción declara `miFloat` como una variable de punto flotante y le asigna un valor:

```
float miFloat = 3.14;
```

Del mismo modo, la siguiente instrucción declara `a`, `b` y `c` como variables de tipo `float`:

```
float a, b, c;
a = 10.38;
b = -32.7;
c = 12.0f;
```

El especificador de formato de punto flotante (%f)

También puede emplear el *especificador de formato de punto flotante* (`%f`) para dar formato a su salida. El listado 4.4 muestra un ejemplo de cómo usar el especificador de formato `%f` en la función `printf()`.

TECLEE **LISTADO 4.4** Impresión de resultados de divisiones entera y de punto flotante

```
1:  /* 04L04.c: División entera vs. división de punto flotante */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int int_num1, int_num2, int_num3;   /* Declara variables enteras */
7:     float flt_num1, flt_num2, flt_num3; /* Declara variables de punto
       flotante */
8:
9:     int_num1 = 32 / 10;    /* Tanto el divisor como el dividendo son
       enteros */
10:    flt_num1 = 32 / 10;
11:    int_num2 = 32.0 / 10;  /* El divisor es un entero */
12:    flt_num2 = 32.0 / 10;
13:    int_num3 = 32 / 10.0;  /* El dividendo es un entero */
14:    flt_num3 = 32 / 10.0;
15:
16:    printf("La división entera de 32/10 es: %d\n", int_num1);
```

continúa

LISTADO 4.4 continuación

```
17:     printf("La división de punto flotante de 32/10 es: %f\n", flt_num1);
18:     printf("La división entera de 32.0/10 es: %d\n", int_num2);
19:     printf("La división de punto flotante de 32.0/10 es: %f\n", flt_num2);
20:     printf("La división entera de 32/10.0 es: %d\n", int_num3);
21:     printf("La división de punto flotante de 32/10.0 es: %f\n", flt_num3);
22:     return 0;
23: }
```

El siguiente es el resultado que apareció en la pantalla de mi computadora después de ejecutar el archivo `04L04.exe`:

Me aparecieron varios mensajes de advertencia acerca de las conversiones al compilar el programa del listado 4.4, pero los ignoré todos porque quería crear un archivo ejecutable con el fin de mostrarle las diferencias entre el tipo de datos `int` y el tipo de datos `float`.

SALIDA

```
La división entera de 32/10 es: 3
La división de punto flotante de 32/10 es: 3.000000
La división entera de 32.0/10 es: 3
La división de punto flotante de 32.0/10 es: 3.200000
La división entera de 32/10.0 es: 3
La división de punto flotante de 32/10.0 es: 3.200000
```

ANÁLISIS Dentro de la función `main()`, las dos instrucciones de las líneas 6 y 7 declaran tres variables enteras, `int_num1`, `int_num2` e `int_num3`, y tres variables de punto flotante, `flt_num1`, `flt_num2` y `flt_num3`.

En las líneas 9 y 10 se asigna el resultado de `32/10` a `int_num1` y `flt_num1`, respectivamente; en las líneas 11 y 12 el de `32.0/10` a `int_num2` y `flt_num2`, y en las líneas 13 y 14 el de `32/10.0` a `int_num3` y `flt_num3`.

Luego, en las líneas 16 a 21 se imprimen los resultados de los valores contenidos por las tres variables enteras y las tres variables de punto flotante. Observe que en la función `printf()` se usa `%d` para las variables enteras, y para dar formato a las variables de punto flotante se usa el especificador de formato de punto flotante (`%f`).

Debido al truncamiento que ocurre en la división entera de `32/10`, `flt_num1` contiene `3.000000`, y no `3.200000`, como puede ver en la segunda línea de la salida. Sin embargo, a `flt_num2` y `flt_num3` se les asigna `3.200000` debido a que tanto `32.0/10` como `32/10.0` se consideran divisiones de punto flotante.

Materiał chroniony prawem autorskim

Sin embargo, `int_num2` e `int_num3`, como variables enteras, descartan respectivamente la fracción de las divisiones de punto flotante de `32.0/10` y `32/10.0`. Por lo tanto, usted sólo ve el entero `3` en la tercera y quinta líneas de la salida.

El tipo de datos `double`

En el lenguaje C, un número de punto flotante también se puede representar por medio de otro *tipo de datos*, denominado `double`. En otras palabras, puede especificar una variable usando la palabra reservada `double`, y asignar a dicha variable un número de punto flotante.

La diferencia entre el tipo de datos `double` y el tipo de datos `float` es que el primero utiliza el doble de bits que el segundo. Por lo tanto, un número de punto flotante `double` es de por lo menos 10 dígitos de precisión, aunque el estándar ANSI no lo especifica para el tipo de datos `double`.

En la hora 8, "Uso de operadores condicionales", aprenderá a usar el operador `sizeof` para obtener la longitud en bytes de un tipo de datos especificado en su sistema de cómputo, como `char`, `int`, `float` o `double`.

4

Uso de la notación científica

El lenguaje C usa la *notación científica* para ayudarle a escribir números de punto flotante muy grandes.

En la notación científica se puede representar un número por medio de la combinación de la *mantisa* y el *exponente*. En el formato de la notación, la mantisa va seguida del exponente, el cual tiene como prefijo una `e` o una `E`. Aquí tenemos dos ejemplos:

`mantisaeexponente`

y

`mantisaEexponente`

Observe que tanto `mantisa` como `exponente` son indicadores de posición y debe reemplazarlos con valores numéricos.

Por ejemplo, en notación científica, `5000` se puede representar como `5e3`. De la misma manera, `-300` se puede representar como `-3e2`, y `0.0025` como `2.5e-3`.

De acuerdo con lo anterior, en la notación científica se emplean los especificadores de formato `%e` o `%E` para dar formato a un número de punto flotante. En la función `printf()`, `%e` o `%E` se emplean de la misma manera que `%f`.

Denominación de una variable

Hay algunas cosas que debe tener presentes al crear sus propios nombres de variables. Recuerde que los nombres de variables son identificadores; por lo tanto, debe cumplir las reglas descritas para ellos en la hora anterior.

Así como los nombres de función deben reflejar perfectamente la tarea que realiza la función, los nombres de variables deben describir el valor almacenado en la variable y la finalidad que ésta tiene en su programa o función. Hasta ahora, en la mayoría de los ejemplos de código hemos utilizado nombres de variables de una sola letra, como `i`, pero conforme escriba programas más extensos y funciones más complicadas, va a ser necesario que dé a sus variables nombres significativos.

> En su programa de C, nunca utilice como nombres de variables las palabras reservadas del lenguaje C, ni los nombres de las funciones de la biblioteca estándar de C.

Resumen

En esta lección aprendió algo acerca de las siguientes palabras reservadas y tipos de datos de C:

- Las palabras reservadas del lenguaje C
- El tipo de datos `char` y el especificador de formato `%c`
- El tipo de datos `int` y el especificador de formato `%d`
- El tipo de datos `float` y el especificador de formato `%f`
- Los números de punto flotante pueden tener los sufijos `f` o `F` para especificar `float`. Un número de punto flotante sin un sufijo es, de manera predeterminada, de tipo `double`
- Los rangos posibles de los tipos de datos `char`, `int` y `float`
- El tipo de datos `double`
- La notación científica y los especificadores de formato `%e` y `%E`
- Las reglas que tiene que seguir para crear un nombre de variable válido

En la siguiente lección aprenderá más acerca de la función `printf()` y de otras funciones para el manejo de entrada y salida.

Preguntas y respuestas

P ¿Por qué los caracteres tienen valores numéricos únicos?

R Los caracteres se almacenan en las computadoras en forma de bits. Las combinaciones de bits se pueden utilizar para representar diferentes valores numéricos.

Materiał chroniony prawem autorskim

Un carácter debe tener un valor numérico único a fin de distinguirlo de los demás. Muchos sistemas computacionales manejan el conjunto de caracteres ASCII extendido, el cual contiene un conjunto de valores numéricos únicos hasta para 256 caracteres.

Observe que su computadora podría utilizar un conjunto de caracteres diferente al ASCII. En ese caso, todos los caracteres que emplea el lenguaje C, con excepción del carácter nulo, podrían tener valores numéricos diferentes a los representados por el conjunto de caracteres ASCII.

P ¿Cómo puedo declarar dos variables de tipo carácter?

R Hay dos formas de hacer la declaración. La primera es:

```
char  nombrevariable1, nombrevariable2;
```

La otra forma es

```
char  nombrevariable1;
char  nombrevariable2;
```

P ¿Qué son `%c`, `%d` y `%f`?

R Son especificadores de formato. `%c` se utiliza para obtener el formato de carácter; `%d` es para el formato entero y `%f` es para el formato de punto flotante. `%c`, `%d` y `%f` se utilizan a menudo con funciones de C como `printf()`.

P ¿Cuáles son las principales diferencias entre el tipo de datos `int` (entero) y el tipo de datos `float` (de punto flotante)?

R En primer lugar, un entero no contiene fracciones, y un número de punto flotante sí las tiene. Un número de punto flotante debe tener un punto decimal. En C, el tipo de datos `float` ocupa más bits que el tipo de datos `int`. En otras palabras, el tipo de datos `float` tiene un rango más grande de valores numéricos que el tipo de datos `int`.

Además, la división entera trunca la fracción. Por ejemplo, la división entera de `16/10` es `1`, no `1.6`.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Son iguales las divisiones enteras de `134/100` y `17/10`?
2. ¿Es un número de punto flotante el resultado de `3000 + 1.0`? ¿Y el de `3000/1.0`?

3. ¿Cómo puede representar los siguientes números en notación científica?
   - 3500
   - 0.0035
   - –0.0035
4. ¿Son válidos los siguientes nombres de variables?
   - `7o_Calculo`
   - `Método_de_Tom`
   - `_marca`
   - `Etiqueta_1`

Ejercicios

1. Escriba un programa que imprima los valores numéricos de los caracteres `Z` y `z`.
2. Dados los valores numéricos `72` y `104`, escriba un programa que imprima los dos caracteres correspondientes.
3. ¿Se puede asignar el valor de `72368` a una variable entera de 16 bits?
4. Dada la declaración `double dbl_num = 123.456;`, escriba un programa que imprima el valor de `dbl_num` en los formatos tanto de punto flotante como de notación científica.
5. Escriba un programa que pueda imprimir el valor numérico del carácter de nueva línea (`\n`). (Sugerencia: asigne `'\n'` a una variable de tipo carácter.)

Materiał chroniony prawem autorskim

HORA 5

Manejo de la entrada y salida estándar

E/S, E/S, es hora de trabajar...

—Los siete enanos (o algo así)

En la lección anterior aprendió cómo imprimir caracteres y números enteros y de punto flotante en la pantalla, llamando a la función `printf()`. En esta lección aprenderá más acerca de `printf()`, así como de las funciones que son necesarias para recibir la entrada del usuario o imprimir la salida en la pantalla.

Las funciones:

- `getc()`
- `putc()`
- `getchar()`
- `putchar()`

Antes de abordar estas nuevas funciones, sería bueno darnos una idea de la entrada y salida (E/S) estándar de C.

La entrada y salida estándar

Un archivo contiene caracteres y números relacionados. Debido a que en una computadora todos los caracteres y números están representados por bits, y como un byte es una serie de bits, el lenguaje C trata a un archivo como una serie de bytes. A una serie de bytes también se le conoce como *flujo*. De hecho, el lenguaje C trata de la misma manera a todos los flujos de archivos, aunque algunos de ellos provengan de una unidad de disco o cinta, de una terminal o incluso de una impresora.

Además, en C hay flujos de archivos que están preabiertos para usted; esto quiere decir que siempre están disponibles para que los utilice en sus programas:

- `stdin`—La entrada estándar para lectura.
- `stdout`—La salida estándar para escritura.
- `stderr`—El error estándar para escribir mensajes de error.

Por lo regular, el flujo de archivo de entrada estándar (`stdin`) está asociado a su teclado, mientras que los flujos de archivo de salida estándar (`stdout`) y error estándar (`stderr`) están direccionados a la pantalla de su terminal. Además, muchos sistemas operativos permiten al usuario redireccionar estos flujos de archivo.

De hecho, usted ya utilizó `stdout`. Cuando llamó a la función `printf()` en la lección anterior, en realidad envió la salida al flujo de archivo predeterminado `stdout`, el cual está direccionado a su pantalla.

En las secciones siguientes aprenderá más acerca de `stdin` y `stdout`.

El lenguaje C proporciona muchas funciones para manipular la lectura y escritura de archivos (E/S). El archivo de encabezado stdio.h contiene las declaraciones de dichas funciones. Por lo tanto, incluya siempre el archivo de encabezado stdio.h en su programa de C, antes de realizar cualquier cosa con la E/S de archivos.

Obtención de entrada del usuario

En la actualidad, escribir en el teclado es todavía la forma estándar para introducir información en las computadoras. El lenguaje C tiene diversas funciones para indicar a la computadora que lea entradas del usuario (por lo regular a través del teclado). En esta lección se presentan las funciones `getc()` y `getchar()`.

Uso de la función `getc()`

La función `getc()` lee el siguiente carácter de un flujo de archivo y devuelve el carácter como un entero.

Materiał chroniony prawem autorskim

SINTAXIS

La sintaxis de la función `getc()` es

```
#include <stdio.h>
int getc(FILE *flujo);
```

Aquí, *`FILE *flujo`* declara un flujo de archivo (es decir, una variable). La función devuelve el valor numérico del carácter leído. Si ocurre un error o un fin de archivo, la función devuelve `EOF`.

No se preocupe por ahora de la estructura de `FILE`, ya que se presentan más detalles acerca de esto en la hora 21, "Lectura y escritura de archivos", y en la hora 22, "Funciones de archivo especiales". En esta sección se usa el flujo de entrada estándar `stdin` como el archivo de flujo especificado por *`FILE *flujo`*.

EOF es una constante definida en el archivo de encabezado stdio.h. EOF son las siglas de *end-of-file* (fin de archivo). Por lo regular, el valor de EOF es -1. Pero siga usando EOF, en lugar de -1, para indicar el fin de archivo en sus programas. En esa forma, si más adelante utiliza un compilador o un sistema operativo que utilice un valor diferente, su programa seguirá funcionando.

El listado 5.1 muestra un ejemplo que lee un carácter escrito por el usuario desde el teclado, y después lo despliega en la pantalla.

TECLEE **LISTADO 5.1** Lectura de un carácter introducido por el usuario

```
1:  /* 05L01.c: Lectura de datos mediante una llamada a getc() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int ch;
7:
8:     printf("Escriba, por favor, un carácter:\n");
9:     ch = getc( stdin );
10:    printf("El carácter que acaba de introducir es: %c\n", ch);
11:    return 0;
12: }
```

El siguiente es el resultado desplegado en la pantalla de mi computadora después de ejecutar el archivo `05L01.exe`, escribir el carácter `H` y oprimir la tecla Entrar:

SALIDA

```
Escriba, por favor, un carácter:
H
El carácter que acaba de introducir es: H
```

ANÁLISIS Puede ver en la línea 2 del listado 5.1 que se incluye el archivo de encabezado `stdio.h` para las funciones `getc()` y `printf()` que se utilizan en el programa. Las líneas 4 a 12 presentan el nombre y el cuerpo de la función `main()`.

En la línea 6 se declara la variable entera `ch`; más adelante, en la línea 9, se le asigna el valor de retorno de la función `getc()`. La línea 8 imprime un mensaje que solicita al usuario que escriba un carácter desde el teclado. Como mencioné antes en esta lección, la llamada a la función `printf()` de la línea 8 usa la salida estándar predeterminada `stdout` para mostrar mensajes en la pantalla.

En la línea 9 se pasa el flujo de entrada estándar `stdin` a la función `getc()`, lo cual indica que el flujo de archivo proviene del teclado. Después de que el usuario introduce un carácter, la función `getc()` devuelve el valor numérico (esto es, un entero) del carácter. Observe que en la línea 9 se asigna el valor numérico a la variable entera `ch`.

Después, en la línea 10, el carácter que introdujo el usuario se despliega en la pantalla con la ayuda de `printf()`. Observe que dentro de la llamada a la función `printf()`, en la misma línea, se usa el especificador de formato de carácter (`%c`). (En el ejercicio 1 de esta lección se le solicita que utilice `%d` en un programa para imprimir el valor numérico de un carácter introducido por el usuario.)

Uso de la función getchar()

El lenguaje C proporciona otra función, `getchar()`, que realiza una función similar a `getc()`. Para ser más precisos, la función `getchar()` equivale a `getc(stdin)`.

SINTAXIS

La sintaxis de la función `getchar()` es

```
#include <stdio.h>
int getchar(void);
```

Aquí, *void* indica que no es necesario ningún argumento para llamar a la función. La función devuelve el valor numérico del carácter leído. Si ocurre un error o un fin de archivo, la función devuelve `EOF`.

El programa del listado 5.2 muestra cómo usar la función `getchar()` para leer entradas del usuario.

TECLEE **LISTADO 5.2** Lectura de un carácter mediante una llamada a `getchar()`

```
1:  /* 05L02.c: Lectura de datos mediante una llamada a getchar() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int ch1, ch2;
7:
```

Materiał chroniony prawem autorskim

```
8:     printf("Escriba, por favor, dos caracteres juntos:\n");
9:     ch1 = getc( stdin );
10:    ch2 = getchar( );
11:    printf("El primer carácter que acaba de introducir es: %c\n", ch1);
12:    printf("El segundo carácter que acaba de introducir es: %c\n", ch2);
13:    return 0;
14: }
```

Después de ejecutar el archivo 05L02.exe,[e introducir dos caracteres juntos (H e i) sin espacios entre ellos, oprimí la tecla Entrar y obtuve el siguiente resultado desplegado en la pantalla de mi computadora:

SALIDA

```
Escriba, por favor, dos caracteres juntos:
Hi
El primer carácter que acaba de introducir es: H
El segundo carácter que acaba de introducir es: i
```

ANÁLISIS El programa del listado 5.2 es muy similar al del listado 5.1, con excepción de que el primero lee dos caracteres.

La instrucción de la línea 6 declara dos enteros, ch1 y ch2. En la línea 8 se despliega un mensaje que solicita al usuario que escriba dos caracteres juntos.

Luego, en las líneas 9 y 10 se llama a las funciones getc() y getchar(), respectivamente, para leer los dos caracteres introducidos por el usuario. Observe que en la línea 10 no se pasa nada a la función getchar(). Como mencioné antes, esto se debe a que getchar() usa el flujo de archivo de entrada predeterminado, stdin. Usted puede reemplazar la función getchar() de la línea 10 por getc(stdin), ya que esta última equivale a getchar().

Las líneas 11 y 12 envían a la pantalla los dos caracteres (guardados en ch1 y ch2, respectivamente).

Impresión de salida en la pantalla

Además de getc() y getchar() para la lectura, el lenguaje C proporciona también dos funciones para la escritura, putc() y putchar(). En las dos secciones que siguen se presentan estas funciones.

Uso de la función putc()

La función putc() escribe un carácter en el flujo de archivo especificado, el cual, en este caso, es la salida estándar que se direcciona a su pantalla.

SINTAXIS

La sintaxis de la función `putc()` es

```
#include <stdio.h>
int putc(int c, FILE *flujo);
```

Aquí, el primer argumento, *`int c`*, indica que la salida es un carácter almacenado en la variable entera `c`; el segundo argumento, *`FILE *flujo`*, especifica un flujo de archivo. Si tiene éxito, `putc()` devuelve el carácter escrito; en caso contrario, devuelve `EOF`.

En esta lección se especifica la salida estándar `stdout` como el flujo de archivo de salida de `putc()`.

En el listado 5.3 se utiliza la función `putc()` para desplegar el carácter `A` en la pantalla.

TECLEE

LISTADO 5.3 Cómo desplegar un carácter en la pantalla

```
1:  /* 05L03.c: Cómo desplegar un carácter con putc() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int ch;
7:
8:     ch = 65;   /* el valor numérico de A */
9:     printf("El carácter que tiene el valor numérico de 65 es:\n");
10:    putc(ch, stdout);
11:    return 0;
12: }
```

SALIDA

El siguiente resultado es el que obtuve en mi máquina:

```
El carácter que tiene el valor numérico de 65 es:
A
```

Como ya mencioné, en la línea 2 se incluye el archivo de encabezado `stdio.h`, el cual contiene la declaración de `putc()`.

En la línea 8 se le asigna el valor numérico 65 a la variable entera `ch`, declarada en la línea 6. En el conjunto de caracteres ASCII, el valor numérico del carácter `A` es `65`.

En la línea 9 se despliega un mensaje para recordar al usuario que se pondrá en la pantalla el valor numérico del carácter. Después, en la línea 10, la función `putc()`coloca el carácter `A` en la pantalla. Observe que el primer argumento para la función `putc()` es la variable entera (`ch`) que contiene el valor `65`, y que el segundo argumento es el flujo de archivo de salida estándar, `stdout`.

Materiał chroniony prawem autorskim

Otra función para la escritura: putchar()

Al igual que putc(), también se puede usar putchar() para desplegar un carácter en la pantalla. La única diferencia entre las dos funciones es que putchar() sólo necesita un argumento para contener el carácter. No es necesario especificar el flujo de archivo, debido a que la salida estándar (stdout) se asigna como el flujo de archivo para putchar().

SINTAXIS

La sintaxis de la función putchar() es

```
#include <stdio.h>
int putchar(int c);
```

Aquí, *int c* es el argumento que contiene el valor numérico de un carácter. La función devuelve EOF si ocurre un error; en caso contrario, devuelve el carácter que se escribió.

En el listado 5.4 se muestra un ejemplo del uso de putchar().

TECLEE **LISTADO 5.4** Cómo mostrar caracteres con putchar()

```
1:  /* 05L04.c: Cómo mostrar caracteres con putchar() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     putchar(65);
7:        putchar(10);
8:           putchar(66);
9:              putchar(10);
10:          putchar(67);
11:       putchar(10);
12:    return 0;
13: }
```

Después de ejecutar el archivo 05L04.exe, obtuve los siguientes resultados:

SALIDA

```
A
B
C
```

ANÁLISIS La forma de escribir el programa del listado 5.4 es un poco diferente. No hay ninguna variable declarada en el programa. Más bien, los enteros se pasan directamente a putchar(), como se muestra en las líneas 6 a 11.

Como se habrá dado cuenta, 65, 66 y 67 son, respectivamente, los valores numéricos de los caracteres A, B y C en el conjunto de caracteres ASCII. Seguramente se dio cuenta en el ejercicio 5 de la hora 4, "Tipos de datos y palabras reservadas", que 10 es el valor numérico del carácter de nueva línea (\n).

Por lo tanto, las líneas 6 y 7, respectivamente, colocan el carácter `A` en la pantalla y hacen que la computadora comience al principio de la siguiente línea. De la misma manera, la línea 8 despliega el carácter `B` en la pantalla y la línea 9 inicia una nueva línea. Después, la línea 10 despliega el carácter `C` en la pantalla y la línea 11 inicia otra nueva línea. De acuerdo con esto, `A`, `B` y `C`, se colocan al principio de tres líneas consecutivas, como se observó en la sección que muestra la salida del programa.

Nueva visita a la función `printf()`

La función `printf()` es la primera función de la biblioteca de C que utilizó usted en este libro para imprimir mensajes en la pantalla. `printf()` es una función muy importante en C, así que vale la pena dedicarle más tiempo.

SINTAXIS

La sintaxis de la función `printf()` es

```
#include <stdio.h>
int printf(const char *cadena-de-formato, . . .);
```

Aquí, *`const char *cadena-de-formato`* es el primer argumento que contiene el(los) especificador(es) de formato; ... indica la sección de la expresión que contiene la(s) expresión(es) a formatear de acuerdo con los especificadores de formato. El número de especificadores de formato que hay en el primer argumento determina el número de expresiones. Si tiene éxito, la función devuelve el número de expresiones formateadas. Si ocurre un error, devuelve un valor negativo.

`const char *` se explica más adelante en este libro. Por el momento, considere el primer argumento para la función `printf()` como una cadena (una serie de caracteres entre comillas dobles) con algunos especificadores de formato dentro de ella. Por ejemplo, puede pasar como primer argumento a la función `"La suma de dos enteros %d + %d es: %d.\n"`.

La figura 5.1 muestra la relación entre la cadena de formato y las expresiones. Observe que los especificadores de formato y las expresiones coinciden en un orden de izquierda a derecha.

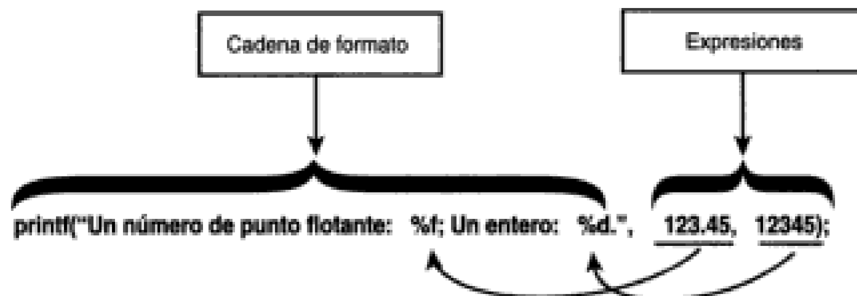


FIGURA 5.1
La relación en `printf()` entre la cadena de formato y las expresiones.

Materiał chroniony prawem autorskim

Recuerde que dentro de la cadena de formato debe usar exactamente el mismo número de expresiones y de especificadores de formato.

Los siguientes son todos los especificadores de formato que se pueden usar en `printf()`:

- `%c` El especificador de formato de carácter.
- `%d` El especificador de formato entero.
- `%i` El especificador de formato entero (equivale a `%d`).
- `%f` El especificador de formato de punto flotante.
- `%e` El especificador de formato de notación científica (observe la e minúscula).
- `%E` El especificador de formato de notación científica (observe la E mayúscula).
- `%g` Utiliza `%f` o `%e`, cualquiera que resulte más corto.
- `%G` Utiliza `%f` o `%E`, cualquiera que resulte más corto.
- `%o` El especificador de formato octal sin signo.
- `%s` El especificador de formato de cadena.
- `%u` El especificador de formato entero sin signo.
- `%x` El especificador de formato hexadecimal sin signo (observe la x minúscula).
- `%X` El especificador de formato hexadecimal sin signo (observe la X mayúscula).
- `%p` Despliega el argumento correspondiente que es un apuntador.
- `%n` Registra el número de caracteres escritos hasta el momento.
- `%%` Muestra un signo de porcentaje (`%`).

Entre los especificadores de formato de esta lista, ya presentamos `%c`, `%d`, `%f`, `%e` y `%E`. Más adelante se explican varios más. La siguiente sección muestra cómo convertir números decimales a hexadecimales utilizando `%x` o `%X`.

Conversión a números hexadecimales

Ya que en una computadora todos los datos son binarios (una serie de ceros y unos), cualquier dato con el que trabajemos o que imprimamos es sólo una clase de representación, legible para los humanos, de los datos binarios. Como programador, a menudo es necesario tratar directamente con datos binarios, pero consume demasiado tiempo descifrar una cadena de ceros y unos para convertirla en datos numéricos o caracteres que sean significativos.

La solución a este problema es la *notación hexadecimal* (o hex), la cual es una especie de forma abreviada para representar números binarios. La notación hexadecimal es un término medio entre el sistema numérico de base 2 (o binario) legible para la computadora, y nuestro más conocido sistema de base 10 (o decimal). Convertir números de hexadecimal a decimal (o de binario a hexadecimal) y a la inversa, es mucho más fácil (y más rápido) que convertirlos directamente de binario a decimal, o viceversa.

La diferencia entre un número hexadecimal y uno decimal es que el primero es un sistema de numeración de base 16. Un número hexadecimal se puede representar mediante cuatro bits. (2^4 es igual a 16, lo que significa que cuatro bits pueden producir 16 números únicos.)

Los números hexadecimales del 0 al 9 emplean los mismos símbolos numéricos que se encuentran en los números decimales del 0 al 9. Las letras `A`, `B`, `C`, `D`, `E` y `F`, en mayúsculas, se usan para representar los números del 10 al 15. (Asimismo, las letras minúsculas `a`, `b`, `c`, `d`, `e` y `f` se emplean para representar estos números hexadecimales. Las mayúsculas y minúsculas hexadecimales son intercambiables y en realidad se trata sólo de una cuestión de estilo.)

El listado 5.5 presenta un ejemplo de la conversión de números decimales a hexadecimales utilizando `%x` o `%X` en la función `printf()`.

TECLEE **LISTADO 5.5** Conversión a números hexadecimales

```
1:  /* 05L05.c: Conversión a números hexadecimales */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     printf("Hex(mayúsculas)   Hex(minúsculas)   Decimal\n");
7:     printf("%X                %x                %d\n", 0, 0, 0);
8:     printf("%X                %x                %d\n", 1, 1, 1);
9:     printf("%X                %x                %d\n", 2, 2, 2);
10:    printf("%X                %x                %d\n", 3, 3, 3);
11:    printf("%X                %x                %d\n", 4, 4, 4);
12:    printf("%X                %x                %d\n", 5, 5, 5);
13:    printf("%X                %x                %d\n", 6, 6, 6);
14:    printf("%X                %x                %d\n", 7, 7, 7);
15:    printf("%X                %x                %d\n", 8, 8, 8);
16:    printf("%X                %x                %d\n", 9, 9, 9);
17:    printf("%X                %x                %d\n", 10, 10, 10);
18:    printf("%X                %x                %d\n", 11, 11, 11);
19:    printf("%X                %x                %d\n", 12, 12, 12);
20:    printf("%X                %x                %d\n", 13, 13, 13);
21:    printf("%X                %x                %d\n", 14, 14, 14);
22:    printf("%X                %x                %d\n", 15, 15, 15);
23:    return 0;
24: }
```

Materiał chroniony prawem autorskim

El siguiente es el resultado que obtuve al ejecutar el archivo `05L05.exe` en mi computadora:

SALIDA

```
Hex(mayúsculas)    Hex(minúsculas)    Decimal
0                  0                  0
1                  1                  1
2                  2                  2
3                  3                  3
4                  4                  4
5                  5                  5
6                  6                  6
7                  7                  7
8                  8                  8
9                  9                  9
A                  a                  10
B                  b                  11
C                  c                  12
D                  d                  13
E                  e                  14
F                  f                  15
```

ANÁLISIS No se alarme si ve demasiadas llamadas a la función `printf()` en el listado 5.5. De hecho, el programa de este listado es muy sencillo. El cuerpo de la función va de las líneas 5 a la 23.

La función `printf()` de la línea 6 imprime un encabezado que contiene tres campos: `Hex(mayúsculas)`, `Hex(minúsculas)` y `Decimal`.

Las líneas 7 a 22 imprimen los números hexadecimales y decimales del 0 al 15. Para realizar la tarea se hacen 16 llamadas a `printf()`. Cada una de las llamadas a `printf()` tiene una cadena de formato como primer argumento seguida de tres enteros que denotan tres expresiones. Observe que dentro de la cadena de formato de cada llamada a `printf()` se usan los especificadores hexadecimales `%X` y `%x`, para convertir las expresiones correspondientes al formato hexadecimal (tanto en mayúsculas como en minúsculas).

En realidad, nadie escribiría un programa como el del listado 5.5. En su lugar se puede utilizar un ciclo para llamar en forma repetida a la función `printf()`. Los ciclos (o iteraciones) se presentan en la hora 7, "Ciclos".

Especificación del ancho mínimo del campo

En un especificador de formato, el lenguaje C le permite agregar un entero entre el signo de porcentaje (`%`) y la letra. El entero se llama *especificador del ancho mínimo del campo*, debido a que especifica dicha característica y asegura que la salida alcance el ancho mínimo. Por ejemplo, en `%10f`, `10` es un especificador del ancho mínimo del campo que asegura que la salida sea de por lo menos 10 caracteres de amplitud. Esto es muy útil al imprimir una columna de números.

El ejemplo del listado 5.6 muestra cómo usar el especificador del ancho mínimo del campo.

TECLEE **LISTADO 5.6** Especificación del ancho mínimo del campo

```
1:  /* 05L06.c: Especificación del ancho mínimo del campo */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int num1, num2;
7:
8:     num1 = 12;
9:     num2 = 12345;
10:    printf("%d\n", num1);
11:    printf("%d\n", num2);
12:    printf("%5d\n", num1);
13:    printf("%05d\n", num1);
14:    printf("%2d\n", num2);
15:    return 0;
16: }
```

SALIDA El siguiente es el resultado que obtuve al ejecutar el archivo `05L06.exe`:

```
12
12345
   12
00012
12345
```

ANÁLISIS En la línea 6 del listado 5.6 se declaran dos variables enteras, `num1` y `num2`, y en las líneas 8 y 9 se les asignan los valores `12` y `12345`, respectivamente.

Sin utilizar ningún especificador del ancho mínimo del campo, las líneas 10 y 11 imprimen los dos enteros llamando a la función `printf()`. Puede ver en la sección de salida que el resultado de la instrucción de la línea 10 es `12`, el cual ocupa el espacio de dos caracteres, mientras que el resultado `12345` de la línea 11 ocupa el espacio de cinco caracteres.

En la línea 12 se especifica un ancho mínimo del campo de `5`, mediante `%5d`. Por lo tanto, la salida de la línea 12 ocupa el espacio de cinco caracteres, ya que son tres espacios en blanco más los dos caracteres del `12`. (Vea la tercera línea de resultados en la sección de salida.)

El especificador `%05d` que está en la función `printf()` que se muestra en la línea 13, indica que el ancho mínimo del campo es `5`, y el `0` indica que se emplean ceros para "rellenar" los espacios. Por lo tanto, puede ver que el resultado de la ejecución de la instrucción de la línea 13 es `00012`.

Materiał chroniony prawem autorskim

El especificador `%2d` de la línea 14 asigna `2` al ancho mínimo del campo, pero usted sigue viendo la salida completa de `12345` en la línea 14. Esto significa que cuando el ancho mínimo del campo es más corto que el ancho de la salida, se toma este último y el resultado se sigue imprimiendo completo.

Alineación de la salida

Como habrá notado en la sección anterior, toda la salida se alinea a la derecha. En otras palabras, toda la salida se coloca en el extremo derecho del campo de manera predeterminada, en tanto que el ancho del campo sea mayor que el de la salida.

Usted puede alinear la salida a la izquierda. Para hacerlo, necesita ponerle un signo de menos (`-`) como prefijo al especificador del ancho mínimo del campo. Por ejemplo, `%-12d` especifica que el ancho mínimo del campo es `12`, pero alinea la salida a partir del extremo izquierdo del campo.

El listado 5.7 proporciona un ejemplo de cómo alinear la salida a la izquierda o a la derecha.

TECLEE **LISTADO 5.7** Salida alineada a izquierda o derecha

```
1:  /* 05L07.c: Alineación de la salida */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int num1, num2, num3, num4, num5;
7:
8:     num1 = 1;
9:     num2 = 12;
10:    num3 = 123;
11:    num4 = 1234;
12:    num5 = 12345;
13:    printf("%8d  %-8d\n", num1, num1);
14:    printf("%8d  %-8d\n", num2, num2);
15:    printf("%8d  %-8d\n", num3, num3);
16:    printf("%8d  %-8d\n", num4, num4);
17:    printf("%8d  %-8d\n", num5, num5);
18:    return 0;
19: }
```

Obtuve los siguientes resultados después de ejecutar el archivo `05L07.exe`:

SALIDA

```
       1  1
      12  12
     123  123
    1234  1234
   12345  12345
```

ANÁLISIS En el listado 5.7 hay cinco variables enteras, `num1`, `num2`, `num3`, `num4` y `num5`, las cuales se declaran en la línea 6 y se les asignan valores en las líneas 8 a 12.

Estos valores representados por las cinco variables se imprimen después por medio de las funciones `printf()` de las líneas 13 a 17. Observe que todas las llamadas a `printf()` tienen el mismo primer argumento: `"%8d %-8d\n"`. Aquí, el primer especificador de formato, `%8d`, alinea la salida al extremo derecho del campo, y el segundo especificador de formato, `%-8d`, alinea la salida al extremo izquierdo del campo.

Después de ejecutar las instrucciones de las líneas 13 a 17, se lleva a cabo la alineación y se ponen los resultados en la pantalla como sigue:

```
       1 1
      12 12
     123 123
    1234 1234
   12345 12345
```

Uso del especificador de precisión

Puede colocar un punto (`.`) y un entero justo después del especificador del ancho mínimo del campo. La combinación del punto y el entero conforma un *especificador de precisión*. Puede usar el especificador de precisión para determinar el número de posiciones decimales para los números de punto flotante, o para especificar el ancho máximo (o longitud) del campo para enteros o cadenas. (Las cadenas de C se presentan en la hora 13, "Cadenas".)

Por ejemplo, con `%10.3f`, la longitud del ancho mínimo del campo se especifica de 10 caracteres, y al número de posiciones decimales se le asigna el 3. (Recuerde, el número predeterminado de posiciones decimales es de 6.) Para los enteros, `%3.8d` indica que el ancho mínimo del campo es de `3`, y el máximo es de `8`.

El listado 5.8 presenta un ejemplo del uso de los especificadores de precisión.

TECLEE **LISTADO 5.8** Uso de los especificadores de precisión

```
1:  /* 05L08.c: Uso de los especificadores de precisión */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int int_num;
7:     double flt_num;
8:
9:     int_num = 123;
10:    flt_num = 123.456789;
11:    printf("Formato de entero predeterminado:    %d\n", int_num);
```

Materiał chroniony prawem autorskim

```
12:    printf("Con especificador de precisión:          %2.8d\n", int_num);
13:    printf("Formato de punto flotante predeterminado: %f\n", flt_num);
14:    printf("Con especificador de precisión:          %-10.2f\n", flt_num);
15:    return 0;
16: }
```

Después de ejecutar en mi computadora el archivo `05L08.exe`, obtuve en la pantalla los siguientes resultados:

SALIDA

```
Formato de entero predeterminado:         123
Con especificador de precisión:           00000123
Formato de punto flotante predeterminado: 123.456789
Con especificador de precisión:           123.46
```

ANÁLISIS El programa del listado 5.8 declara una variable entera, `int_num`, en la línea 6, y en la línea 7 un número de punto flotante, `flt_num`. Las líneas 9 y 10 asignan `123` y `123.456789` a `int_num` y a `flt_num`, respectivamente.

En la línea 11 se especifica el formato de entero predeterminado para la variable entera `int_num`, mientras que la instrucción de la línea 12 indica el formato de entero con un especificador de precisión que determina que el ancho máximo del campo es de ocho caracteres. Por lo tanto, puede ver que se incluyeron cinco ceros antes del entero `123` de la segunda línea de los resultados.

Para la variable de punto flotante, `flt_num`, la línea 13 imprime un valor de punto flotante en el formato predeterminado, y la línea 14 reduce a dos las posiciones decimales poniendo el especificador de precisión `.2` dentro del especificador de formato `%-10.2f`. Observe que también se especifica la alineación a la izquierda por medio del signo de menos (`-`) en el especificador de formato de punto flotante.

La instrucción de la línea 14 produce el número de punto flotante `123.46` de la cuarta línea de los resultados, por medio del especificador de precisión para dos posiciones decimales. Por lo tanto, `123.456789` redondeado a dos posiciones decimales se convierte en `123.46`.

Resumen

En esta lección aprendió los siguientes conceptos, especificadores y funciones importantes:

- El lenguaje C trata a un archivo como una serie de bytes.
- `stdin`, `stdout` y `stderr` son tres flujos de archivo que son pre-abiertos y siempre están disponibles para que usted los utilice.
- Se pueden utilizar las funciones `getc()` y `getchar()` de la biblioteca de C para leer un carácter de la entrada estándar.
- Se pueden utilizar las funciones `putc()` y `putchar()` de la biblioteca de C para imprimir un carácter en la salida estándar.

- Se puede emplear `%x` o `%X` para convertir números decimales en hexadecimales.
- El ancho mínimo del campo se puede especificar y asegurar agregando un entero en el especificador de formato.
- Es posible alinear una salida ya sea al extremo izquierdo o al derecho del campo de salida.
- Se puede utilizar un especificador de precisión para especificar el número de posiciones decimales de los números de punto flotante, o el ancho máximo del campo para enteros o cadenas.

En la siguiente lección aprenderá acerca de algunos operadores importantes de C.

Preguntas y respuestas

P ¿Qué son `stdin`, `stdout` y `stderr`?

R En C, un archivo es tratado como una serie de bytes, a la cual se le llama flujo de archivo. `stdin`, `stdout` y `stderr` son flujos de archivo pre-abiertos. `stdin` es la entrada estándar de lectura; `stdout` es la salida estándar de escritura; `stderr` es el error estándar para escribir mensajes de error.

P ¿Qué valor decimal equivale al número hexadecimal 32?

R El hexadecimal, o hex para abreviar, es un sistema numérico de base 16. Por lo tanto, 32 (hex) es igual a $3*16^1+2*16^0$, o 50 en decimal.

P ¿Son equivalentes `getc(stdin)` y `getchar()`?

R Debido a que la función `getchar()` lee de manera predeterminada del flujo de archivo `stdin`, `getc(stdin)` y `getchar()` sí son equivalentes.

P En la función `printf("El entero %d es lo mismo que el hexadecimal %x", 12, 12)`, ¿cuál es la relación entre los especificadores de formato y las expresiones?

R Los dos especificadores de formato, `%d` y `%x`, especifican los formatos de valores numéricos contenidos en la sección de expresiones. Así, el primer valor numérico de `12` se imprimirá en un formato entero, mientras que el segundo `12` (el de la sección de expresiones) se mostrará en el formato hexadecimal. Hablando en forma general, el número de especificadores de formato de la sección de formato debe coincidir con el número de expresiones de la correspondiente sección de expresiones.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección.

Materiał chroniony prawem autorskim

Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Es posible alinear los resultados al extremo izquierdo del campo de salida en lugar de al derecho?
2. ¿Cuál es la diferencia entre `putc()` y `putchar()`?
3. ¿Qué valor devuelve `getchar()`?
4. Dentro de `%10.3f`, ¿qué parte es el especificador del ancho mínimo del campo, y qué parte es el especificador de precisión?

Ejercicios

1. Escriba un programa para poner juntos en la pantalla los caracteres `B`, `y` y `e`.
2. Despliegue los números `123` y `123.456` alineados al extremo izquierdo del campo.
3. Dados los tres enteros `15`, `150` y `1500`, escriba un programa que los imprima en la pantalla en formato hexadecimal.
4. Escriba un programa que use `getchar()` y `putchar()` para leer un carácter introducido por el usuario y escríbalo en la pantalla.
5. Si compila el siguiente programa de C, ¿qué mensajes de advertencia o error obtendrá?

```
main()
{
   int ch;
   ch = getchar();
   putchar(ch);
   return 0;
}
```

5

Parte II

Operadores e instrucciones de control de flujo

Hora

6 Manejo de datos

7 Ciclos

8 Uso de operadores condicionales

9 Modificadores de datos y funciones matemáticas

10 Control de flujo del programa

Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

Hora 6

Manejo de datos

"La cuestión es", dijo Humpty Dumpty,
"saber quién es el que manda; eso es todo."

—L. Carroll

Puede imaginar a los operadores de C como verbos que le permiten manipular datos (los cuales son como sustantivos). De hecho, en la hora 3, "La estructura de un programa de C", aprendió algunos operadores como + (suma), - (resta), * (multiplicación), / (división) y % (residuo). El lenguaje C tiene un conjunto completo de operadores. En esta hora aprenderá acerca de otros operadores, como son:

- Los operadores de asignación aritmética
- El operador de negación
- Los operadores de incremento y decremento
- Los operadores relacionales
- Los operadores de conversión

Los operadores de asignación aritmética

Antes de abordar los operadores de asignación aritmética, veamos primero más de cerca al propio operador de asignación.

El operador de asignación (=)

En el lenguaje C, al operador = se le llama *operador de asignación*, mismo que usted ha visto y utilizado en varias horas.

La forma general de la instrucción para usar un operador de asignación es:

```
operando_de_la_izquierda = operando_de_la_derecha;
```

Aquí, la instrucción hace que el valor del *operando_de_la_derecha* se asigne (o se escriba) en la localidad de memoria del *operando_de_la_izquierda*. De esta manera, después de la asignación, el valor del *operando_de_la_izquierda* será igual al del *operando_de_la_derecha*. Adicionalmente, toda la expresión de asignación se evalúa al mismo valor que se asigna al *operando_de_la_izquierda*.

Por ejemplo, la instrucción `a = 5;` escribe el valor del operando de la derecha (`5`) en la localidad de memoria de la variable entera `a` (la cual, en este caso, es el operando de la izquierda).

De la misma manera, la instrucción `b = a = 5;` asigna primero el valor `5` a la variable entera `a`, y luego a la variable entera `b`. Después de la ejecución de la instrucción, tanto `a` como `b` contienen el valor de `5`.

Es importante recordar que el operando de la izquierda del operador de asignación debe ser una expresión en la cual pueda escribir datos en forma legal. Una expresión como `6 = a`, aunque a primera vista podría parecer correcta, en realidad está al revés y no funcionará. El operador de asignación = siempre funciona de derecha a izquierda; por lo tanto, el valor del lado izquierdo debe ser alguna variable que pueda recibir datos desde la expresión de la derecha.

No confunda el operador de asignación (=) con el operador relacional, == (llamado *operador de igualdad*). El operador == se presenta más adelante en esta hora.

Combinación de operadores aritméticos con =

Considere este ejemplo: ¿Cómo asignaría la suma de las variables enteras `x` y `y` a la variable entera `z`?

Materiał chroniony prawem autorskim

Utilizando el operador de asignación (=) y el operador de suma (+), puede obtener la siguiente instrucción:

```
z = x + y;
```

Como puede observar, es muy sencillo. Ahora bien, considere el mismo ejemplo, pero en lugar de asignar el resultado a la tercera variable, z, escribiremos el resultado de la suma en la variable entera x:

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

Recuerde, el operador = siempre funciona de derecha a izquierda, de modo que primero se evaluará la parte derecha. Aquí, en la parte derecha del operador de asignación (=), se realiza la suma de x y y; en la parte izquierda del operador se reemplaza el valor previo de x con el resultado de la suma de la parte derecha.

El lenguaje C le proporciona un nuevo operador, +=, para hacer la suma y la asignación juntas. Por lo tanto, puede reescribir la instrucción `x = x + y;` como:

```
x += y;
```

La combinación del operador de asignación (=) con los operadores aritméticos, +, -, *, / y %, le ofrece otro tipo de operadores, los *operadores de asignación aritmética*:

Operador	*Descripción*
+=	Operador de asignación de suma
-=	Operador de asignación de resta
*=	Operador de asignación de multiplicación
/=	Operador de asignación de división
%=	Operador de asignación de residuo

A continuación se muestran instrucciones equivalentes:

`x += y;` equivale a `x = x + y;`

`x -= y;` equivale a `x = x - y;`

`x *= y;` equivale a `x = x * y;`

`x /= y;` equivale a `x = x / y;`

`x %= y;` equivale a `x = x % y;`

Observe que la instrucción

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

no equivale a la instrucción

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

debido a que

```
z *= x + y
```

multiplica z por toda la parte derecha de la instrucción; por lo tanto, el resultado sería

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

El listado 6.1 presenta un ejemplo del uso de algunos operadores de asignación aritmética.

TECLEE **LISTADO 6.1** Uso de los operadores de asignación aritmética

```
1:  /* 06L01.c: Uso de los operadores de asignación aritmética */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y, z;
7:
8:     x = 1;   /* inicializa x */
9:     y = 3;   /* inicializa y */
10:    z = 10;  /* inicializa z */
11:    printf("Dadas x = %d, y = %d, y z = %d,\n", x, y, z);
12:
13:    x = x + y;
14:    printf("x = x + y  asigna %d a x;\n", x);
15:
16:    x = 1;  /* restablece x */
17:    x += y;
18:    printf("x += y  asigna %d a x;\n", x);
19:
20:    x = 1;  /* restablece x */
21:    z = z * x + y;
22:    printf("z = z * x + y  asigna %d a z;\n", z);
23:
24:    z = 10;  /* restablece z */
25:    z = z * (x + y);
26:    printf("z = z * (x + y) asigna %d a z;\n", z);
27:
28:    z = 10;  /* restablece z */
29:    z *= x + y;
30:    printf("z *= x + y asigna %d a z.\n", z);
31:
32:    return 0;
33: }
```

Materiał chroniony prawem autorskim

Después de compilar y enlazar este programa se crea un archivo ejecutable. En mi máquina este archivo ejecutable se denomina `06L01.exe`. Los resultados obtenidos después de ejecutar dicho archivo son:

SALIDA

```
Dadas x = 1, y = 3, y z = 10,
x = x + y  asigna 4 a x;
x += y  asigna 4 a x;
z = z * x + y  asigna 13 a z;
z = z * (x + y) asigna 40 a z;
z *= x + y asigna 40 a z.
```

ANÁLISIS La línea 2 del listado 6.1 incluye el archivo de encabezado `stdio.h` mediante la directiva `include`. Este archivo de encabezado se requiere para las funciones `printf()` que se emplean en las líneas 4 a 33.

Las líneas 8 a 10 inicializan tres variables enteras, `x`, `y` y `z`, las cuales están declaradas en la línea 6. Después, la línea 11 imprime los valores iniciales asignados a `x`, `y` y `z`.

La instrucción de la línea 13 utiliza un operador de suma y un operador de asignación para sumar los valores contenidos en `x` y `y`, y luego asigna el resultado a `x`. La línea 14 despliega el resultado en la pantalla.

Las líneas 17 y 18 realizan la misma suma y despliegan de nuevo el resultado, después de restablecer el valor de `x` a 1 en la línea 16. En esta ocasión se utiliza el operador de asignación aritmética `+=`.

El valor de `x` se restablece en la línea 20. La línea 21 realiza una multiplicación y una suma y guarda el resultado en la variable entera `z`; esto es, `z = z * x + y;`. La llamada a `printf()` de la línea 22 despliega como resultado el `13` en la pantalla. Una vez más, la instrucción `x = 1;` de la línea 20 restablece el valor de la variable entera `x`.

Las líneas 24 a 30 muestran los resultados de dos cálculos. De hecho, ambos resultados son iguales (es decir, `40`) debido a que los dos cálculos de las líneas 25 y 29 son equivalentes. La única diferencia entre las dos instrucciones de dichas líneas es que en la línea 29 se utiliza el operador de asignación aritmética `*=`.

Obtención de negaciones de valores numéricos

Si desea cambiar el signo de un número, puede colocar el operador menos (`-`) justo antes del número. Por ejemplo, puede obtener el valor negativo del entero `7` cambiando su signo de esta manera: `-7`. Aquí, `-` es el operador menos.

Al símbolo - utilizado de esta forma se le llama operador de *negación o unario menos*. Esto se debe a que el operador toma sólo un operando: la expresión inmediata a su derecha. El tipo de datos del operando puede ser cualquier número entero o de punto flotante.

También puede aplicar el operador de negación a una variable entera o a una de punto flotante. Por ejemplo, dada `x = 1.234`, `-x` es igual a `-1.234`. O dada `x = -1.234`, `-x` equivale a `1.234`, ya que la negación de un valor negativo da como resultado un número positivo.

No confunda el operador de negación, o unario menos, con el operador de resta, aunque ambos utilicen el mismo símbolo. Por ejemplo, la siguiente instrucción:

```
z = x - -y;
```

es en realidad la misma que ésta:

```
z = x - (-y);
```

o que ésta:

```
z = x + y;
```

Aquí, en ambas instrucciones, el primer símbolo - se usa como el operador de resta, mientras que el segundo es un operador de negación.

Incremento o decremento en una unidad

Los operadores de incremento y decremento son muy prácticos cuando usted desea sumar o restar 1 a una variable. El símbolo del operador de incremento es `++`. El operador de decremento es `--`.

Por ejemplo, puede reescribir la instrucción `x = x + 1;` como `++x;`, o reemplazar `x = x - 1;` con `--x;`.

En realidad existen dos versiones de los operadores de incremento y decremento. En la instrucción `++x;`, en donde `++` aparece antes de su operando, el operador de incremento se denomina *operador de preincremento*. Esto se refiere al orden en que suceden las cosas: el operador primero suma 1 a `x`, y después produce el nuevo valor de `x`. Asimismo, en la instrucción `--x;`, el *operador de predecremento* primero resta 1 a `x` y luego produce el nuevo valor de `x`.

Si tiene una expresión como `x++`, en la que `++` aparece después de su operando, entonces está empleando un *operador de posincremento*. Del mismo modo, en `x--`, al operador de decremento se le llama *operador de posdecremento*.

Por ejemplo, en la instrucción `y = x++;`, primero se le asigna a `y` el valor original de `x`, y después `x` se incrementa en 1.

El operador de posdecremento tiene una historia parecida. En la instrucción `y = x--;` primero se asigna el valor de `x` a `y`, y después se decrementa `x`. El programa del listado 6.2 muestra las diferencias entre las dos versiones de los operadores de incremento y decremento.

Materiał chroniony prawem autorskim

TECLEE **LISTADO 6.2** Uso de los operadores de pre y posincremento y decremento.

```
1:  /* 06L02.c: operadores de pre o posincremento/decremento*/
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int w, x, y, z, resultado;
7:
8:     w = x = y = z = 1;    /* inicializa x y y */
9:     printf("Dadas w = %d, x = %d, y = %d, y z = %d,\n", w, x, y, z);
10:
11:    resultado = ++w;
12:    printf("El resultado de la evaluación de ++w es %d y ahora w es %d\n",
       resultado, w);
13:    resultado = x++;
14:    printf("El resultado de la evaluación de x++ es %d y ahora x es %d\n",
       resultado, x);
15:    resultado = --y;
16:    printf("El resultado de la evaluación de --y es %d y ahora y es %d\n",
       resultado, y);
17:    resultado = z--;
18:    printf("El resultado de la evaluación de z-- es %d y ahora z es %d\n",
       resultado, z);
19:    return 0;
20: }
```

Se obtuvo el siguiente resultado al ejecutar el archivo `06L02.exe`:

SALIDA

```
Dadas w = 1, x = 1, y = 1, z = 1,
El resultado de la evaluación de ++w es 2 y ahora w es 2
El resultado de la evaluación de x++ es 1 y ahora x es 2
El resultado de la evaluación de --y es 0 y ahora y es 0
El resultado de la evaluación de z-- es 1 y ahora z es 0
```

ANÁLISIS Dentro de la función `main()`, la línea 8 del listado 6.2 asigna `1` a cada una de las variables enteras, `w`, `x`, `y`, y `z`. La llamada a `printf()` de la línea 9 despliega los valores contenidos en las cuatro variables enteras.

Después se ejecuta la instrucción de la línea 11 y el resultado del preincremento de `w` se asigna a la variable entera `resultado`. En la línea 12 se imprime en la pantalla el valor de `resultado`, el cual es `2`, junto con el valor de `w` después de la instrucción de preincremento. Observe que `w` sigue siendo `2` ya que el incremento tuvo lugar antes de asignar el nuevo valor de `w` a `resultado`.

Las líneas 13 y 14 obtienen el posincremento de `x` e imprimen el resultado. Como usted sabe, el resultado se obtiene antes de incrementar el valor de `x`. Por lo tanto, usted ve el valor de `1` (el valor anterior de `x`) en el resultado de `x++`, mientras que el nuevo valor de `x` es `2`, ya que se incrementó después de la asignación a `resultado` de la línea 13. El operador de predecremento de la línea 15 provoca que se reduzca en 1 el valor de `y` antes

de asignar el nuevo valor a la variable entera `resultado`. Por lo tanto, usted ve `0` en la pantalla como el resultado de `--y`, el cual refleja el nuevo valor de `y`, que es también `0`.

Sin embargo, el operador de posdecremento de la línea 17 no tiene efecto sobre la asignación, ya que el valor original de `z` se da a la variable entera `resultado` antes de que `z` disminuya en 1. El posdecremento actúa como si la línea 17 fuera simplemente `resultado = z`, disminuyendo `z` en 1 después de ejecutar la instrucción. La línea 18 imprime entonces el resultado `1`, el cual es por supuesto el valor original de `z`, junto con `0`, que es el valor de `z` después del posdecremento.

Mayor que o menor que

Hay seis tipos de relaciones entre dos expresiones: igual a, diferente de, mayor que, menor que, mayor o igual a, y menor o igual a. De acuerdo con esto, el lenguaje C proporciona estos seis *operadores relacionales*:

Operador	*Descripción*
`==`	Igual a
`!=`	Diferente de
`>`	Mayor que
`<`	Menor que
`>=`	Mayor o igual a
`<=`	Menor o igual a

Todos los operadores relacionales tienen una precedencia menor que los operadores aritméticos. Por lo tanto, todas las operaciones aritméticas a ambos lados de un operador relacional se llevan a cabo antes de hacer cualquier comparación. Debe colocar paréntesis para encerrar las operaciones de operadores que se deban realizar primero.

Entre los seis operadores relacionales, `>`, `<`, `>=` y `<=` tienen una mayor precedencia que los operadores `==` y `!=`.

Por ejemplo, la expresión

```
x * y < z + 3
```

se interpreta como

```
(x * y) < (z + 3)
```

Otro punto importante es que todas las expresiones relacionales producen un resultado de `0` o `1`. En otras palabras, una expresión relacional se evalúa como `1` si se cumple la relación especificada. En caso contrario, produce `0`.

Por ejemplo, dadas `x = 3` y `y = 5`, la expresión relacional `x < y` da un resultado de `1`.

El listado 6.3 muestra más ejemplos del uso de operadores relacionales.

Material chroniony prawem autorskim

Cuando en una expresión aparecen juntos varios operadores relacionales diferentes, los operandos que están en medio se asocian con los operadores en un orden determinado. La *precedencia de los operadores* se refiere al orden en que se agrupan los operadores y los operandos. Primero se agrupan con sus operandos los operadores que tienen la mayor precedencia dentro de la expresión.

Por ejemplo, en la expresión

```
z + x * y - 3
```

el operador * tiene mayor precedencia que los operadores + y -. Por lo tanto, `x * y` se evaluará primero, y su resultado se convertirá en el operando de la derecha del operador +. Ese resultado se da entonces al operador – como su operando de la izquierda.

Si desea anular la precedencia predeterminada de los operadores, puede utilizar paréntesis para agrupar los operandos dentro de una expresión. Por ejemplo, si lo que en realidad quería era multiplicar z + x por y - 3, entonces podría reescribir la expresión anterior como

```
(z + x) * (y - 3)
```

Además, siempre puede usar los paréntesis cuando no esté seguro de los efectos de la precedencia de los operadores, o simplemente cuando desee que su código sea más fácil de leer.

TECLEE **LISTADO 6.3** Resultados producidos por expresiones relacionales

```
1:  /* 06L03.c: Uso de operadores relacionales */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y;
7:     double z;
8:
9:     x = 7;
10:    y = 25;
11:    z = 24.46;
12:    printf("Dadas x = %d, y = %d, y z = %.2f,\n", x, y, z);
13:    printf("x >= y  produce: %d\n", x >= y);
14:    printf("x == y  produce: %d\n", x == y);
15:    printf("x < z   produce: %d\n", x < z);
16:    printf("y > z   produce: %d\n", y > z);
17:    printf("x != y - 18  produce: %d\n", x != y - 18);
18:    printf("x + y != z   produce: %d\n", x + y != z);
19:    return 0;
20: }
```

Después de ejecutar el archivo `06L03.exe`, se desplegaron los siguientes resultados en la pantalla de mi computadora:

SALIDA

```
Dadas x = 7, y = 25, y z = 24.46,
x >= y  produce: 0
x == y  produce: 0
x < z   produce: 1
y > z   produce: 1
x != y - 18  produce: 0
x + y != z   produce: 1
```

ANÁLISIS Hay dos variables enteras, `x` y `y`, y una variable de punto flotante, `z`, declaradas en las líneas 6 y 7, respectivamente.

En las líneas 9 a 11 se inicializan las tres variables. La línea 12 imprime los valores asignados a las variables.

Debido a que el valor de `x` es `7` y el valor de `y` es `25`, `y` es mayor que `x`. Por lo tanto, la línea 13 imprime `0`, que es el resultado que produce la expresión relacional `x >= y`.

Asimismo, la expresión relacional `x == y` de la línea 14 produce `0`.

Las líneas 15 y 16 imprimen el resultado de `1`, mismo que producen las dos expresiones `x < z` y `y > z`.

La instrucción de la línea 17 muestra `0`, que es el resultado de la expresión relacional `x != y - 18`. Ya que `y - 18` produce `7`, y el valor de `x` es `7`, la relación `!=` no se cumple. La expresión `x + y != z` de la línea 18 produce `1`, mismo que se muestra en la pantalla.

Tenga cuidado al comparar dos valores con una expresión de igualdad. Debido al truncamiento o al redondeo, algunas expresiones relacionales, que son verdaderas algebraicamente, podrían producir 0 en vez de 1. Por ejemplo, observe la siguiente expresión relacional:

```
1 / 2 + 1 / 2 == 1
```

Esto es algebraicamente correcto y uno esperaría que se evaluara como `1`.

Sin embargo, la expresión produce `0`, lo que significa que no se cumple la relación de igualdad. Esto se debe a que el truncamiento de la división entera, es decir, `1 / 2`, produce un entero: `0`, no `0.5`.

Otro ejemplo es `1.0 / 3.0`, el cual produce `0.33333...`. Éste es un número con una cantidad infinita de posiciones decimales. Pero la computadora sólo puede contener un número limitado de posiciones decimales. Por lo tanto, la expresión

```
1.0 / 3.0 + 1.0 / 3.0 + 1.0 / 3.0 == 1.0
```

podría no producir `1` en algunas computadoras, aunque la expresión sea algebraicamente correcta.

Materiał chroniony prawem autorskim

Uso del operador de conversión explícita

En C, usted puede convertir el tipo de datos de una variable, expresión o constante a otro diferente utilizando como prefijo el operador de conversión explícita. Esta conversión no cambia el operando en sí; cuando se evalúa el operador de conversión, produce el mismo valor (pero representado como un tipo diferente), el cual se puede utilizar en el resto de la expresión.

La forma general del operador de conversión explícita es

```
(tipo_de_datos) x
```

Aquí, `tipo de datos` especifica el nuevo tipo de datos que desea. x es una variable (o constante o expresión) de un tipo de datos diferente. Para formar un operador de conversión explícita se deben incluir los paréntesis `(` y `)` alrededor del nuevo tipo de datos.

Por ejemplo, la expresión `(float)5` convierte el entero `5` a un número de punto flotante, `5.0`.

El programa del listado 6.4 muestra otro ejemplo del uso del operador de conversión explícita.

TECLEE **LISTADO 6.4** Jugando con el operador de conversión explícita

```
1:  /* 06L04.c: Uso del operador de conversión explícita */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y;
7:
8:     x = 7;
9:     y = 5;
10:    printf("Dadas x = %d, y = %d\n", x, y);
11:    printf("x / y produce: %d\n",  x / y);
12:    printf("(float)x / y produce: %f\n",  (float)x / y);
13:    return 0;
14: }
```

Obtuve los siguientes resultados al ejecutar el programa `06L04.exe` en mi computadora:

SALIDA

```
Dadas x = 7, y = 5
x / y produce: 1
(float)x / y produce: 1.400000
```

ANÁLISIS En el listado 6.4 hay dos variables enteras, x y y, declaradas en la línea 6, e inicializadas en las líneas 8 y 9, respectivamente. La línea 10 despliega los valores contenidos en las variables enteras x y y.

La instrucción de la línea 11 imprime la división entera de `x / y`. El resultado de dicha división es `1`, debido a que se trunca la parte de la fracción.

Sin embargo, el operador de conversión `(float)` de la línea 12 convierte el valor de `x` a un valor de punto flotante. Por lo tanto, la expresión `(float)x / y` se convierte en una división de punto flotante y devuelve un número de punto flotante. Es por ello que usted ve que se muestra en la pantalla el número de punto flotante `1.400000`, después de ejecutar la instrucción de la línea 12.

Resumen

En esta lección aprendió acerca de los siguientes operadores importantes:

- El operador de asignación `=`, el cual tiene dos operandos (uno a cada lado). El valor del operando de la derecha se asigna al operando de la izquierda. El operando de la izquierda debe ser un tipo de variable que pueda aceptar el nuevo valor.
- Los operadores de asignación aritmética `+=`, `-=`, `*=`, `/=` y `%=` son combinaciones de los operadores aritméticos con el operador de asignación.
- El operador de negación o unario menos (`-`), el cual obtiene la negación de un valor numérico.
- Las dos versiones del operador de incremento, `++`. Usted sabe que en `++x`, el operador `++` se denomina operador de preincremento, y en `x++`, `++` es el operador de posincremento.
- Las dos versiones del operador de decremento, `--`. Usted aprendió que, por ejemplo, en `--x`, el operador `--` es el operador de predecremento, y en `x--`, `--`es el operador de posdecremento.
- Los seis operadores relacionales de C: `==` (igual a), `!=` (diferente de), `>` (mayor que), `<` (menor que), `>=` (mayor o igual a) y `<=` (menor o igual a).
- Cómo cambiar el tipo de datos de una expresión anteponiendo un operador de conversión explícita a los datos.

En la siguiente lección aprenderá acerca de los ciclos en el lenguaje C.

Preguntas y respuestas

P ¿Cuál es la diferencia entre el operador de preincremento y el operador de posincremento?

R El operador de preincremento primero incrementa en 1 el valor del operando, y luego produce el valor modificado. Por otra parte, el operador de posincremento primero produce el valor original de su operando y después incrementa el

Materiał chroniony prawem autorskim

operando. Por ejemplo, dada la expresión `x = 1`, la expresión `++x` produce `2`, mientras que la expresión `x++` se evalúa como `1` antes de modificar a `x`.

P ¿Es lo mismo el operador de negación, o unario menos, (-) que el operador de resta (-)?

R No, aunque ambos operadores comparten el mismo símbolo, no son lo mismo. El significado del símbolo lo determina el contexto en el que aparece. El operador de negación se usa para cambiar el signo de un valor numérico. En otras palabras, el operador de negación produce la negación del valor. El operador de resta es un operador aritmético que realiza una resta entre dos operandos.

P ¿Qué operador tiene una mayor precedencia, un relacional o un aritmético?

R Un operador aritmético tiene mayor precedencia que un operador relacional. Por ejemplo, en la expresión `x * y + z > x + y`, la precedencia de operadores de mayor a menor va de `*` a `+` y por último a `>`. La expresión completa se interpreta entonces como `((x * y) + z) > (x + y)`.

P ¿Qué valor produce una expresión relacional?

R Una expresión relacional se evalúa como `0` o `1`. Si en una expresión la relación que indica un operador relacional es verdadera, la expresión se evalúa como `1`; en caso contrario se evalúa como `0`.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Cuál es la diferencia entre el operador `=` y el operador `==`?
2. En la expresión `x + - y - - z`, ¿cuáles operadores son de resta y cuáles son de negación?
3. Dadas `x = 15` y `y = 4`, ¿qué valores producen, respectivamente, las expresiones `x / y` y `(float)x / y`?
4. ¿Es la expresión `y *= x + 5` equivalente a la expresión `y = y * x + 5`?

Ejercicios

1. Dadas `x = 1` y `y = 3`, escriba un programa que imprima los resultados de estas expresiones: `x += y`, `x += -y`, `x -= y`, `x -= -y`, `x *= y`, `x *= -y`.

2. Dadas `x = 3` y `y = 6`, ¿cuál es el valor de `z` después de ejecutar la instrucción `z = x * y == 18;`?

3. Escriba un programa que inicialice la variable entera `x` con `1` y muestre resultados con las siguientes instrucciones:

```
printf("x++ produce:   %d\n", x++);
printf("Ahora x contiene: %d\n", x);
```

4. Reescriba el programa del ejercicio 3. Esta vez incluya las siguientes instrucciones:

```
printf("x = x++ produce: %d\n", x = x++);
printf("Ahora x contiene:   %d\n", x);
```

   ¿Qué obtiene después de ejecutar el programa? ¿Puede explicar el porqué de dicho resultado?

5. Se supone que el siguiente programa debe comparar las variables `x` y `y` en cuanto a igualdad. ¿Qué está mal en el programa? (Sugerencia: ejecute el programa para ver qué imprime.)

```
#include <stdio.h>

main()
{
   int x, y;

   x = y = 0;
   printf("El resultado de la comparación es: %d\n",  x = y);
   return 0;
}
```

Materiał chroniony prawem autorskim

HORA 7

Ciclos

Cielo y tierra:
Cánticos sutra no escuchados
Repetidos…
—Proverbio Zen

En las lecciones anteriores aprendió los aspectos básicos de un programa en C, diversas funciones de C importantes, la E/S estándar y algunos operadores útiles. En esta lección aprenderá una característica muy importante del lenguaje C: los ciclos. Los ciclos, también llamados *iteraciones*, se usan en la programación para realizar una y otra vez el mismo conjunto de instrucciones hasta satisfacer una determinada condición.

En C hay tres instrucciones diseñadas para los ciclos:

- La instrucción `while`
- La instrucción `do-while`
- La instrucción `for`

En las secciones que siguen veremos estas instrucciones.

El ciclo while

La finalidad de la palabra reservada `while` es ejecutar una instrucción una y otra vez mientras que una condición dada sea cierta. Cuando la condición del ciclo `while` ya no es lógicamente verdadera, el ciclo termina y la ejecución del programa continúa en la siguiente instrucción después del ciclo.

La forma general de la instrucción `while` es

```
while (expresión)
   instrucción;
```

Aquí, *`expresión`* es la condición de la instrucción `while`. Esta expresión se evalúa primero. Si la expresión se evalúa como un valor *diferente de cero*, entonces se ejecuta la *`instrucción`*. Después de esto se evalúa una vez más la *`expresión`*. Entonces, si la expresión se sigue evaluando como un valor diferente de cero, la instrucción se ejecuta una vez más. Este proceso se repite una y otra vez hasta que la *`expresión`* se evalúe como *`cero`*, o falso lógico.

La idea es que el código del ciclo (la *`instrucción`*`;`) haga que la *`expresión`* sea lógicamente falsa la siguiente ocasión que se evalúe, terminando así el ciclo.

Por supuesto, a menudo querrá utilizar la palabra reservada `while`, en lugar de otras instrucciones, para controlar un ciclo. Cuando éste sea el caso, utilice un bloque de instrucciones encerrado con llaves { y }. Cada vez que se evalúe la expresión `while`, todo el bloque se ejecutará si la expresión se evalúa como verdadera.

Veamos ahora un ejemplo del uso de la instrucción `while`. El programa del listado 7.1 utiliza un ciclo `while` para leer continuamente y desplegar después el carácter introducido, *siempre y cuando* éste sea diferente de `'x'`.

TECLEE **LISTADO 7.1** Uso de un ciclo `while`

```
1:  /* 07L01.c: Uso de un ciclo while */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int c;
7:
8:     c = ' ';
9:     printf("Introduzca un carácter:\n(introduzca x para terminar)\n");
10:    while (c != 'x') {
11:       c = getc(stdin);
12:       putchar(c);
13:    }
14:    printf("\nEl ciclo while ha terminado. ¡Hasta pronto! \n");
15:    return 0;
16: }
```

Materiał chroniony prawem autorskim

Aquí tenemos una copia de los resultados desplegados en la pantalla de mi computadora (observe que los caracteres que introduje están en negritas).

SALIDA

```
Introduzca un carácter:
(introduzca x para terminar)
H
H
i
i
x
x
El ciclo while ha terminado. ¡Hasta pronto!
```

ANÁLISIS Como puede ver en los resultados, el programa imprime cada carácter que se teclea, y termina después de `x`.

La línea 8 asigna a la variable `c` el valor `' '` (un carácter de espacio). A esto se le conoce como *inicializar* la variable, y sólo necesitamos inicializarla a un valor distinto de `'x'`.

La línea 10 es la instrucción `while`. La condición que está entre los paréntesis, `c != 'x'`, significa que el ciclo continuará ejecutándose una y otra vez hasta que `c` sea igual a `'x'`. Debido a que acabamos de inicializar a `c` con el carácter `' '`, la relación `c != x` es, desde luego, verdadera. Después del paréntesis de cierre hay una llave de apertura, por lo que el ciclo se ejecutará hasta encontrar una llave de cierre.

Las líneas 11 y 12 leen un carácter y lo imprimen en la pantalla, y al hacerlo asignan el valor del carácter a la variable `c`. La línea 13 es la llave de cierre, lo que indica que la iteración se realizó y la ejecución regresa a la línea 10, la instrucción `while`. El ciclo continuará si el carácter que se introdujo fue diferente de "x"; en caso contrario, `c != x` será lógicamente falsa, y la ejecución pasará a la siguiente instrucción después de la llave de cierre de la línea 13. En este caso, pasa a la llamada a `printf()` de la línea 14.

El ciclo `do-while`

En la instrucción `while` que acabamos de ver, la expresión condicional se coloca al principio del ciclo. Sin embargo, en esta sección verá otra instrucción que se emplea para los ciclos, `do-while`, la cual pone la expresión al final del ciclo. De esta forma se garantiza que las instrucciones del ciclo se ejecuten por lo menos una vez antes de verificar la expresión. Observe que las instrucciones de un ciclo `while` no se ejecutan si la expresión condicional se evalúa como cero la primera vez.

La forma general de la instrucción `do-while` es

```
do {
   instrucción1;
   instrucción2;
   .
   .
   .
} while (expresión);
```

Aquí, las instrucciones que están dentro del bloque se ejecutan una vez, y luego se evalúa la *expresión* a fin de determinar si el ciclo continúa. El ciclo `do-while` continúa si la expresión se evalúa como un valor diferente de cero; en caso contrario, la ejecución sigue en la siguiente instrucción después del ciclo.

Observe que la instrucción `do-while` termina con un punto y coma, lo cual es una diferencia importante con respecto a las instrucciones `if` y `while`.

El programa del listado 7.2 despliega los caracteres desde la `A` a la `G`, así como sus respectivos valores númericos, mediante un ciclo `do-while` que repite la impresión y el incremento.

TECLEE **LISTADO 7.2** Uso de un ciclo `do-while`

```
1:  /* 07L02.c: Uso de un ciclo do-while */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     i = 65;
9:     do {
10:       printf("El valor decimal de %c es %d.\n", i, i);
11:       i++;
12:    } while (i<72);
13:    return 0;
14: }
```

Después de ejecutar el archivo `07L02.exe`, se mostraron en mi pantalla los caracteres de la `A` a la `G`, junto con sus valores numéricos, como sigue:

SALIDA

```
El valor decimal de A es 65.
El valor decimal de B es 66.
El valor decimal de C es 67.
El valor decimal de D es 68.
El valor decimal de E es 69.
El valor decimal de F es 70.
El valor decimal de G es 71.
```

Materiał chroniony prawem autorskim

ANÁLISIS La instrucción de la línea 8 del listado 7.2 inicializa la variable entera `i` con `65`. Dicha variable se declaró en la línea 6.

Las líneas 9 a 12 contienen el ciclo `do-while`. La expresión `i<72` está al final del ciclo, en la línea 12. Cuando el ciclo inicia por primera vez, se ejecutan las instrucciones de las líneas 10 y 11 antes de evaluar la expresión. Debido a que la variable entera `i` contiene el valor inicial de `65`, la función `printf()` de la línea 10 despliega en la pantalla el valor numérico así como el carácter `A` correspondiente.

Después de incrementar en 1 la variable entera `i` en la línea 11, el control del programa llega al final del ciclo `do-while`. Entonces se evalúa la expresión `i<72`. Si la relación de la expresión se cumple todavía, el control del programa salta al inicio del ciclo `do-while`, y se repite el proceso. Cuando la expresión se evalúa como `0` después de incrementar `i` a 72 (`i` es entonces igual a 72 y por lo tanto no es menor que 72), el ciclo `do-while` termina de inmediato.

Ciclos bajo la instrucción `for`

La forma general de la instrucción `for` es

```
for (expresión1; expresión2; expresión3)
   instrucción;
```

o bien,

```
for (expresión1; expresión2; expresión3) {
   instrucción1;
   instrucción2;
   .
   .
   .
}
```

Puede ver en este ejemplo que la instrucción `for` utiliza tres expresiones (*`expresión1`*, *`expresión2`* y *`expresión3`*) que están separadas entre sí por un punto y coma.

Un ciclo `for` puede controlar una sola instrucción, como en el primer ejemplo, o varias instrucciones, como *`instrucción1`* e *`instrucción2`*, colocadas dentro de las llaves (`{` y `}`).

La primera vez que se ejecuta la instrucción `for`, evalúa primero *`expresión1`*, que por lo regular se usa para inicializar una o más variables.

La segunda expresión, *`expresión2`*, actúa del mismo modo que la expresión condicional de un ciclo `while` o `do-while`. Esta segunda expresión se evalúa inmediatamente después de *`expresión1`*, y más adelante se evalúa de nuevo después de cada ciclo exitoso de

la instrucción `for`. Si *expresión2* se evalúa como un valor diferente de cero (verdadero lógico), entonces se ejecutan las instrucciones que están dentro de las llaves. En caso contrario, el ciclo se detiene y la ejecución continúa en la instrucción siguiente al ciclo.

La tercera expresión del ciclo `for`, *expresión3*, no se evalúa la primera vez que se encuentra el ciclo `for`. Sin embargo se evalúa después de cada iteración y antes de que la instrucción verifique de nuevo la *expresión2*.

En la hora 5, "Manejo de la entrada y salida estándar", vio un ejemplo (listado 5.5) que convierte los números decimales del 0 al 15 en hexadecimales. En ese momento, las conversiones tenían que escribirse en instrucciones separadas. Ahora, con la instrucción `for`, puede reescribir el programa del listado 5.5 en una forma muy eficiente. El listado 7.3 muestra la versión reescrita de ese programa.

TECLEE **LISTADO 7.3** Conversión del 0 al 15 a números hexadecimales

```
1:  /* 07L03.c: Conversión del 0 al 15 a números hexadecimales */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     printf("Hex(mayúsculas)   Hex(minúsculas)   Decimal\n");
9:     for (i=0; i<16; i++){
10:       printf("%X                %x                %d\n", i, i, i);
11:    }
12:    return 0;
13: }
```

Obtuve los siguientes resultados después de crear y ejecutar el archivo `07L03.exe` (que, de hecho, son los mismos del archivo ejecutable `05L05.exe` de la hora 5).

SALIDA

```
Hex(mayúsculas)   Hex(minúsculas)   Decimal
0                 0                 0
1                 1                 1
2                 2                 2
3                 3                 3
4                 4                 4
5                 5                 5
6                 6                 6
7                 7                 7
8                 8                 8
9                 9                 9
A                 a                 10
B                 b                 11
C                 c                 12
```

Materiał chroniony prawem autorskim

```
D          d          13
E          e          14
F          f          15
```

ANÁLISIS Veamos ahora el código del listado 7.3. Como usted sabe, la línea 2 incluye el archivo de encabezado `stdio.h` para la función `printf()` que se usa más adelante en el programa.

Dentro del cuerpo de la función `main()`, la instrucción de la línea 6 declara una variable entera, `i`. La línea 8 despliega en la pantalla la línea de encabezado de los resultados.

Las líneas 9 a 11 contienen la instrucción `for`. Observe que la primera expresión de dicha instrucción es `i = 0`, la cual es una expresión de asignación que inicializa a `0` la variable entera `i`.

La segunda expresión de la instrucción `for` es `i < 16`, la cual es una expresión relacional. Esta instrucción se evalúa como diferente de cero (verdadera), siempre y cuando se cumpla la relación que indica el operador menor que (<). Como se mencionó antes, la instrucción `for` evalúa la segunda expresión cada vez que termina una iteración con éxito. Si el valor de `i` es menor que `16`, lo que significa que la expresión relacional sigue siendo verdadera, la instrucción `for` iniciará otra iteración. En caso contrario detendrá el ciclo y saldrá de él.

La tercera expresión de la instrucción `for` es `i++`. Cuando se evalúa esta expresión, la variable entera `i` se incrementa en `1`. Esto se hace después de que se ejecutan las instrucciones que están dentro del cuerpo de la instrucción `for`. Aquí no importa si en la tercera expresión se utiliza el operador de posincremento (`i++`) o el operador de preincremento (`++i`).

En otras palabras, cuando se encuentra el ciclo `for` por vez primera, se asigna `0` a `i` y se evalúa la expresión `i<16`, la cual es verdadera; por lo tanto se ejecutan las instrucciones que están dentro del cuerpo del ciclo `for`. Después de la ejecución del ciclo `for`, se ejecuta la instrucción `i++` incrementando `i` en `1`, y se evalúa de nuevo `i<16` como verdadera, por lo que se ejecuta otra vez el cuerpo del ciclo. Las iteraciones siguen hasta que la expresión condicional `i<16` ya no sea verdadera.

Como puede ver en la línea 10, sólo hay una instrucción dentro del cuerpo del ciclo `for`. La instrucción contiene la función `printf()`, la cual se emplea para desplegar los números hexadecimales (tanto en mayúsculas como en minúsculas) convertidos a partir de los valores decimales utilizando los especificadores de formato `%X` y `%x`.

La variable entera `i` proporciona el valor decimal. Como vimos anteriormente, `i` contiene el valor inicial de `0` justo antes y durante el primer ciclo. Después de cada iteración, `i` se incrementa en `1` debido a la tercera expresión, `i++`, de la instrucción `for`. El último valor

que proporciona `i` es `15`. Cuando `i` llega a `16`, la relación que indica la segunda expresión, `i<16`, ya no es verdadera. Por lo tanto, el ciclo se detiene y la ejecución de la instrucción `for` termina.

Entonces, la instrucción de la línea 12 devuelve `0` para indicar una terminación normal del programa y, por último, la función `main()` termina y regresa el control al sistema operativo.

Como puede ver, con la instrucción `for` puede escribir un programa muy conciso. De hecho, el programa del listado 7.3 es 11 líneas más breve que el del listado 5.5, aunque ambos programas hacen exactamente lo mismo.

Sin embargo, puede hacer aún más corto el programa del listado 7.3 descartando las llaves (`{` y `}`), ya que sólo hay una instrucción dentro del bloque de instrucciones.

La instrucción nula

Como quizá haya notado, la instrucción `for` no termina con un punto y coma. Esta instrucción tiene un bloque de instrucciones que termina con una llave de cierre (`}`), o una sola instrucción que termina con un punto y coma. La siguiente instrucción `for` contiene una sola instrucción:

```
for (i=0; i<8; i++)
      sum += i;
```

Observe que se quitaron las llaves (`{` y `}`), debido a que la instrucción `for` contiene solamente una instrucción.

Consideremos ahora una instrucción como ésta:

```
for (i=0; i<8; i++);
```

Aquí un punto y coma sigue a la instrucción `for`.

En el lenguaje C hay una instrucción especial denominada *instrucción nula*. Una instrucción nula sólo contiene un punto y coma. En otras palabras, una instrucción nula es una instrucción sin ninguna expresión.

Por lo tanto, al revisar la instrucción `for (i=0; i<8; i++);`, puede ver que en realidad se trata de una instrucción `for` con una instrucción nula. En otras palabras, puede reescribirla como

```
for (i=0; i<8; i++)
   ;
```

Debido a que la instrucción nula no tiene ninguna expresión, la expresión `for` no hace otra cosa que iterar. Más adelante verá algunos ejemplos del uso de la instrucción nula con la instrucción `for`.

Materiał chroniony prawem autorskim

> Ya que la instrucción nula es perfectamente legal en C, debe poner atención al colocar puntos y comas en sus instrucciones `for`. Por ejemplo, suponga que pretende escribir un ciclo como éste:
>
> ```
> for (i=0; i<8; i++)
>    suma += i;
> ```
>
> Sin embargo, si accidentalmente coloca un punto y coma al final de la instrucción `for` como aquí,
>
> ```
> for (i=0; i<8; i++);
>    sum += i;
> ```
>
> el compilador de C seguirá aceptándola, pero los resultados de las dos instrucciones serán muy distintos. (Para un ejemplo, vea el ejercicio 1 de esta lección.)

7

Sólo recuerde que el ciclo `do-while` es la única instrucción de ciclos que utiliza un punto y coma inmediatamente después de ella, como parte de su sintaxis. Las instrucciones `while` y `for` están seguidas por un ciclo, el cual puede ser una sola instrucción seguida de un punto y coma, un bloque de instrucciones entre llaves { y } al que no le sigue un punto y coma, o simplemente un punto y coma (instrucción nula).

Uso de expresiones complejas en una instrucción `for`

El lenguaje C le permite utilizar el operador de coma para combinar varias expresiones dentro de las tres partes de la instrucción `for`.

Por ejemplo, la siguiente forma es válida en C:

```
for (i=0, j=10; i!=j; i++, j--){
   /* bloque de instrucciones */
}
```

En la primera expresión de este ejemplo se inicializan las dos variables enteras `i` y `j` con `0` y `10`, respectivamente, cuando se encuentra por primera vez la instrucción `for`. Luego, en el segundo campo, se evalúa y verifica la expresión relacional `i!=j`. Si se evalúa como cero (falsa), el ciclo termina. En la tercera expresión, `i` se incrementa en `1` y `j` se reduce en `1` después de cada iteración del ciclo. Entonces se evalúa de nuevo la expresión `i!=j` para determinar si se ejecuta otra vez el ciclo.

Veamos ahora un programa real. El listado 7.4 muestra un ejemplo de cómo utilizar varias expresiones en la instrucción `for`.

TECLEE **LISTADO 7.4** Incorporación de varias expresiones a la instrucción `for`

```
1:  /* 07L04.c: Varias expresiones */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=0, j=8; i<8; i++, j--)
9:        printf("%d  +  %d  =  %d\n", i, j, i+j);
10:    return 0;
11: }
```

Obtuve los siguientes resultados después de ejecutar el archivo `07L04.exe`:

SALIDA

```
0  +  8  =  8
1  +  7  =  8
2  +  6  =  8
3  +  5  =  8
4  +  4  =  8
5  +  3  =  8
6  +  2  =  8
7  +  1  =  8
```

ANÁLISIS La línea 6 del listado 7.4 declara dos variables enteras, `i` y `j`, las cuales se utilizan en un ciclo `for`.

En la primera expresión de la instrucción `for`, en la línea 8, `i` se inicializa con `0` y a `j` se le asigna el valor de `8`. La segunda expresión contiene una condición, `i < 8`, la cual le indica a la computadora que continúe las iteraciones, siempre y cuando el valor de `i` sea menor que `8`.

En cada iteración, después de que se ejecuta la instrucción de la línea 9 controlada por el `for`, se evalúa la tercera expresión, lo que provoca que se aumente (incremente) `i` en 1, mientras que `j` se reduce (decrementa) en 1. Debido a que sólo hay una instrucción dentro del ciclo `for`, no se utilizan llaves (`{` y `}`) para formar un bloque de instrucciones.

Durante el ciclo, la instrucción de la línea 9 despliega en la pantalla la suma de `i` y `j`, lo cual produce ocho resultados, uno en cada iteración, mediante la suma de los valores de las dos variables, `i` y `j`.

Agregar varias expresiones dentro de la instrucción `for` es una forma muy conveniente de manipular más de una variable en un ciclo. Para aprender más sobre cómo utilizar varias expresiones en un ciclo `for`, observe el ejemplo del listado 7.5.

Materiał chroniony prawem autorskim

TECLEE

LISTADO 7.5 Otro ejemplo de cómo utilizar varias expresiones en la instrucción `for`

```
1:  /* 07L05.c: Otro ejemplo de varias expresiones */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=0, j=1; i<8; i++, j++)
9:        printf("%d  -  %d  =  %d\n", j, i, j - i);
10:    return 0;
11: }
```

Los siguientes resultados se mostraron en la pantalla de mi máquina después de ejecutar el archivo `07L05.exe`:

SALIDA

```
1 - 0 = 1
2 - 1 = 1
3 - 2 = 1
4 - 3 = 1
5 - 4 = 1
6 - 5 = 1
7 - 6 = 1
8 - 7 = 1
```

ANÁLISIS

En la línea 6 del programa del listado 7.5 se declaran dos variables enteras, `i` y `j`.

Observe que en la primera expresión de la instrucción `for` de la línea 8 hay dos expresiones de asignación, `i=0` y `j=1`. Estas dos expresiones de asignación inicializan las variables enteras `i` y `j`, respectivamente.

En el segundo campo hay una expresión relacional, `i<8`, la cual es la condición que se debe satisfacer antes de poder llevar a cabo el ciclo. Debido a que `i` comienza en `0` y se incrementa en `1` después de cada ciclo, hay un total de 8 iteraciones que realizará la instrucción `for`.

La tercera expresión contiene dos expresiones, `i++` y `j++`, que incrementan en `1` las dos variables enteras después de que se ejecuta la instrucción de la línea 9.

La función `printf()` de la línea 9 despliega la resta de las dos variables enteras, `j` e `i`, dentro del ciclo `for`. Debido a que solamente hay una instrucción en el bloque de instrucciones, las llaves (`{` y `}`) no son necesarias.

Uso de ciclos anidados

Con frecuencia necesita crear un ciclo aunque ya esté dentro de uno. Puede colocar un ciclo (un ciclo interno) dentro de otro (un ciclo externo) para formar *ciclos anidados*. Cuando el programa llegue al ciclo interno, éste se ejecutará como cualquier otra instrucción dentro del ciclo externo.

El listado 7.6 es un ejemplo de cómo funcionan los ciclos anidados.

TECLEE **LISTADO 7.6** Uso de ciclos anidados

```
1:  /* 07L06.c: Demostración de ciclos anidados */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=1; i<=3; i++) {   /* ciclo externo */
9:        printf("Inicio de la iteración %d del ciclo externo.\n", i);
10:       for (j=1; j<=4; j++)  /* ciclo interno */
11:          printf("    Iteración %d del ciclo interno.\n", j);
12:       printf("Fin de la iteración %d del ciclo externo.\n", i);
13:    }
14:    return 0;
15: }
```

Los siguientes resultados se obtuvieron mediante la ejecución del archivo `07L06.exe`:

SALIDA

```
Inicio de la iteración 1 del ciclo externo.
    iteración 1 del ciclo interno.
    iteración 2 del ciclo interno.
    iteración 3 del ciclo interno.
    iteración 4 del ciclo interno.
Fin de la iteración 1 del ciclo externo.
Inicio de la iteración 2 del ciclo externo.
    iteración 1 del ciclo interno.
    iteración 2 del ciclo interno.
    iteración 3 del ciclo interno.
    iteración 4 del ciclo interno.
Fin de la iteración 2 del ciclo externo.
Inicio de la iteración 3 del ciclo externo.
    iteración 1 del ciclo interno.
    iteración 2 del ciclo interno.
    iteración 3 del ciclo interno.
    iteración 4 del ciclo interno.
Fin de la iteración 3 del ciclo externo.
```

Materiał chroniony prawem autorskim

ANÁLISIS En el listado 7.6 se anidan dos ciclos `for`. El ciclo externo comienza en la línea 8 y termina en la línea 13, mientras que el ciclo interno inicia en la línea 10 y termina en la 11.

El ciclo interno sólo comprende una instrucción que imprime el número de iteración de acuerdo con el valor numérico de la variable entera `j`. Como puede ver en la línea 10, `j` se inicializa con `1`, y se incrementa en `1` después de cada ciclo (es decir, de cada iteración). La ejecución del ciclo interno se detiene cuando el valor de `j` es mayor que `4`.

Además del ciclo interior, el ciclo externo tiene otras dos instrucciones en las líneas 9 y 12, respectivamente. La función `printf()` de la línea 9 despliega un mensaje que muestra el inicio de una iteración del ciclo externo. En la línea 12 se imprime un mensaje para mostrar el final de la iteración del ciclo externo.

Puede ver en los resultados que el ciclo interno termina antes de que el ciclo externo comience otra iteración. Cuando el ciclo externo inicia otra iteración, se encuentra con el ciclo interno y lo ejecuta de nuevo. Los resultados del programa del listado 7.6 muestran claramente el orden de ejecución de los ciclos interno y externo.

No confunda los dos operadores relacionales (< y <=) y no los utilice en forma equivocada en las expresiones de los ciclos.

Por ejemplo, lo siguiente

```
for (j=1; j<10; j++){
    /* bloque de instrucciones */
}
```

significa que si j es menor que 10, el ciclo continúa. De esta manera, el número total de iteraciones es 9. Sin embargo, en el siguiente ejemplo,

```
for (j=1; j<=10; j++){
    /* bloque de instrucciones */
}
```

el número total de iteraciones es 10 debido a que en este caso se evalúa la expresión relacional j<=10. Observe que la expresión se evalúa como 1 (verdadero lógico) siempre y cuando j sea menor o igual a 10.

Por lo tanto, puede ver que la diferencia entre los operadores < y <= hace que el ciclo del primer ejemplo tenga una iteración menos que el del segundo ejemplo.

Resumen

En esta lección aprendió los siguientes conceptos e instrucciones importantes:

- Se pueden utilizar ciclos para realizar una y otra vez la misma instrucción hasta que se satisfagan las condiciones especificadas.
- Los ciclos hacen que su programa sea más conciso.
- En C hay tres instrucciones que se utilizan para los ciclos: `while`, `do-while` y `for`.
- La instrucción `while` contiene una expresión, la cual es la expresión condicional que controla el ciclo.
- La instrucción `while` no termina con un punto y coma.
- La instrucción `do-while` coloca su expresión condicional al final del ciclo.
- La instrucción `do-while` termina con un punto y coma.
- Hay tres expresiones en la instrucción `for`. La segunda es la expresión condicional.
- La instrucción `for` no termina con un punto y coma.
- En la instrucción `for` se pueden usar varias expresiones, combinadas mediante comas.
- En un ciclo anidado, el ciclo interno termina antes de que el ciclo externo continúe su iteración.

En la siguiente lección aprenderá acerca de más operadores utilizados en el lenguaje C.

Preguntas y respuestas

P ¿Cuál es la diferencia entre las instrucciones `while` y `do-while`?

R La principal diferencia es que en la instrucción `while` se evalúa la expresión condicional al inicio del ciclo, mientras que en la instrucción `do-while` se evalúa al final del ciclo. Por lo tanto, se garantiza que las instrucciones que controla la instrucción `do-while` se ejecuten por lo menos una vez, mientras que en una instrucción `while` es posible que no se ejecuten nunca.

P ¿Cómo funciona un ciclo `for`?

R En la instrucción `for` hay tres expresiones. El primer campo contiene un inicializador que se evalúa primero y sólo una vez antes de la iteración. La segunda expresión es la expresión condicional, la cual se debe evaluar como diferente de cero (verdadero lógico) antes de que se ejecuten las instrucciones que controla la instrucción `for`. Si la expresión condicional se evalúa como un valor distinto de cero (verdadera), lo que significa que se satisface la condición, se lleva a cabo una

Materiał chroniony prawem autorskim

iteración del ciclo `for`. Después de cada iteración se evalúa la tercera expresión, y luego se evalúa de nuevo la segunda. Este proceso con la segunda y tercera expresiones se repite hasta que la expresión condicional se evalúe como cero (falso lógico).

P ¿Puede terminar con un punto y coma la instrucción `while`?

R Por definición, la instrucción `while` no termina con un punto y coma. Sin embargo, en C es válido poner un punto y coma justo después de la instrucción `while` como éste: `while(expresión);`, lo que significa que hay una instrucción nula controlada por la instrucción `while`. Recuerde que el resultado puede ser muy diferente al que espera si coloca accidentalmente un punto y coma al final de la instrucción `while`.

P Si se anidan juntos dos ciclos, ¿cuál debe terminar primero, el interno o el externo?

R El ciclo interno debe terminar primero. Luego continuará el ciclo externo hasta el final, y después comenzará otra iteración si aún se cumple la condición especificada.

7

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Puede imprimir algo el siguiente ciclo `while`?

```
int k = 100;
while (k<100){
    printf("%c", k);
    k++;
}
```

2. ¿Puede imprimir algo el siguiente ciclo `do-while`?

```
int k = 100;
do {
    printf("%c", k);
    k++;
} while (k<100);
```

3. ¿Tienen el mismo número de iteraciones los siguientes ciclos `for`?

```
for (j=0; j<8; j++);
for (k=1; k<=8; k++);
```

4. ¿Es el siguiente ciclo `for`

   ```
   for (j=65; j<72; j++) printf("%c", j);
   ```

   equivalente al siguiente ciclo `while`?

   ```
   int k = 65;
   while (k<72)
       printf("%c", k);
       k++;
   }
   ```

Ejercicios

1. ¿Cuál es la diferencia entre los siguientes fragmentos de código?

   ```
   for (i=0, j=1; i<8; i++, j++)
        printf("%d  +  %d  =  %d\n", i, j, i+j);
   for (i=0, j=1; i<8; i++, j++);
        printf("%d  +  %d  =  %d\n", i, j, i+j);
   ```

2. Escriba y ejecute un programa que contenga los dos fragmentos de código mostrados en el ejercicio 1. ¿Qué verá en la pantalla?
3. Reescriba el programa del listado 7.1. Esta vez hágalo de manera que la instrucción `while` continúe iterando hasta que el usuario introduzca el carácter `K`.
4. Reescriba el programa mostrado en el listado 7.2 reemplazando el ciclo `do-while` con un ciclo `for`.
5. Reescriba el programa del listado 7.6. Esta vez utilice un ciclo `while` como ciclo externo, y un ciclo `do-while` como ciclo interno.

Materiał chroniony prawem autorskim

Paso de apuntadores a funciones

Como usted sabe, un nombre de arreglo al que no le sigue un índice se interpreta como un apuntador al primer elemento del arreglo. De hecho, la dirección del primer elemento de un arreglo es la dirección de inicio del mismo. Por lo tanto, puede asignar la dirección de inicio de un arreglo a un apuntador y pasar después a la función el nombre del apuntador, en lugar del arreglo sin especificación de tamaño.

El listado 16.5 muestra un ejemplo del paso de apuntadores a funciones, lo cual es similar a pasar arreglos a funciones.

TECLEE **LISTADO 16.5** Paso de apuntadores a funciones

```
1:  /* 16L05.c: Paso de apuntadores a funciones */
2:  #include <stdio.h>
3:
4:  void ImpCh(char *ch);
5:  int SumaDatos(int *lista, int max);
6:  main()
7:  {
8:     char cadena[] = "¡Es una cadena!";
9:     char *aptr_cadena;
10:    int lista[5] = {1, 2, 3, 4, 5};
11:    int *aptr_int;
12:
13:    /* asigna la dirección al apuntador */
14:    aptr_cadena = cadena;
15:    ImpCh(aptr_cadena);
16:    ImpCh(cadena);
17:
18:    /* asigna la dirección al apuntador */
19:    aptr_int = lista;
20:    printf("La suma devuelta por SumaDatos() es: %d\n",
21:             SumaDatos(aptr_int, 5));
22:    printf("La suma devuelta por SumaDatos() es: %d\n",
23:             SumaDatos(lista, 5));
24:    return 0;
25: }
26: /* definición de función */
27: void ImpCh(char *ch)
28: {
29:    printf("%s\n", ch);
30: }
31: /* definición de función */
32: int SumaDatos(int *lista, int max)
33: {
34:    int i;
35:    int suma = 0;
36:
```

Materiał chroniony prawem autorskim

Después de ejecutar en mi máquina el archivo `16L04.exe`, e introducir tres enteros, `10`, `20` y `30`, obtuve los siguientes resultados:

SALIDA

```
Escriba tres enteros separados por espacios:
10  20  30
La suma de los tres enteros es: 60
```

ANÁLISIS La finalidad del programa del listado 16.4 es obtener tres enteros introducidos por el usuario, y luego pasarlos como un arreglo a una función denominada `SumaTres()` para realizar una operación de suma.

La línea 4 declara la función `SumaTres()`. Observe que en la expresión del argumento se usa el arreglo sin dimensionar `lista[]`, lo cual indica que el argumento contiene la dirección de inicio del arreglo `lista`.

En la línea 8 se declaran el arreglo `lista` y la variable entera `suma`. La llamada a `printf()` de la línea 10 despliega un mensaje que solicita al usuario que introduzca tres enteros. Luego, la línea 11 utiliza `scanf()` para recuperar los enteros que escribió el usuario y almacenarlos dentro de las tres ubicaciones de memoria de los elementos del arreglo entero al que hacen referencia `&lista[0]`, `&lista[1]` y `&lista[2]`, respectivamente.

La instrucción de la línea 12 llama a la función `SumaTres()` con el nombre del arreglo como argumento. En realidad, la expresión `SumaTres(lista)` está pasando la dirección de inicio del arreglo `lista` (`&lista[0]`) a la función `SumaTres()`.

La definición de la función `SumaTres()` está en las líneas 18 a 26; la función suma los valores de los tres elementos del arreglo y devuelve el resultado de la suma. Dicho resultado se asigna a la variable entera suma en la línea 12 y se imprime en la línea 13.

También puede especificar el tamaño de un arreglo que se pasa a una función. Por ejemplo:

```
funcion(char cadena[16]);
```

equivale a la instrucción

```
funcion(char cadena[]);
```

Recuerde que el compilador puede determinar el tamaño del arreglo sin especificación de tamaño cadena[].

En arreglos multidimensionales, siempre se debe usar en la declaración el formato de un arreglo sin especificación de tamaño. (Vea, más adelante en esta hora, la sección "Cómo pasar arreglos multidimensionales como argumentos".)

En la línea 19 se asigna la dirección de inicio del arreglo `lista` de tipo `int` al apuntador `aptr_int`. Antes de hacer algo con el elemento `lista[2]` de este arreglo, imprimo su valor, que en este momento es 3 (vea el resultado que produce la línea 20). En la línea 21 se le da otro valor al elemento `lista[2]`, -3, a través del operador de indirección `*(aptr_int + 2)`. La llamada a `printf()` de la línea 22 imprime el valor actualizado de `lista[2]`.

Apuntadores y funciones

Antes de comenzar a hablar acerca de cómo pasar apuntadores a funciones, veamos primero cómo pasar arreglos a funciones.

Paso de arreglos a funciones

En la práctica, por lo regular es difícil pasar más de cinco o seis argumentos a una función. Una forma de ahorrar el número de argumentos que se pasan a una función es utilizar arreglos. Puede poner todas las variables del mismo tipo en un arreglo y luego transferir el arreglo como un solo argumento.

El programa del listado 16.4 muestra cómo pasar un arreglo de enteros a una función.

TECLEE **LISTADO 16.4** Cómo pasar arreglos a funciones

```
1:  /* 16L04.c: Cómo pasar arreglos a funciones */
2:  #include <stdio.h>
3:
4:  int SumaTres (int lista[]);
5:
6:  main()
7:  {
8:     int suma, lista[3];
9:
10:    printf("Escriba tres enteros separados por espacios:\n");
11:    scanf("%d%d%d", &lista[0], &lista[1], &lista[2]);
12:    suma = SumaTres(lista);
13:    printf("La suma de los tres enteros es: %d\n", suma);
14:
15:    return 0;
16: }
17:
18: int SumaTres(int lista[])
19: {
20:    int i;
21:    int resultado = 0;
22:
23:    for (i=0; i<3; i++)
24:       resultado += lista[i];
25:    return resultado;
26: }
```

Materiał chroniony prawem autorskim

```
6:     char cadena[] = "¡Es una cadena!";
7:     char *aptr_cadena;
8:     int lista[] = {1, 2, 3, 4, 5};
9:     int *aptr_int;
10:
11:    /* acceso al arreglo de tipo char */
12:    aptr_cadena = cadena;
13:    printf("Antes del cambio, cadena contiene: %s\n", cadena);
14:    printf("Antes del cambio, cadena[4] contiene: %c\n", cadena[4]);
15:    *(aptr_cadena + 4) = 'U';
16:    printf("Después del cambio, cadena[4] contiene: %c\n", cadena[4]);
17:    printf("Después del cambio, cadena contiene: %s\n", cadena);
18:    /* acceso al arreglo de tipo int */
19:    aptr_int = lista;
20:    printf("Antes del cambio, lista[2] contiene: %d\n", lista[2]);
21:    *(aptr_int + 2) = -3;
22:    printf("Después del cambio, lista[2] contiene: %d\n", lista[2]);
23:
24:    return 0;
25: }
```

Después de crear y ejecutar en mi computadora el archivo `16L03.exe`, se desplegaron los siguientes resultados:

SALIDA

```
Antes del cambio, cadena contiene: ¡Es una cadena!
Antes del cambio, cadena[4] contiene: u
Después del cambio, cadena[4] contiene: U
Después del cambio, cadena contiene: ¡Es Una cadena!
Antes del cambio, lista[2] contiene: 3
Después del cambio, lista[2] contiene: -3
```

ANÁLISIS La finalidad del programa del listado 16.3 es mostrarle cómo acceder a un arreglo de tipo `char`, `cadena`, y a un arreglo de tipo `int`, `lista`. En las líneas 6 y 8 se declaran `cadena` y `lista` y se inicializan con una cadena y con un conjunto de enteros, respectivamente. En las líneas 7 y 9 se declaran un apuntador de tipo `char`, `aptr_cadena`, y un apuntador de tipo `int`, `aptr_int`.

La línea 12 asigna la dirección de inicio del arreglo `cadena` al apuntador `aptr_cadena`. Las instrucciones de las líneas 13 y 14 muestran el contenido de la cadena guardada en el arreglo `cadena`, así como el carácter contenido por el elemento `cadena[4]` del arreglo antes de cambiar `cadena`.

La instrucción de la línea 15 muestra que la constante de carácter `'U'` se asigna al elemento del arreglo `cadena` al que apunta la expresión `*(aptr_cadena + 4)`.

Para verificar que el contenido del elemento cadena se haya actualizado, las líneas 16 y 17 imprimen el elemento y la cadena completa, respectivamente. Los resultados indican que `'U'` sustituyó a la constante de carácter original, `'u'`.

La instrucción de la línea 11 muestra la diferencia entre los dos apuntadores `int`, es decir, la resta de `aptr_int2` y `aptr_int1`. El resultado es `5`.

La línea 12 asigna entonces otra dirección de memoria al apuntador `aptr_int2`, que es a la que hace referencia la expresión `aptr_int1-5`. `aptr_int2` apunta ahora a una ubicación de memoria que está 10 bytes más abajo que la ubicación de memoria a la que apunta `aptr_int1` (vea el resultado que produce la línea 13). La diferencia entre `aptr_int2` y `aptr_int1` se obtiene mediante la resta de los dos apuntadores, la cual es `-5` (ya que en mi máquina un `int` es de dos bytes) como lo imprimió la instrucción de la línea 14.

Apuntadores y arreglos

Como se indicó en lecciones anteriores, los apuntadores y los arreglos tienen una estrecha relación. Usted puede acceder a un arreglo a través de un apuntador que contenga la dirección de inicio del arreglo. La siguiente subsección presenta la forma de acceder a elementos de un arreglo a través de apuntadores.

Acceso de arreglos mediante apuntadores

Debido a que un nombre de arreglo al que no le sigue un índice se interpreta como un apuntador al primer elemento del arreglo, usted puede asignar la dirección de inicio de un arreglo a un apuntador del mismo tipo de datos; luego puede acceder a cualquier elemento del arreglo sumando el entero apropiado al apuntador. El valor del entero que utiliza es el mismo que el valor del índice del elemento al que desea acceder.

En otras palabras, dado un arreglo, `arreglo`, y un apuntador, `aptr_arreglo`, si `arreglo` y `aptr_arreglo` son del mismo tipo de datos, y `aptr_arreglo` contiene la dirección de inicio del arreglo, es decir

```
aptr_arreglo = arreglo;
```

entonces la expresión `arreglo[n]` equivale a la expresión

```
*(aptr_arreglo + n)
```

Aquí, `n` es el índice del arreglo.

El listado 16.3 muestra cómo acceder a arreglos y modificar valores de elementos de arreglos utilizando apuntadores

TECLEE **LISTADO 16.3** Cómo acceder a arreglos por medio de apuntadores

```
1:  /* 16L03.c: Cómo acceder a arreglos mediante apuntadores */
2:  #include <stdio.h>
3:
4:  main()
5:  {
```

Materiał chroniony prawem autorskim

Resta de apuntadores

Usted puede restar el valor de un apuntador de otro para obtener la distancia entre las dos ubicaciones de memoria. Por ejemplo, dadas dos variables de apuntador de tipo `char`, `aptr_cadena1` y `aptr_cadena2`, puede calcular el desplazamiento entre las dos ubicaciones de memoria a las que apuntan ambos apuntadores, de la siguiente manera:

```
aptr_cadena2 - aptr_cadena1
```

Para obtener resultados significativos, es mejor restar solamente apuntadores del mismo tipo de datos.

El listado 16.2 muestra un ejemplo de cómo realizar una resta en una variable de apuntador de tipo `int`.

TECLEE **LISTADO 16.2** Cómo realizar una resta en apuntadores

```
1:  /* 16L02.c: Resta de apuntadores */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int *aptr_int1, *aptr_int2;
7:
8:     printf("Posición de aptr_int1: %p\n", aptr_int1);
9:     aptr_int2 = aptr_int1 + 5;
10:    printf("Posición de aptr_int2 = aptr_int1 + 5: %p\n", aptr_int2);
11:    printf("La resta de aptr_int2 - aptr_int1: %d\n", aptr_int2 - aptr_int1);
12:    aptr_int2 = aptr_int1 - 5;
13:    printf("Posición de aptr_int2 = aptr_int1 - 5: %p\n", aptr_int2);
14:    printf("La resta de aptr_int2 - aptr_int1: %d\n", aptr_int2 - aptr_int1);
15:
16:    return 0;
17: }
```

Después de ejecutar el archivo `16L02.exe` en mi máquina, se desplegaron los siguientes resultados en la pantalla:

SALIDA

```
Posición de aptr_int1: 0x0128
Posición de aptr_int2 = aptr_int1 + 5: 0x0132
La resta de aptr_int2 - aptr_int1: 5
Posición de aptr_int2 = aptr_int1 - 5: 0x011E
La resta de aptr_int2 - aptr_int1: -5
```

ANÁLISIS El programa del listado 16.2 declara dos variables de apuntador de tipo `int`, `aptr_int1` y `aptr_int2`, en la línea 6. La instrucción de la línea 8 imprime la posición de memoria que contiene `aptr_int1`. La línea 9 asigna a `aptr_int2` la dirección de memoria a la que hace referencia `aptr_int1+5`. Después se imprime el contenido de `aptr_int2` en la línea 10.

significa moverse a la ubicación de memoria que está un byte más arriba que la ubicación actual, `0x000B`, a la que hace referencia el apuntador `aptr_ch`.

La línea 16 muestra la ubicación de memoria a la que hace referencia la variable de apuntador de tipo `int`, `aptr_int`, en `0x028B`. Debido a que en mi sistema el tamaño de `int` es de 2 bytes, la expresión `aptr_int+1` mueve a `aptr_int` a la ubicación de memoria que está 2 bytes más arriba que la ubicación actual a la que apunta `aptr_int`. Ése es exactamente el resultado que usted ve en la línea 17. De la misma manera, la línea 18 muestra que `aptr_int+2` se mueve para hacer referencia a `0x028F`, que está 4 bytes más arriba (`2*sizeof(int)`) que `0x028B`. La expresión `aptr_int-1` de la línea 19 hace referencia a la ubicación de memoria `0x0289`; y `aptr_int-2`, de la línea 20, hace referencia a la ubicación `0x0287`.

En mi sistema, el tamaño del tipo de datos `double` es de 8 bytes de longitud. Por lo tanto, la expresión `aptr_db+1` se interpreta como la dirección de memoria a la que hace referencia `aptr_db` más `8` bytes, es decir, `0x0128+8`, lo cual da `0x0130` en formato hexadecimal, como puede ver en la línea 23.

Las líneas 24 a 26 imprimen las direcciones de memoria a las que hacen referencia `aptr_db+2`, `aptr_db-1` y `aptr_db-2`, respectivamente, lo cual demuestra que el compilador utilizó el mismo tamaño escalar de `double` en la aritmética de los apuntadores.

Los apuntadores son muy útiles cuando se usan en forma adecuada. Sin embargo, un apuntador puede meterle en problemas con rapidez si contiene un valor equivocado. Por ejemplo, un error común es asignar un valor derecho a un apuntador que en realidad espera un valor izquierdo. Afortunadamente, muchos compiladores de C detectarán dichos errores y emitirán un mensaje de advertencia.

Existe otro error común que el compilador no siempre detecta: utilizar apuntadores sin inicializar. Por ejemplo, el siguiente código tiene un problema potencial:

```
int x, *aptr_int;
x = 8;
*aptr_int = x;
```

El problema es que el apuntador aptr_int no está inicializado; apunta a una ubicación de memoria desconocida. Por lo tanto, asignar un valor, como en este caso 8, a una ubicación de memoria desconocida, resulta peligroso. Podría sobreescribir algún dato importante que ya esté guardado en esa ubicación de memoria, lo que causaría un serio problema. La solución es asegurarse de que un apuntador esté apuntando a una ubicación de memoria válida y legal antes de utilizarlo.

Para evitar el problema potencial, puede escribir el código de arriba de la siguiente manera:

```
int x, *aptr_int;
x = 8;
aptr_int = &x;   /* inicializa el apuntador */
```

Materiał chroniony prawem autorskim

```
11:     printf("Posición después de aptr_ch + 1: %p\n", aptr_ch + 1);
12:     printf("Posición después de aptr_ch + 2: %p\n", aptr_ch + 2);
13:     printf("Posición después de aptr_ch - 1: %p\n", aptr_ch - 1);
14:     printf("Posición después de aptr_ch - 2: %p\n", aptr_ch - 2);
15:     /* apuntador de tipo int aptr_int */
16:     printf("Posición actual de aptr_int: %p\n", aptr_int);
17:     printf("Posición después de aptr_int + 1: %p\n", aptr_int + 1);
18:     printf("Posición después de aptr_int + 2: %p\n", aptr_int + 2);
19:     printf("Posición después de aptr_int - 1: %p\n", aptr_int - 1);
20:     printf("Posición después de aptr_int - 2: %p\n", aptr_int - 2);
21:     /* apuntador de tipo double aptr_db */
22:     printf("Posición actual de aptr_db: %p\n", aptr_db);
23:     printf("Posición después de aptr_db + 1: %p\n", aptr_db + 1);
24:     printf("Posición después de aptr_db + 2: %p\n", aptr_db + 2);
25:     printf("Posición después de aptr_db - 1: %p\n", aptr_db - 1);
26:     printf("Posición después de aptr_db - 2: %p\n", aptr_db - 2);
27:
28:     return 0;
29: }
```

Obtuve los siguientes resultados al ejecutar en mi máquina el archivo `16L01.exe` del programa del listado 16.1. Usted podría obtener direcciones diferentes en su computadora, así como diferentes desplazamientos, dependiendo del tamaño de los tipos de datos de su sistema.

SALIDA

```
Posición actual de aptr_ch: 0x000B
Posición después de aptr_ch + 1: 0x000C
Posición después de aptr_ch + 2: 0x000D
Posición después de aptr_ch - 1: 0x000A
Posición después de aptr_ch - 2: 0x0009
Posición actual de aptr_int: 0x028B
Posición después de aptr_int + 1: 0x028D
Posición después de aptr_int + 2: 0x028F
Posición después de aptr_int - 1: 0x0289
Posición después de aptr_int - 2: 0x0287
Posición actual de aptr_db: 0x0128
Posición después de aptr_db + 1: 0x0130
Posición después de aptr_db + 2: 0x0138
Posición después de aptr_db - 1: 0x0120
Posición después de aptr_db - 2: 0x0118
```

ANÁLISIS Como puede ver en el listado 16.1, hay tres apuntadores de diferentes tipos, `aptr_ch`, `aptr_int` y `aptr_db`, declarados en las líneas 6 a 8. Entre ellos, `aptr_ch` es un apuntador a un carácter, `aptr_int` es un apuntador a un entero, y `aptr_db` es un apuntador a un tipo doble.

Después, la instrucción de la línea 10 muestra la dirección de memoria, `0x000B`, contenida por la variable de apuntador de tipo `char`, `aptr_ch`. Las líneas 11 y 12 despliegan dos direcciones, `0x000C` y `0x000D`, cuando se le suman `1` y `2` a `aptr_ch`, respectivamente. Asimismo, las líneas 13 y 14 dan `0x000A` y `0x0009` cuando se mueve `aptr_ch` a direcciones más bajas de memoria. Debido a que el tamaño de `char` es 1 byte, `aptr_ch+1`

mueve el apuntador a la ubicación de memoria que está a un byte de distancia de la posición actual de `aptr_cadena`.

Observe que los enteros que se suman o se restan a los apuntadores de diferentes tipos de datos tienen tamaños diferentes. En otras palabras, sumar (o restar) 1 a un apuntador no necesariamente le indica al compilador que sume (o reste) un byte a la dirección, sino más bien que ajuste la dirección para que ésta salte sobre un elemento del tipo del apuntador. Verá más detalles acerca de esto en las siguientes secciones.

El tamaño escalar de los apuntadores

La forma general para cambiar la posición de un apuntador es

nombre_apuntador + n

Aquí, *n* es un entero cuyo valor puede ser positivo o negativo. *nombre_apuntador* es el nombre de una variable de apuntador que tiene la siguiente declaración:

*especificador_de_tipo_de_datos *nombre_apuntador*;

Cuando el compilador de C lee la expresión *nombre_apuntador + n*, la interpreta de la siguiente manera:

*nombre_apuntador + n ** `sizeof(`*especificador_de_tipo_de_datos*`)`

Observe que se utiliza el operador `sizeof` para obtener el número de bytes del tipo de datos especificado. Por lo tanto, para la variable de apuntador de tipo `char`, `aptr_cadena`, la expresión `aptr_cadena + 1` significa en realidad

`aptr_cadena + 1 * sizeof(char).`

Debido a que el tamaño de un carácter es de un byte de longitud, `aptr_cadena + 1` le indica al compilador que se mueva a la ubicación de memoria que está 1 byte después de la ubicación actual a la que hace referencia el apuntador.

El programa del listado 16.1 muestra cómo los tamaños escalares de diferentes tipos de datos afectan los desplazamientos que se suman o se restan a los apuntadores.

TECLEE **LISTADO 16.1** Cómo mover apuntadores de diferentes tipos de datos

```
1:  /* 16L01.c: Aritmética de los apuntadores */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char *aptr_ch;
7:     int *aptr_int;
8:     double *aptr_db;
9:     /* apuntador de tipo char aptr_ch */
10:    printf("Posición actual de aptr_ch: %p\n", aptr_ch);
```

Materiał chroniony prawem autorskim

HORA 1

Uso de ap s

Piensa dos veces y actúa una.

—Proverbio chino

En la hora 11, "Apuntadores", las bases del uso de los apuntadores de C. Debido a que los apuntadores muy útiles en la programación, vale la pena dedicar otra hora para más acerca de ellos. En esta lección se exponen los siguientes temas:

- Aritmética de los apuntadores
- Paso de arreglos a funciones
- Paso de apuntadores a
- Apuntadores a funciones

Aritmética de los ntadores

En C, usted puede mover la posición un apuntador sumándole o restándole enteros. Por ejemplo, dada una de apuntador a carácter, `aptr_cadena`, la expresión.

```
aptr_cadena + 1
```

Materiał chroniony prawem autorskim

3. Reescriba el programa del listado 15.3. Esta vez haga una función que tome un número variable de argumentos de tipo `int` y que realice la operación de multiplicación sobre ellos.
4. Escriba otra vez el programa del listado 15.3. En esta ocasión imprima todos los argumentos que se pasen a la función `SumaDobles()`. ¿Extrae `va_arg()` cada argumento de la lista de argumentos que se pasa a `SumaDobles()` en el mismo orden (es decir, de izquierda a derecha)?

Materiał chroniony prawem autorskim

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios del taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. Dadas las siguientes declaraciones de función, ¿cuáles funciones tienen un número fijo de argumentos, cuáles son funciones sin argumentos y cuáles tienen un número variable de argumentos?
    - `int funcion_1(int x, float y);`
    - `void funcion_2(char *cadena);`
    - `char *asctime(const struct tm *timeptr);`
    - `int funcion_3(void);`
    - `char funcion_4(char c, ...);`
    - `void funcion_5(void);`
2. ¿Cuál de las dos expresiones siguientes es una definición de función?
    ```
    int funcion_1(int x, int y);
    int funcion_2(int x, int y){return x+y;}
    ```
3. ¿Qué tipo de datos devuelve una función cuando se omite el especificador de tipo?
4. ¿Cuál de las siguientes declaraciones de función es ilegal?
    - `double funcion_1(int x, ...);`
    - `void funcion_2(int x, int y, ...);`
    - `char funcion_3(...);`
    - `int funcion_4(int, int, int, int);`

Ejercicios

1. Reescriba el programa del listado 15.2. Esta vez utilice el especificador de formato `%c`, en lugar de `%s`, para imprimir en su computadora la cadena de caracteres de la hora local.
2. Declare y defina una función, llamada `MultiDos()`, que pueda multiplicar dos variables enteras. Llame a la función `MultiDos()` desde la función `main()` y pásele dos enteros. Imprima después en la pantalla el resultado que devuelva la función `MultiDos()`.

- `va_start()`, `va_arg()` y `va_end()`, todas incluidas en `stdarg.h`, son necesarias para procesar un número variable de argumentos que se pasa a una función.
- `time()`, `localtime()` y `asctime()` son tres funciones de hora que proporciona C. Se pueden usar juntas para obtener una cadena de caracteres que contenga información acerca de la fecha y hora locales con base en la hora calendario.

En la siguiente lección aprenderá más acerca de los apuntadores y su aplicación en C.

Preguntas y respuestas

P ¿Cuál es la principal diferencia entre una declaración y una definición de función?

R La principal diferencia es que la declaración de una función no reserva ningún espacio de memoria, ni especifica lo que hace la función. Una declaración de función sólo hace referencia a la definición de función que está colocada en otra parte. Además especifica qué tipo de argumentos y valores se pasan a la función y se devuelven de ella. Por otra parte, una definición de función reserva el espacio de memoria y especifica las tareas que puede llevar a cabo la función.

P ¿Por qué son necesarios los prototipos de función?

R Al declarar una función con prototipos, usted especifica no sólo el tipo de datos que devuelve la función, sino también los tipos de datos y nombres de los argumentos que se le pasan. Con la ayuda de un prototipo de función, el compilador puede realizar en forma automática una verificación de tipos sobre la definición de la función, lo cual le ahorra tiempo a usted en la depuración del programa.

P ¿Puede una función devolver un apuntador?

R Sí. De hecho, una función puede devolver un solo valor que sea de cualquier tipo de datos, excepto un arreglo o una función. Un valor de apuntador —es decir, la dirección— devuelto por una función puede hacer referencia a un arreglo de caracteres o a una ubicación de memoria que almacene otros tipos de datos. Por ejemplo, la función `asctime()` de la biblioteca de C devuelve un apuntador que apunta a una cadena de caracteres convertida a partir de una estructura de fecha-hora.

P ¿Es posible utilizar juntas la programación descendente y la programación ascendente para resolver un problema?

R Sí. En la práctica puede descubrir que en realidad es una buena idea combinar ambos enfoques de programación estructurada para resolver problemas. Emplear estos dos tipos de programación puede hacer que su programa sea fácil de escribir y entender.

Materiał chroniony prawem autorskim

Comprensión de la programación estructurada

Ya aprendió lo básico de la declaración y definición de funciones. Antes de pasar a la siguiente hora, hablemos un poco de la *programación estructurada* en el diseño de programas.

La programación estructurada es una de las mejores metodologías de programación. Existen básicamente dos tipos de programación estructurada: descendente y ascendente.

Cuando usted comienza a escribir un programa para resolver un problema, una manera de hacerlo es trabajar sobre las piezas más pequeñas del problema. Primero define y escribe funciones para cada pieza. Después de escribir y probar las funciones, comienza a ensamblarlas para construir un programa que pueda resolver el problema. A este enfoque se le conoce normalmente como *programación ascendente*.

Por otra parte, para resolver un problema puede trabajar primero en un bosquejo y comenzar su programación a un nivel más alto. Por ejemplo, puede trabajar al principio sobre la función `main()`, y pasar después al nivel inferior siguiente hasta escribir las funciones del nivel más bajo. A este tipo de enfoque se le denomina *programación descendente*.

Descubrirá que resulta útil combinar estos dos tipos de programación estructurada y alternar su uso para solucionar problemas reales.

Resumen

En esta lección aprendió los siguientes conceptos importantes acerca de las funciones de C:

- Una declaración de función hace referencia a una función que está definida en otra parte, y especifica qué tipos de argumentos y valores se pasan a la función y se devuelven de ella.
- Una definición de función reserva el espacio de memoria y define lo que hace la función, así como el número y tipo de argumentos que se le pasan.
- Se puede declarar una función para que devuelva cualquier tipo de datos, excepto un arreglo o una función.
- La instrucción `return` que se usa en la definición de funciones devuelve un solo valor, cuyo tipo debe coincidir con el señalado en la declaración de la función.
- Una llamada a una función es una expresión que puede emplearse como una sola instrucción o dentro de otras expresiones o instrucciones.
- El tipo de datos `void` es necesario en la declaración de una función que no toma ningún argumento.
- Para declarar una función que toma un número variable de argumentos, tiene que especificar por lo menos el primer argumento, y usar los puntos suspensivos (`...`) para representar al resto de los argumentos que se pasan a la función.

Después de ejecutar el archivo `15L03.exe`, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
Dado un argumento: 1.5
El total de argumentos es: 1
El resultado que devuelve SumaDobles() es: 1.5

Dados los argumentos: 1.5 y 2.5
El total de argumentos es: 2
El resultado que devuelve SumaDobles() es: 4.0

Dados los argumentos: 1.5, 2.5, y 3.5
El total de argumentos es: 3
El resultado que devuelve SumaDobles() es: 7.5

Dados los argumentos: 1.5, 2.5, 3.5, y 4.5
El total de argumentos es: 4
El resultado que devuelve SumaDobles() es: 12.0
```

ANÁLISIS El programa del listado 15.3 contiene una función que puede tomar un número variable de argumentos de tipo `double`, realizar la operación de suma sobre estos argumentos y devolver después el resultado a la función `main()`.

La declaración de la línea 5 le indica al compilador que la función `SumaDobles()` toma un número variable de argumentos. El primer argumento para `SumaDobles()` es una variable entera que contiene el número de los argumentos restantes que se pasan a la función cada vez que se llama a `SumaDobles()`. En otras palabras, el primer argumento indica el número de los argumentos restantes a procesar.

En las líneas 30 a 42 se da la definición de `SumaDobles()`, y en la línea 32 se declara el arreglo de tipo `va_list`, `listaarg`. Como ya se mencionó, se tiene que llamar a la macro `va_start()` antes de procesar los argumentos. Así, la línea 37 invoca a `va_start()` para inicializar el arreglo `listaarg`. El ciclo `for` de las líneas 38 y 39 extrae el siguiente argumento `double` guardado en el arreglo `listaarg` llamando a `va_arg()`. Luego, cada argumento se suma en una variable local `double` llamada `resultado`.

En la línea 40 se llama a la función `va_end()` después de extraer y procesar todos los argumentos almacenados en `listaarg`. Después se devuelve el valor de `resultado` al llamador de la función `SumaDobles()`, que en este caso es la función `main()`.

Para finalizar el procesamiento de argumentos variables en un programa de C, se debe llamar a la función `va_end()`. De no hacerlo, el comportamiento del programa será impredecible.

Como puede ver, dentro de la función `main()` se llama cuatro veces a `SumaDobles()`, cada vez con un número diferente de argumentos. Estos argumentos que se pasan a `SumaDobles()` se despliegan mediante las llamadas a las funciones `printf()` de las líneas 14, 17, 20 y 23. Además, se imprimen en la pantalla los cuatro diferentes resultados que devuelve `SumaDobles()`.

Materiał chroniony prawem autorskim

El listado 15.3 muestra cómo utilizar `va_start()`, `va_arg()` y `va_end()` en una función que toma un número variable de argumentos.

TECLEE **LISTADO 15.3** Procesamiento de argumentos variables

```
1:  /* 15L03.c: Procesamiento de argumentos variables */
2:  #include <stdio.h>
3:  #include <stdarg.h>
4:
5:  double SumaDobles(int x, ...);
6:
7:  main ()
8:  {
9:     double d1 = 1.5;
10:    double d2 = 2.5;
11:    double d3 = 3.5;
12:    double d4 = 4.5;
13:
14:    printf("Dado un argumento: %2.1f\n", d1);
15:    printf("El resultado que devuelve SumaDobles() es: %2.1f\n\n",
16:       SumaDobles(1, d1));
17:    printf("Dados los argumentos: %2.1f y %2.1f\n", d1, d2);
18:    printf("El resultado que devuelve SumaDobles() es: %2.1f\n\n",
19:       SumaDobles(2, d1, d2));
20:    printf("Dados los argumentos: %2.1f, %2.1f y %2.1f\n", d1, d2, d3);
21:    printf("El resultado que devuelve SumaDobles() es: %2.1f\n\n",
22:       SumaDobles(3, d1, d2, d3));
23:    printf("Dados los argumentos: %2.1f, %2.1f, %2.1f, y %2.1f\n",
24:           d1, d2, d3, d4);
25:    printf("El resultado que devuelve SumaDobles() es: %2.1f\n",
26:       SumaDobles(4, d1, d2, d3, d4));
27:    return 0;
28: }
29: /* definición de SumaDobles() */
30: double SumaDobles(int x, ...)
31: {
32:    va_list   listaarg;
33:    int i;
34:    double resultado = 0.0;
35:
36:    printf("El total de argumentos es: %d\n", x);
37:    va_start (listaarg, x);
38:    for (i=0; i<x; i++)
39:       resultado += va_arg(listaarg, double);
40:    va_end (listaarg);
41:    return resultado;
42: }
```

Por ejemplo, la declaración de la función `printf()` sería algo como lo siguiente:

```
int printf(const char *formato, ...);
```

Procesamiento de argumentos variables

En el archivo de encabezado `stdarg.h` hay tres rutinas declaradas, las cuales le permiten escribir funciones que toman un número variable de argumentos. Éstas son: `va_start()`, `va_arg()` y `va_end()`.

Además, en `stdarg.h` se incluye el tipo de datos `va_list`, el cual define un tipo de arreglo adecuado para contener los elementos de datos que necesitan `va_start()`, `va_arg()` y `va_end()`.

Para inicializar un determinado arreglo que necesitan `va_arg()` y `va_end()`, tiene que utilizar la rutina macro `va_start()` antes de procesar cualquier argumento.

SINTAXIS

La sintaxis de la macro `va_start()` es

```
#include <stdarg.h>
void va_start(va_list ap, nomarg);
```

Aquí, *ap* es el nombre del arreglo que está por inicializar la rutina macro `va_start()`. *nomarg* debe ser el argumento que está antes de los puntos suspensivos (`...`) en la declaración de la función.

Al utilizar la macro `va_arg()`, usted puede tratar con una expresión que tenga el tipo y valor del siguiente argumento. En otras palabras, la macro `va_arg()` se puede utilizar para obtener el siguiente argumento que se pasa a la función.

SINTAXIS

La sintaxis de la macro `va_arg()` es

```
#include <stdarg.h>
type va_arg(va_list ap, tipo_datos);
```

Aquí, *ap* es el nombre del arreglo que inicializa la rutina macro `va_start()`. *tipo_datos* es el tipo de datos del argumento que se pasa a la función.

Para facilitar una devolución normal de su función, debe usar la función `va_end()` en su programa después de procesar todos los argumentos.

SINTAXIS

La sintaxis de la función `va_end()` es

```
#include <stdarg.h>
void va_end(va_list ap);
```

Aquí, *ap* es el nombre del arreglo que inicializa la rutina macro `va_start()`.

Recuerde incluir en su programa el archivo de encabezado `stdarg.h` antes de llamar a `va_start()`, `va_arg()` o `va_end()`.

Materiał chroniony prawem autorskim

22, `asctime(localtime(&ahora))`, obtiene la expresión de hora local de la hora calendario llamando a `localtime()`, y convierte la hora local en una cadena de caracteres con la ayuda de `asctime()`. La cadena de caracteres que representa la fecha y hora se imprime entonces mediante la llamada a `printf()` de las líneas 21 y 22, la cual tiene el siguiente formato:

```
Sat Apr 05 11:50:10 1997\n\0
```

Observe que en la cadena de caracteres convertida y devuelta por la función `asctime()`, hay un carácter de línea nueva agregado justo antes del carácter nulo.

Funciones con un número fijo de argumentos

Ya ha visto varios ejemplos que declaran y llaman funciones con un número fijo de argumentos. Por ejemplo, la declaración de `funcion_1()` de la línea 4 del listado 15.1

```
int funcion_1(int x, int y);
```

contiene el prototipo de dos argumentos, `x` y `y`.

Para declarar una función con un número fijo de argumentos, necesita especificar el tipo de datos de cada argumento. Además se recomienda indicar los nombres de los argumentos, para que el compilador pueda verificar que los tipos y nombres de argumentos señalados en la declaración de una función coincidan con la implementación de la definición de la misma.

Prototipo de un número variable de argumentos

Como tal vez recuerde, la sintaxis de la función `printf()` es

```
int printf(const char *formato[, argumento, ...]);
```

Aquí, los puntos suspensivos (...) representan un número variable de argumentos. En otras palabras, además del primer argumento que es una cadena de caracteres, la función `printf()` puede tomar un número no especificado de argumentos adicionales, tantos como el compilador permita. Los corchetes (`[` y `]`) indican que los argumentos no especificados son opcionales.

La siguiente es una forma general para declarar una función con un número variable de argumentos:

```
especificador_de_tipo_de_datos  nombre_de_función(
      especificador_de_tipo_de_datos  nombre_de_argumento1, ...
);
```

Observe que el nombre del primer argumento está seguido por los puntos suspensivos (`...`) que representan el resto de los argumentos no especificados.

Aquí, *Hora calendario* da la fecha y hora actuales con base en el calendario Gregoriano. *Hora local* representa el tiempo calendario en una zona horaria específica. *Horario de verano* es la hora local bajo las reglas del horario de verano.

En esta sección se presentan brevemente tres funciones de fecha y hora: `time()`, `localtime()` y `asctime()`.

La función `time()` devuelve la hora calendario.

SINTAXIS

La sintaxis de la función `time()` es

```
#include <time.h>
time_t time(time_t *tiempo);
```

Aquí, `time_t` es el tipo aritmético que se usa para representar el tiempo. `tiempo` es una variable de apuntador que apunta a una ubicación de memoria que puede contener la hora calendario que devuelve esta función. La función `time()` devuelve `-1` si la hora calendario no está disponible en la computadora.

La función `localtime()` devuelve la hora local convertida a partir de la hora calendario.

SINTAXIS

La sintaxis de la función `localtime()` es

```
#include <time.h>
struct tm *localtime(const time_t *tiempo);
```

Aquí, `tm` es una estructura que contiene los componentes de la hora calendario. `struct` es la palabra reservada para estructura, la cual es otro tipo de datos de C. (El concepto de estructura se presenta en la hora 19, "Estructuras de datos".) `tiempo` es una variable de apuntador que apunta a una ubicación de memoria que contiene la hora calendario que devuelve la función `time()`.

Para convertir la fecha y hora representadas por la estructura `tm`, puede llamar a la función `asctime()`.

SINTAXIS

La sintaxis de la función `asctime()` es

```
#include <time.h>
char *asctime(const struct tm *aptrtiempo);
```

Aquí, `aptrtiempo` es un apuntador que hace referencia a la estructura `tm` que devuelven las funciones de fecha y hora como `localtime()`. La función `asctime()` convierte en cadena de caracteres la fecha y hora representadas por `tm`.

Como se muestra en el listado 15.2, la instrucción de la línea 17 declara una variable de tipo `time_t` llamada `ahora`. La línea 20 almacena la hora calendario en la ubicación de memoria a la que hace referencia la variable `ahora`. Observe que el argumento que se pasa a la función `time()` debe ser el valor izquierdo de una variable; por lo tanto, se utiliza el operador de dirección (`&`) antepuesto a `ahora`. Luego, la expresión de la línea

Materiał chroniony prawem autorskim

Obtuve los siguientes resultados después de ejecutar el archivo `15L02.exe` del programa del listado 15.2:

SALIDA

```
Antes de llamar a la función FechaHora().
Dentro de FechaHora().
La fecha y hora actuales son: Sat Apr 05 11:50:10 1997

Después de llamar a la función FechaHora().
```

ANÁLISIS La finalidad del programa del listado 15.2 es mostrarle cómo declarar y llamar una función sin pasar argumentos. El programa imprime la fecha y hora actuales de su computadora llamando a la función `FechaHora()`, declarada en la línea 5. Debido a que no es necesario pasar ningún argumento a la función, en la declaración de `FechaHora()` se usa el tipo de datos `void` como prototipo.

Además se usa otra palabra reservada `void` antes del nombre de la función `FechaHora()` para indicar que la función tampoco devuelve ningún valor (vea la línea 5).

Las instrucciones de las líneas 9 y 11 imprimen mensajes antes y después de llamar a la función `FechaHora()` desde el interior de la función `main()`.

En la línea 10 se llama a la función mediante la instrucción `FechaHora();`. Observe que no se debe pasar ningún argumento a esta función, debido a que su prototipo es `void`.

La definición de `FechaHora()` está en las líneas 15 a 23; la función obtiene la hora calendario y la convierte a una cadena de caracteres llamando a varias funciones de la biblioteca de C: `time()`, `localtime()` y `asctime()`. Luego, la función `printf()` imprime en la pantalla la cadena de caracteres que contiene la fecha y hora actuales, mediante el especificador de formato `%s`. Como puede ver, los resultados muestran en mi pantalla que al momento de ejecutar el archivo `15L02.exe` la fecha y hora fueron las siguientes:

```
Sat Apr 05 11:50:10 1997
```

`time()`, `localtime()` y `asctime()` son funciones de fecha y hora que proporciona el lenguaje C. En la siguiente subsección veremos estas funciones. Tal vez haya notado que se incluye el archivo de encabezado `time.h` al principio del programa del listado 15.2 antes de usar estas funciones de tiempo.

Uso de time(), localtime() y asctime()

Hay un grupo de funciones de C que se conocen como *funciones de fecha y hora*. Las declaraciones de todas las funciones de fecha y hora están incluidas en el archivo de encabezado `time.h`. Estas funciones pueden dar tres tipos de fecha y hora:

- Hora calendario
- Hora local
- Horario de verano

Funciones sin argumentos

El primer caso es el de una función que no toma ningún argumento. Por ejemplo, la función de biblioteca de C `getchar()` no necesita argumentos. Se puede usar en un programa de la siguiente manera:

```
int c;
c = getchar();
```

Como puede ver, cuando se llama a la función en la segunda instrucción, se deja en blanco el espacio entre los paréntesis (`(` y `)`).

En C, la declaración de la función `getchar()` puede ser algo como lo siguiente:

```
int getchar(void);
```

Observe que en la declaración se utiliza la palabra reservada `void` para indicarle al compilador que esta función no necesita ningún argumento. El compilador emitirá un mensaje de error si se pasa algún argumento a `getchar()` más adelante en el programa, cuando se llame a la función.

Por lo tanto, para una función sin ningún argumento, se usa el tipo de datos `void` como el prototipo de la declaración de la función.

El programa del listado 15.2 muestra otro ejemplo del uso de `void` en la declaración de funciones.

TECLEE **LISTADO 15.2** Uso de `void` en la declaración de funciones

```
1:  /* 15L02.c: Funciones sin argumentos */
2:  #include <stdio.h>
3:  #include <time.h>
4:
5:  void FechaHora(void);
6:
7:  main()
8:  {
9:      printf("Antes de llamar a la función FechaHora().\n");
10:     FechaHora();
11:     printf("Después de llamar a la función FechaHora().\n");
12:     return 0;
13: }
14: /* Definición de FechaHora() */
15: void FechaHora(void)
16: {
17:     time_t ahora;
18:
19:     printf("Dentro de FechaHora().\n");
20:     time(&ahora);
21:     printf("La fecha y hora actuales son: %s\n",
22:         asctime(localtime(&ahora)));
23: }
```

Materiał chroniony prawem autorskim

Después de crear y ejecutar el archivo `15L01.exe` del programa del listado 15.1, se desplegaron los siguientes resultados en la pantalla:

SALIDA

```
Pasa a funcion_1  80 y 10.
Dentro de funcion_1.
funcion_1 devuelve 90.
Pasa a funcion_2  100.123456 y 10.123456.
Dentro de funcion_2.
funcion_2 devuelve 90.000000.
```

ANÁLISIS La finalidad del programa del listado 15.1 es mostrarle cómo declarar y definir funciones. La instrucción de la línea 4 es una declaración de función con un prototipo. La declaración hace referencia a `funcion_1`, la cual está definida más adelante en el listado 15.1. El tipo de retorno de `funcion_1` es `int`, y el prototipo incluye dos variables `int`, `x` y `y`.

En las líneas 5 a 9 se define la segunda función, `funcion_2`, antes de llamarla. Como puede ver, el tipo de retorno de `funcion_2` es `double`, y se pasan a la función dos variables `double`. Observe que los nombres de las dos variables son también `x` y `y`. No se preocupe porque `funcion_1` y `funcion_2` compartan los mismos nombres de argumentos. No existe conflicto debido a que estos argumentos están en diferentes bloques de función y los argumentos para funciones tienen un alcance de bloque.

Después, en la función `main()` definida en las líneas 11 a 23, se declaran e inicializan en las líneas 13 a 16 dos variables `int`, `x1` y `y1`, y dos variables `double`, `x2` y `y2`. La instrucción de la línea 18 muestra los valores de `x1` y `y1` que se pasan a `funcion_1`. La línea 19 llama a `función_1` y despliega el valor de retorno de `funcion_1`.

Asimismo, las líneas 20 y 21 imprimen los valores de `x2` y `y2` que se pasan a `funcion_2`, así como el valor que devuelve `funcion_2` después de llamar y ejecutar la función.

Las líneas 25 a 29 contienen la definición de `funcion_1`, la cual especifica que la función realiza una suma de dos variables enteras (vea la línea 28) e imprime la cadena `dentro de funcion_1` de la línea 27.

Prototipos de funciones

En las siguientes secciones estudiará tres casos referentes a argumentos que se pasan a funciones. En el primer caso, las funciones no toman argumentos; en el segundo, las funciones toman un número fijo de argumentos; y en el tercer caso, las funciones toman un número variable de argumentos.

LISTADO 15.1 continuación

```
7:     printf("Dentro de funcion_2.\n");
8:     return (x - y);
9:  }
10:
11: main()
12: {
13:    int x1 = 80;
14:    int y1 = 10;
15:    double x2 = 100.123456;
16:    double y2 = 10.123456;
17:
18:    printf("Pasa a  funcion_1  %d y %d.\n", x1, y1);
19:    printf("funcion_1 devuelve %d.\n", funcion_1(x1, y1));
20:    printf("Pasa a funcion_2  %f y %f.\n", x2, y2);
21:    printf("funcion_2 devuelve %f.\n", funcion_2(x2, y2));
22:    return 0;
23: }
24: /* definición de funcion_1()*/
25: int funcion_1(int x, int y)
26: {
27:    printf("Dentro de funcion_1.\n");
28:    return (x + y);
29: }
```

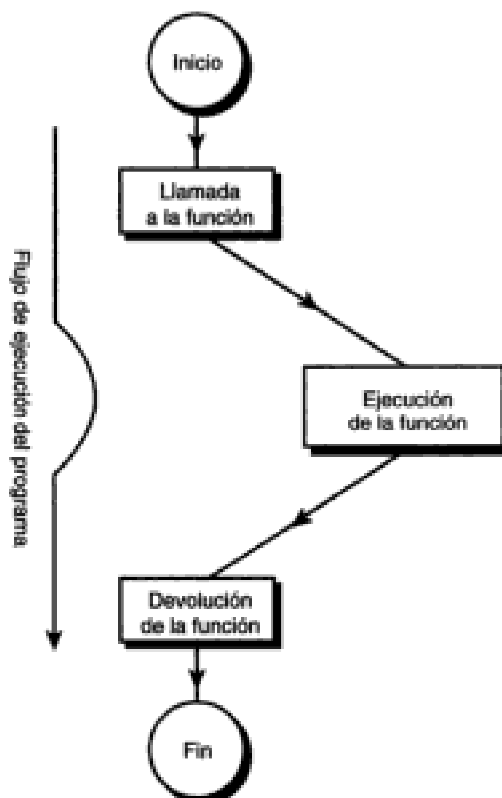


FIGURA 15.1
La ejecución del programa salta a una función invocada cuando se hace una llamada a dicha función.

Materiał chroniony prawem autorskim

Uso de prototipos

Antes de la creación del estándar ANSI, una declaración de función sólo incluía el tipo de retorno de la función. Con el estándar ANSI se pueden agregar a la declaración de la función el número y los tipos de los argumentos que se pasan a la función. El número y los tipos de un argumento se llaman *prototipo de la función*.

La forma general de una declaración de función, incluyendo su prototipo, es la siguiente:

```
especificador_de_tipo_de_datos nombre_de_función(
      especificador_de_tipo_de_datos nombre_de_argumento1,
      especificador_de_tipo_de_datos nombre_de_argumento2,
      especificador_de_tipo_de_datos nombre_de_argumento3,
      .
      .
      .
      especificador_de_tipo_de_datos nombre_de_argumentoN,
);
```

La finalidad de usar un prototipo de función es ayudarle al compilador a verificar si los tipos de datos de los argumentos que se pasan a la función coinciden con lo que la función espera. Si los tipos de datos no coinciden, el compilador emite un mensaje de error.

Aunque los nombres de argumentos, como `nombre_de_argumento1`, `nombre_de_argumento2`, etcétera, son opcionales, se recomienda incluirlos para que el compilador pueda identificar todas las faltas de correspondencia de los nombres de argumentos.

Cómo hacer llamadas a las funciones

Como se muestra en la figura 15.1, cuando se hace una llamada a una función, la ejecución del programa salta a la función y termina la tarea asignada a ésta. Entonces, la ejecución del programa continúa después de que regresa de la función que se llamó.

Una *llamada a una función* es una expresión que se puede usar como una sola instrucción o dentro de otras instrucciones.

El listado 15.1 proporciona un ejemplo de la declaración y definición de funciones, así como de la llamada a las funciones.

TECLEE **LISTADO 15.1** Cómo hacer llamadas a las funciones después de declararlas y definirlas

```
1:  /* 15L01.c: Cómo hacer llamadas a las funciones */
2:  #include <stdio.h>
3:
4:  int funcion_1(int x, int y);
5:  double funcion_2(double x, double y)
6:  {
```

continúa

Declaración de funciones

Como usted sabe, tiene que declarar o definir una variable antes de poder utilizarla. Esto también se aplica a las funciones. En C, debe declarar o definir una función antes de que pueda llamarla.

Declaración en comparación con definición

De acuerdo con el estándar ANSI, la *declaración* de una variable o función especifica la interpretación y atributos de un conjunto de identificadores. Por otra parte, la *definición* requiere que el compilador de C reserve espacio de almacenamiento para una variable o función nombrada por un identificador.

Una declaración de variable es una definición, pero una declaración de función no lo es. Una declaración de función hace referencia a una función que está definida en otra parte y especifica qué tipo de valor devuelve la función. Una definición de función define lo que hace la función, además de dar el número y tipo de argumentos que se pasan a la función.

Una declaración de función no es una definición de función. Si coloca una definición de función en su archivo fuente antes de llamar por primera vez a la función, no necesita declarar la función. En caso contrario, debe declarar la función antes de invocarla.

Por ejemplo, utilicé la función `printf()` en casi todos los programas de muestra del libro. Tuve que incluir en cada programa un archivo de encabezado, `stdio.h`, debido a que dicho archivo contiene la declaración de `printf()`, la cual le indica al compilador el tipo de devolución y el prototipo de la función. La definición de la función `printf()` se coloca en cualquier lugar. En C, la definición de esta función se guarda en un archivo de biblioteca que se invoca durante las etapas de enlace.

Especificación de los tipos de retorno

Se puede declarar una función para que devuelva cualquier tipo de datos, excepto un arreglo o una función. La instrucción `return` que se usa en una definición de función devuelve un solo valor cuyo tipo debe coincidir con el señalado en la declaración de la función.

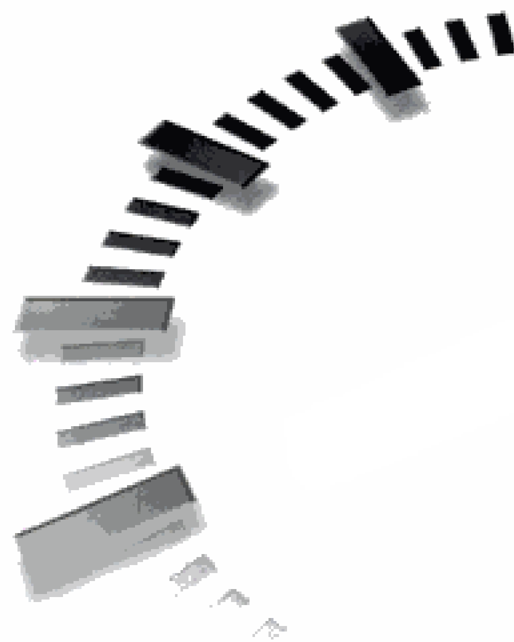
Si no se especifica ningún tipo de datos explícito para la función, el tipo de retorno de una función será `int` de manera predeterminada. Antes del nombre de la función se coloca un especificador de tipo de datos, como se muestra a continuación:

```
especificador_de_tipo_de_datos nombre_de_función();
```

Aquí, *`especificador_de_tipo_de_datos`* especifica el tipo de datos que debe regresar la función. *`nombre_de_función`* es el nombre de la función que debe seguir las reglas de nombrado de identificadores de C.

De hecho, esta forma de declaración representa la forma tradicional de declaración de funciones desde antes de la creación del estándar ANSI. Después de configurar el estándar ANSI, se agrega a la declaración de la función el prototipo de función.

Materiał chroniony prawem autorskim



HORA 15

Funciones

La forma sigue a la función.

—L. H. Sullivan

Tal vez haya notado en la hora 14, "Alcance y clases de almacenamiento", que siempre se da primero la definición de una función, antes de llamar a la función desde una función `main()`. De hecho, puede colocar la definición de una función en donde usted quiera, siempre y cuando lo haga antes del primer lugar en donde se llame a la función. Aprenderá muchas características de las funciones en los temas que trataremos en esta lección:

- Declaración de funciones
- Prototipos
- Valores que devuelven las funciones
- Argumentos para las funciones
- Programación estructurada

Además, en esta hora se presentan varias funciones y macros de la biblioteca de C, como `time()`, `localtime()`, `asctime()`, `va_start()`, `va_arg()` y `va_end()`.

Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

PARTE IV

Funciones y asignación dinámica de memoria

Hora

15 Funciones

16 Uso de apuntadores

17 Asignación de memoria

18 Tipos de datos y funciones especiales

Ejercicios

1. Escriba las declaraciones de las siguientes variables y del apuntador:
   - Una variable `int` con alcance de bloque y almacenamiento temporal
   - Una variable de constante de carácter con alcance de bloque
   - Una variable local `float` con almacenamiento permanente
   - Una variable `int` de registro
   - Un apuntador `char` inicializado con un carácter nulo
2. Reescriba el programa del listado 14.2. Esta vez pase la variable `x` de tipo `int` y la variable `y` de tipo `float` como argumentos a `funcion_1()`. ¿Qué obtiene en la pantalla después de ejecutar el programa?
3. Compile y ejecute el siguiente programa. ¿Qué obtiene en la pantalla y por qué?

```
#include <stdio.h>
int main()
{
    int i;

    for (i=0; i<5; i++){
        int x = 0;
        static int y = 0;
        printf("x=%d, y=%d\n", x++, y++);
    }
    return 0;
}
```

4. Reescriba la función `suma_dos()` del listado 14.3 para imprimir el resultado anterior de la suma, así como el valor del contador.

Taller

Para ayudarle a consolidar la comprensión de esta lección, le recomiendo responder el cuestionario y terminar los ejercicios del taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. Dado el siguiente fragmento de código, ¿cuáles variables son globales, y cuáles son locales con alcance de bloque?

   ```
   int x = 0;
   float y = 0.0;
   int miFuncion()
   {
      int i, j;
      float y;
      . . .
      {
         int x, y;
         . . .
      }
      . . .
   }
   ```

2. ¿Cómo sabe el compilador qué variable utilizar cuando se definen dos variables con el mismo nombre?
3. Identifique la clase de almacenamiento de cada declaración del siguiente fragmento de código:

   ```
   int i = 0;
   static int x;
   extern float y;
   int miFuncion()
   {
      int i, j;
      extern float z;
      register long s;
      static int indice;
      const char cadena[] = "Mensaje de advertencia.";
      . . .
   }
   ```

4. Dada la siguiente declaración:

   ```
   const char cadena[] = "El especificador const";
   ```

   ¿es válida la instrucción `cadena[9] = '-';`?

Materiał chroniony prawem autorskim

- Para asegurarse de que los valores guardados por una variable no cambien, puede declarar la variable con el modificador `const`.
- Si desea permitir que el compilador sepa que el valor de una variable se puede cambiar sin una instrucción explícita de asignación, declare la variable con el modificador `volatile` a fin de que el compilador desactive las optimizaciones sobre expresiones que comprendan a la variable.

En la siguiente lección aprenderá acerca de las declaraciones y prototipos de funciones de C.

Preguntas y respuestas

P ¿Puede una variable global ser ocultada por una variable local con alcance de bloque?

R Sí. Si una variable local comparte el mismo nombre con una variable global, esta última puede ser ocultada por la variable local para el alcance del bloque dentro del cual está definida la variable local. Sin embargo, no es posible ver a la variable local fuera del bloque, por lo que la variable global se hace visible de nuevo.

P ¿Por qué necesita el especificador `static`?

R En muchos casos, el valor de una variable es necesario, incluso si se ha salido del alcance del bloque en el que está declarada la variable. De manera predeterminada, una variable con alcance de bloque tiene un almacenamiento de memoria temporal, es decir, el tiempo de vida de la variable comienza cuando se ejecuta el bloque y se declara la variable, y termina cuando finaliza la ejecución de ese bloque. Por lo tanto, para declarar una variable con duración permanente, tiene que usar el especificador `static` para indicarle al compilador que la ubicación de memoria de la variable y el valor ahí almacenado se deben conservar después de la ejecución del bloque.

P ¿Puede el uso del especificador `register` garantizar la mejora en el rendimiento de un programa?

R En realidad no. Declarar una variable con el especificador `register` sólo sugiere al compilador que debe almacenar la variable en un registro. Pero no hay garantía de que esto suceda. El compilador puede ignorar la solicitud con base en la disponibilidad de registros o en otras restricciones.

P Cuando declara una variable con el especificador `extern`, ¿define la variable o hace referencia a una variable global en otra parte?

R Cuando se declara una variable con el especificador `extern`, el compilador considera la declaración de la variable como una referencia, más que como una definición. Por lo tanto, el compilador buscará en otra parte la variable global a la que hace referencia la variable con `extern`.

```
    volatile char car_teclado;  /* una variable volatile */
    .
    .
    .
}
```

Resumen

En esta lección aprendió los siguientes conceptos importantes acerca de los alcances y las clases de almacenamiento de C:

- Una variable que se declara dentro de un bloque tiene alcance de bloque. A dicha variable también se le llama variable local y sólo es visible dentro del bloque.
- Una etiqueta de `goto` tiene un alcance de función, lo que significa que es visible a través del bloque completo de la función dentro de la que se coloca la etiqueta. Dos etiquetas `goto` no pueden compartir el mismo nombre dentro de un bloque de función.
- Una variable que se declara con el especificador `static` fuera de una función tiene alcance de archivo, lo que quiere decir que es visible a lo largo de todo el archivo fuente en el que está declarada.
- Se dice que una variable que se declara fuera de una función tiene un alcance de programa. Dicha variable también se denomina variable global. Una variable global es visible en todos los archivos fuente que conforman un programa ejecutable.
- Una variable con alcance de bloque tiene la visibilidad más limitada. Por otra parte, una variable con alcance de programa es la más visible, y puede verse en todos los archivos, funciones y otros bloques que conforman el programa.
- La clase de almacenamiento de una variable se refiere a la combinación de sus regiones espacial y temporal (es decir, a su alcance y duración).
- Una variable con alcance de bloque tiene una duración `auto` de manera predeterminada, y su almacenamiento de memoria es temporal.
- Una variable declarada con el especificador `static` tiene un almacenamiento de memoria permanente, incluso después de que se llamó a la función en la que está declarada la variable y se salió del alcance de la función.
- Una variable declarada con el especificador `register` podría almacenarse en un registro para acelerar el rendimiento de un programa; sin embargo, el compilador puede ignorar el especificador si no hay un registro disponible o si se aplican otras restricciones.
- También puede referirse a una variable global definida en otra parte, mediante el especificador `extern` en el archivo fuente actual.

Materiał chroniony prawem autorskim

Por ejemplo, la siguiente expresión le indica al compilador que `pi` es una variable cuyo valor no debe cambiar:

```
const double pi = 3.141593;
```

De la misma manera, el valor del arreglo de caracteres `cadena` declarado en la siguiente instrucción tampoco debe cambiar:

```
const char cadena[] = "Una constante de cadena";
```

Por lo tanto, no es válido hacer algo como lo siguiente:

```
cadena[0] = 'a';  /* No se permite aquí. */
```

Además, puede declarar una variable de apuntador con el modificador `const` a fin de que no se pueda modificar el objeto al que señala. Por ejemplo, considere la siguiente declaración de apuntador con el modificador `const`:

```
char const *aptr_cadena = "Una constante de cadena";
```

Después de la inicialización, no puede modificar el contenido de la cadena a la que apunta `aptr_cadena`. Por ejemplo, no se permite la siguiente instrucción:

```
*aptr_cadena = 'a';     /* No se permite aquí. */
```

Sin embargo, el propio apuntador `aptr_cadena` puede asignarse a una dirección diferente de una cadena que esté declarada con `char const`.

El modificador `volatile`

A veces necesita declarar una variable cuyo valor se pueda cambiar en su programa sin ninguna instrucción explícita de asignación. Esto es cierto, especialmente cuando trata de manera directa con el hardware. Por ejemplo, podría declarar una variable global que contenga caracteres introducidos por el usuario. La dirección de la variable se pasa a un registro de dispositivo que acepta caracteres desde el teclado. Sin embargo, cuando el compilador de C optimiza su programa en forma automática, no tiene la intención de actualizar el valor que contiene la variable, a menos que ésta esté en la parte izquierda de un operador de asignación (=). En otras palabras, es probable que el valor de la variable no cambie aun cuando el usuario introduzca caracteres desde el teclado.

Para solicitar al compilador que desactive ciertas optimizaciones sobre una variable, puede declararla con el modificador `volatile`. Por ejemplo, en el siguiente fragmento de código, la variable `car_teclado` declarada con el modificador `volatile`, indica al compilador que no optimice ninguna expresión de la variable debido a que el valor guardado en ella podría cambiar sin la ejecución de ninguna expresión explícita de asignación:

```
void lee_teclado()
{
```

La solución es utilizar el especificador `extern` que proporciona el lenguaje C para aludir a una variable global definida en cualquier otra parte. En este caso, usted declara una variable global en el archivo A, y luego declara de nuevo la variable utilizando el especificador `extern` en el archivo B. Ésta no es una declaración aparte, sino que especifica la declaración original del archivo A.

Por ejemplo, suponga que tiene dos variables globales de tipo `int`, `y` y `z`, que están definidas en un archivo, y luego, en otro archivo, podría tener las siguientes declaraciones:

```
int x = 0;          /* una variable global */
extern int y;       /* una alusión a una variable global y */
int main()
{
   extern int z;  /* una alusión a una variable global z */
   int i;          /* una variable local */
   .
   .
   .
   return 0;
}
```

Como puede ver, hay dos variables enteras, `y` y `z`, que están declaradas con el especificador `extern`, tanto fuera como dentro de la función `main()`, respectivamente. Cuando el compilador ve las dos declaraciones, sabe que en realidad son alusiones a las variables globales `y` y `z` definidas en alguna otra parte.

Para hacer portable su programa entre diferentes plataformas de cómputo, puede aplicar en él las siguientes reglas al declarar o aludir a variables globales:

- Al declarar una variable global, puede ignorar el especificador `extern`, pero tiene que incluir un inicializador.
- Debe emplear el especificador `extern` (sin un inicializador) cuando aluda a una variable global definida en otra parte.

Los modificadores de clase de almacenamiento

Además de los cuatro especificadores de clase de almacenamiento presentados en las secciones anteriores, C también le proporciona dos modificadores de clase de almacenamiento (o *calificadores*, como a veces se les llama) que puede usar para indicarle al compilador de C cómo podría acceder a las variables.

El modificador `const`

Si declara una variable con el modificador `const`, el contenido de la misma no se puede modificar una vez inicializada.

Material chroniony prawem autorskim

El especificador `register`

La palabra *register* se tomó prestada de la terminología de hardware computacional. Cada computadora tiene un cierto número de registros para contener datos y realizar operaciones de cálculo aritmético o lógico. Debido a que los registros están ubicados dentro del chip de la CPU (unidad central de procesamiento), resulta mucho más rápido acceder a un registro que a una ubicación de memoria que reside fuera del chip. Por lo tanto, almacenar variables en registros podría ayudar a acelerar su programa.

El lenguaje C le proporciona el especificador `register`. Usted puede aplicar este especificador a variables cuando crea que sea necesario colocarlas dentro de los registros de la computadora.

Sin embargo, el especificador `register` sólo hace una sugerencia al compilador. En otras palabras, no se garantiza que una variable especificada con la palabra reservada `register` se almacene en un registro. El compilador puede ignorar la sugerencia si no hay un registro disponible, o si se aplica alguna otra restricción.

No es válido tomar la dirección de una variable que esté declarada con el especificador `register`, ya que se pretende que la variable se almacene en un registro y no en memoria. Un registro de la CPU no tiene una dirección de memoria a la que usted pueda acceder.

En el siguiente fragmento de código se declara una variable entera `i` con el especificador `register`:

```
int main()
{
   /* alcance de bloque con el especificador register */
   register int i;
   . . .
   for (i=0; i<MAX_NUM; i++){
      /* algunas instrucciones */
   }
   . . .
   return 0;
}
```

La declaración de `i` sugiere que el compilador almacene la variable en un registro. Debido a que `i` se usa de manera intensiva en el ciclo `for`, almacenarla en un registro podría aumentar la velocidad de este código.

El especificador `extern`

Como vimos antes en esta hora, en la sección titulada "Alcance de programa", una variable con alcance de programa es visible a través de todos los archivos fuente que conforman un programa ejecutable. A una variable de este tipo también se le conoce como *variable global*.

He aquí la pregunta: ¿Cómo puede ser vista en el archivo B una variable declarada en el archivo A? En otras palabras, ¿cómo sabe el compilador que la variable utilizada en el archivo B es en realidad la misma variable declarada en el archivo A?

El alcance de archivo y la jerarquía de los alcances

En la primera parte de esta hora mencioné tres de los cuatro tipos de alcance: de bloque, de función y de programa. Ahora es el momento de presentar el cuarto alcance: el *alcance de archivo.*

En C, se dice que una variable global declarada con el especificador `static` tiene un alcance de archivo. Una variable con alcance de archivo es visible desde el punto de su declaración hasta el final del archivo. Aquí, el archivo se refiere al archivo de programa que contiene el código fuente. La mayoría de los programas grandes consta de varios archivos de programa.

El siguiente fragmento de código fuente muestra variables con alcance de archivo:

```
int x = 0;               /* alcance de programa */
static int y = 0;       /* alcance de archivo */
static float z = 0.0;   /* alcance de archivo */
int main()
{
   int i;   /* alcance de bloque */
   .
   .
   .
   return 0;
}
```

Aquí, la variable `y` de tipo `int` y la variable `z` de tipo `float` tienen alcance de archivo.

La figura 14.1 muestra la jerarquía de los cuatro alcances. Como puede ver, una variable con alcance de bloque es la más limitada y no es visible fuera del bloque en el que está declarada. Por otra parte, una variable con alcance de programa es visible dentro de todos los archivos, funciones y otros bloques que conforman un programa.

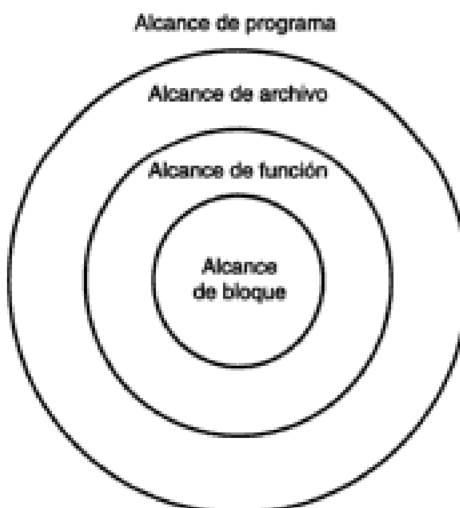


FIGURA 14.1
La jerarquía de los cuatro alcances.

Materiał chroniony prawem autorskim

Se desplegaron los siguientes resultados en la pantalla después de ejecutar el archivo 14L03.exe:

SALIDA

```
Llamada 1 a la función,
la suma de 0 y 5 es 5.

Llamada 2 a la función,
la suma de 1 y 4 es 5.

Llamada 3 a la función,
la suma de 2 y 3 es 5.

Llamada 4 a la función,
la suma de 3 y 2 es 5.

Llamada 5 a la función,
la suma de 4 y 1 es 5.
```

ANÁLISIS La finalidad del programa del listado 14.3 es llamar una función para sumar dos enteros, e imprimir después en la pantalla los resultados que devuelve la función. La función se llama varias veces. Se establece un contador para determinar cuántas veces se llama a la función.

Esta función, llamada `suma_dos ()`, se declara en las líneas 4 a 10. Hay dos argumentos de tipo `int`, `x` y `y`, que se pasan a la función y cuya suma se devuelve en la línea 9. Observe que en la línea 6 hay una variable entera, `contador`, que se declara con el especificador `static`. Los valores que almacena `contador` se conservan debido a que la duración de la variable es permanente. En otras palabras, aunque el alcance de `contador` esté dentro del bloque de la función `suma_dos()`, la ubicación de memoria de `contador` y el valor guardado en ella no se modifican después de que se llama a la función `suma_dos()`, y el control de la ejecución regresa a la función `main()`. Observe que la inicialización de `contador` a `1` sólo se lleva a cabo la primera vez que se llama a `suma_dos()`; después de eso, ésta conserva cada valor previo cada vez que se llama a la función.

Por lo tanto, la variable `contador` se usa como un contador para llevar un registro del número de llamadas que recibe la función `suma_dos()`. De hecho, la función `printf()` de la línea 8 imprime el valor guardado por la variable `contador` cada vez que se llama a la función `suma_dos()`. Además, `contador` se incrementa en uno cada vez que se ejecuta la función `printf()`.

El ciclo `for`, declarado en las líneas 16 a 18 dentro de la función `main()`, llama a la función `suma_dos ()` en cinco ocasiones. Los valores de las dos variables enteras, `i` y `j`, se pasan a la función `suma_dos()` en donde se suman. Después, la llamada a la función `printf()` de las líneas 17 y 18 despliega en la pantalla el valor de retorno de la función `suma_dos()`.

Puede ver en los resultados que, en efecto, el valor guardado por `contador` se incrementa en uno cada vez que se llama a la función `suma_dos()`, y se conserva después de salir de la función, debido a que la variable entera `contador` está declarada con `static`. Puede ver que `contador` solamente se inicializa a 1 una vez, cuando se llama por vez primera a la función `suma_dos()`.

El especificador `static`

Por otra parte, el especificador `static` se puede aplicar a variables con alcance de bloque, o con alcance de programa. Una variable que está dentro de una función tiene una duración permanente cuando se declara con el especificador `static`. En otras palabras, la ubicación de memoria asignada a la variable no se destruye cuando se sale del alcance de la variable, sino que el valor de la variable se mantiene fuera del alcance. Si la ejecución regresa alguna vez al alcance de la variable, aún estará ahí el último valor almacenado en ella.

Por ejemplo, en el siguiente fragmento de código:

```
int main()
{
   int i;          /* alcance de bloque y duración temporal */
   static int j;  /* alcance de bloque y duración permanente */
   .
   .
   return 0;
}
```

la variable entera `i` tiene una duración temporal (auto) de manera predeterminada. Pero la otra variable entera, `j`, tiene una duración permanente debido al especificador de clase de almacenamiento `static`.

El programa del listado 14.3 muestra el efecto que tiene el especificador `static` sobre las variables.

TECLEE **LISTADO 14.3** Uso del especificador `static`

```
1:  /* 14L03.c: Uso del especificador static */
2:  #include <stdio.h>
3:  /* la función suma_dos */
4:  int suma_dos (int x, int y)
5:  {
6:     static int contador = 1;
7:
8:     printf("Llamada %d a la función,\n", contador++);
9:     return (x + y);
10: }
11: /* la función main */
12: main()
13: {
14:    int i, j;
15:
16:    for (i=0, j=5; i<5; i++, j--)
17:       printf("la suma de %d y %d es %d.\n\n",
18:              i, j, suma_dos (i, j));
19:    return 0;
20: }
```

Materiał chroniony prawem autorskim

valor de la variable local `x` con alcance de bloque, mientras que el valor de `y` sigue siendo el de la variable global `y`.

En las líneas 19 a 23 hay un bloque anidado, dentro del cual se declara e inicializa la variable `y` de tipo `double` con alcance de bloque. Al igual que la variable `x` que está dentro del bloque `main()`, esta variable `y`, que está dentro del bloque anidado, sustituye a la variable global `y`. La instrucción de la línea 22 despliega en la pantalla los valores de las variables locales `x` y `y`.

Debido a que una variable global es visible entre los diferentes archivos fuente de un programa, su uso incrementa la complejidad de su programa, lo que a su vez lo hace más difícil de depurar y de darle mantenimiento. Por lo general, no se recomienda que declare y use variables globales, a menos que sea absolutamente necesario. Por ejemplo, puede declarar una variable global cuyo valor lo usen (pero nunca lo cambien) diversas subrutinas de su programa. (En la hora 23, "Compilación: el preprocesador de C", aprenderá a usar la directiva `#define` para definir constantes que se utilicen en muchas partes de un programa.)

Antes de presentar el alcance de archivo, permítame hablar primero acerca de los especificadores de clases de almacenamiento.

Los especificadores de clases de almacenamiento

En C, la *clase de almacenamiento* de una variable se refiere a la combinación de sus regiones espacial y temporal.

Ya aprendió acerca del alcance, el cual especifica la región espacial de una variable. Concentrémonos ahora en la *duración*, la cual indica la región temporal de una variable.

Existen cuatro especificadores y dos modificadores que se pueden emplear para indicar la duración de una variable. Éstos se presentan en las siguientes secciones.

El especificador `auto`

El especificador `auto` se usa para indicar que la ubicación de memoria de una variable es temporal. En otras palabras, el espacio de memoria reservado para una variable se puede borrar o reubicar cuando la variable esté fuera de su alcance.

Sólo las variables que están dentro del alcance de bloque se pueden declarar con el especificador `auto`. Sin embargo, en la práctica rara vez se usa la palabra clave `auto`, debido a que la duración de una variable con alcance de bloque es temporal de manera predeterminada.

LISTADO 14.2 continuación

```
6:
7:  void funcion_1()
8:  {
9:      printf("Desde la funcion_1:\n  x=%d, y=%f\n", x, y);
10: }
11:
12: main()
13: {
14:     int x = 4321;   /* alcance de bloque 1*/
15:
16:     funcion_1();
17:     printf("Dentro del bloque main:\n  x=%d, y=%f\n", x, y);
18:     /* un bloque anidado */
19:     {
20:        double y = 7.654321;  /* alcance de bloque 2 */
21:        funcion_1();
22:        printf("Dentro del bloque anidado:\n  x=%d, y=%f\n", x, y);
23:     }
24:     return 0;
25: }
```

Después de crear y ejecutar el archivo `14L02.exe` en mi máquina, se desplegaron los siguientes resultados en la pantalla:

SALIDA

```
Desde la funcion_1:
  x=1234, y=1.234567
Dentro del bloque main:
  x=4321, y=1.234567
Desde la función_1:
  x=1234, y=1.234567
Dentro del bloque anidado:
  x=4321, y=7.654321
```

ANÁLISIS Como puede ver en el listado 14.2, en las líneas 4 y 5 se declaran dos variables globales, `x` y `y`, con un alcance de programa.

En las líneas 7 a 10 se declara la función denominada `funcion_1()`. (En la siguiente hora verá más detalles acerca de la declaración y prototipos de funciones.) La `funcion_1()` contiene sólo una instrucción; ésta imprime los valores que contienen tanto `x` como `y`. Debido a que no se hace una declaración de variables para `x` o `y` dentro del bloque de la función, se usan los valores de las variables globales `x` y `y` para la instrucción que está dentro de la función. Para demostrar esto, se llama dos veces a `funcion_1()` en las líneas 16 y 21, respectivamente, desde dos bloques anidados. Los resultados muestran que los valores de las variables globales `x` y `y` se pasan a la función `printf()` que está dentro del cuerpo de la función `funcion_1()`.

Después, la línea 14 declara otra variable entera, `x`, con un alcance de bloque, la cual puede reemplazar a la variable global `x` que está dentro del bloque de la función `main()`. El resultado de la instrucción de la línea 17 muestra que el valor de `x` corresponde al

Materiał chroniony prawem autorskim

```
   .
   .
   .
   goto  inicio;  /* la instrucción goto */
   .
   .
   .
   return 0;
}
```

Aquí, la etiqueta `inicio` es visible desde el principio hasta el final de la función `main()`. Por lo tanto, no debe haber más de una etiqueta con el mismo nombre dentro de la función `main()`.

Alcance de programa

Se dice que una variable tiene *alcance de programa* cuando se declara fuera de una función. Por ejemplo, observe el siguiente código:

```
int x = 0;          /* alcance de programa */
float y = 0.0;      /* alcance de programa */
int main()
{
   int i;   /* alcance de bloque */
   .
   .
   .
   return 0;
}
```

Aquí, la variable `x` de tipo `int` y la variable `y` de tipo `float` tienen un alcance de programa.

Las variables con alcance de programa también se conocen como *variables globales*, las cuales son visibles entre todos los archivos fuente que conforman un programa ejecutable. Observe que se declara una variable global fuera de una función.

El programa del listado 14.2 muestra la relación entre variables con alcance de programa y variables con alcance de bloque.

TECLEE

LISTADO 14.2 La relación entre alcance de programa y alcance de bloque

```
1:  /* 14L02.c: Alcance de programa en comparación con alcance de bloque */
2:  #include <stdio.h>
3:
4:  int x = 1234;          /* alcance de programa */
5:  double y = 1.234567;  /* alcance de programa */
```

continúa

```
i= 3, j= 7
i= 4, j= 6
i= 5, j= 5
i= 6, j= 4
i= 7, j= 3
i= 8, j= 2
i= 9, j= 1
i=10, j= 0
Dentro del bloque externo: i=32
```

ANÁLISIS La finalidad del programa del listado 14.1 es mostrarle los diferentes alcances de variables de bloques anidados. Como puede ver, en el listado hay dos bloques anidados. La variable entera `i` que se declara en la línea 6 es visible dentro del bloque externo encerrado entre las llaves (`{` y `}`) de las líneas 5 y 19. Otras dos variables enteras, `i` y `j`, se declaran en la línea 11 y son visibles sólo dentro del bloque interno que está en las líneas 10 a 16.

Aunque la variable entera `i` del bloque externo tiene el mismo nombre que una de las variables enteras del bloque interno, no se pueden acceder ambas variables al mismo tiempo debido a sus diferentes alcances.

Para demostrar eso, la línea 8 imprime por primera vez el valor contenido por `i` (`32`) dentro del bloque externo. Luego, el ciclo `for` de las líneas 14 y 15 despliega 10 pares de valores asignados a las variables `i` y `j` que están dentro del bloque interno. En este punto, no hay ningún indicio de que la variable entera `i` del bloque externo tenga algún efecto sobre la del bloque interno. Cuando se sale del bloque interno, sus variables ya no son accesibles. En otras palabras, cualquier intento de acceder a `j` desde el bloque externo sería ilegal.

Por último, la instrucción de la línea 17 imprime de nuevo el valor de la variable `i` del bloque externo para determinar si cambió el valor debido a la variable entera `i` del bloque interno. Los resultados muestran que estas dos variables se ocultan una de la otra, y no ocurre ningún conflicto.

Alcance de función

El *alcance de función* indica que una variable está activa y es visible desde el principio hasta el final de una función

En C, sólo una etiqueta de `goto` tiene alcance de función. Por ejemplo, la etiqueta `inicio` de `goto` que se muestra en el siguiente código tiene un alcance de función:

```
int main()
{
   int i;   /* alcance de bloque */
   .
   .
   .
   inicio:   /* Una etiqueta de goto tiene alcance de función */
```

Materiał chroniony prawem autorskim

```
   .
   .
   .
   return 0;
}
```

Por lo regular, una variable con alcance de bloque se conoce como *variable local*. Observe que las variables locales se deben declarar al principio del bloque, antes que otras instrucciones.

Alcance de bloque anidado

También puede declarar variables dentro de un bloque anidado. Cuando una variable que se declara en el bloque externo tiene el mismo nombre que una de las variables del bloque interno, la variable que está dentro del bloque interno oculta a la que está dentro del bloque externo. Esto se cumple para el alcance del bloque interno.

El listado 14.1 muestra un ejemplo de alcances de variables en bloques anidados.

TECLEE **LISTADO 14.1** Impresión de variables con diferentes niveles de alcance

```
1:  /* 14L01.c: Alcances en bloques anidados */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i = 32;   /* alcance de bloque 1*/
7:
8:     printf("Dentro del bloque externo: i=%d\n", i);
9:
10:    {     /* principio del bloque interno */
11:      int i, j;    /* alcance de bloque 2, int i oculta a int i externo */
12:
13:      printf("Dentro del bloque interno:\n");
14:      for (i=0, j=10; i<=10; i++, j--)
15:          printf("i=%2d, j=%2d\n", i, j);
16:    }   /* final del bloque interno */
17:    printf("Dentro del bloque externo:  i=%d\n", i);
18:    return 0;
19: }
```

Después de crear y ejecutar el archivo `14L01.exe` del programa del listado 14.1, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
Dentro del bloque externo: i=32
Dentro del bloque interno:
i= 0, j=10
i= 1, j= 9
i= 2, j= 8
```

- El especificador `auto`
- El especificador `static`
- El especificador `register`
- El especificador `extern`
- El modificador `const`
- El modificador `volatile`

Cómo ocultar datos

Para resolver en la práctica un problema complejo, el programador por lo regular divide el problema en piezas más pequeñas y maneja cada pieza del problema escribiendo una o dos funciones (o rutinas). Luego conjunta todas las funciones para formar un programa completo que pueda emplear para resolver el problema complejo.

En el programa completo podría haber variables que tengan que ser compartidas por todas las funciones. Por otra parte, el uso de algunas otras variables podría estar limitado a sólo ciertas funciones. Es decir, la visibilidad de dichas variables es limitada, y los valores que tienen asignados están ocultos a muchas funciones.

Es muy útil limitar el alcance de las variables cuando varios programadores están trabajando sobre diferentes piezas del mismo programa. Si limitan el alcance de sus variables a sus piezas de código, no tienen que preocuparse por conflictos con variables del mismo nombre utilizadas por otros en partes distintas del programa.

En C, usted puede declarar una variable e indicar su nivel de visibilidad designando su alcance. De esta manera, sólo se puede acceder a las variables con alcance local dentro del bloque en el que se declaran.

Las siguientes secciones le enseñan cómo declarar variables con diferentes alcances.

Alcance de bloque

En esta sección, un *bloque* se refiere a todo conjunto de instrucciones encerradas entre llaves (`{` y `}`). Una variable declarada dentro de un bloque tiene un *alcance de bloque*. Así, la variable está activa y accesible desde el punto de su declaración hasta el final del bloque. En ocasiones, el alcance de bloque se conoce también como *alcance local*.

Por ejemplo, la variable `i` declarada dentro del bloque de la siguiente función `main` tiene un alcance de bloque:

```
int main()
{
   int i;   /* alcance de bloque */
```

Materiał chroniony prawem autorskim

Hora 14

Alcance y clases de almacenamiento

Nadie posee nada, y todo lo que cualquiera tiene es el uso de sus presuntas posesiones.

—P. Wylie

En las horas anteriores aprendió cómo declarar variables de diferentes tipos de datos, así como a inicializar y utilizar dichas variables. Se ha asumido que puede acceder a las variables desde cualquier parte. Ahora, la pregunta es: ¿Puede declarar variables que sean accesibles sólo a ciertas porciones de un programa? En esta lección aprenderá acerca del alcance y de las clases de almacenamiento de datos de C. Los temas principales que trata esta lección son los siguientes:

- Alcance de bloque
- Alcance de función
- Alcance de archivo
- Alcance de programa

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Qué instrucciones son legales en la siguiente lista?
   - `char cadena1[5] = "Texas";`
   - `char cadena2[] = "Una cadena de caracteres";`
   - `char cadena3[2] = "A";`
   - `char cadena4[2] = "TX";`
2. Dada una variable de apuntador `aptr_ch` de tipo `char`, ¿son legales las siguientes instrucciones?
   - `*aptr_ch = 'a';`
   - `aptr_ch = "Una cadena de caracteres";`
   - `aptr_ch = 'x';`
   - `*aptr_ch = "Ésta es la pregunta 2.";`
3. ¿Puede la función `puts()` imprimir el carácter nulo en un arreglo de caracteres?
4. ¿Qué especificador de formato utiliza con la función `scanf()` para leer una cadena, y cuál para leer un número de punto flotante?

Ejercicios

1. Dado un arreglo de caracteres en la siguiente instrucción,

   `char cadena1[] = "Éste es el ejercicio 1.";`

   escriba un programa que copie la cadena `cadena1` a otro arreglo, llamado `cadena2`.
2. Escriba un programa que mida la longitud de una cadena evaluando uno a uno los elementos de un arreglo de caracteres hasta encontrar el carácter nulo. Para probar que obtuvo el resultado correcto, puede usar la función `strlen()` para medir de nuevo la misma cadena.
3. Reescriba el programa del listado 13.4. Esta vez convierta todos los caracteres que estén en mayúsculas a su contraparte en minúsculas.
4. Escriba un programa que use la función `scanf()` para leer dos enteros introducidos por el usuario, sume los dos enteros y luego imprima el resultado en la pantalla.

- La función `puts()` envía todos los caracteres de una cadena a `stdout`, excepto el nulo, y agrega un carácter de línea nueva en la salida.
- Puede leer diferentes elementos de datos con la función `scanf()`, utilizando diferentes especificadores de formato.

En la siguiente lección aprenderá acerca de los conceptos de alcance y almacenamiento en C.

Preguntas y respuestas

P ¿Qué es una cadena? ¿Cómo sabe su longitud?

R En C, una cadena se almacena en un arreglo de caracteres y se termina con un carácter nulo (`'\0'`). El carácter nulo indica a las funciones de cadena (como `puts()` y `strcpy()`) que han llegado al final de la cadena.

La función `strlen()`se puede emplear para medir la longitud de una cadena. Si tiene éxito, la función `strlen()` devuelve el número total de bytes que ocupa la cadena; sin embargo, no se cuenta el carácter nulo de la cadena.

P ¿Cuáles son las principales diferencias entre una constante de cadena y una constante de carácter?

R Una constante de cadena es una serie de caracteres encerrados entre comillas dobles, mientras que una constante de carácter es un solo carácter entre comillas sencillas. El compilador agregará un carácter nulo a la cadena cuando ésta se emplee para inicializar un arreglo. Por lo tanto, se debe reservar un byte adicional para el carácter nulo. Por otra parte, una constante de carácter ocupa sólo un byte de memoria y no se almacena en un arreglo.

P ¿Guarda la función `gets()` el carácter de línea nueva del flujo de entrada estándar?

R No. La función `gets()` sigue leyendo caracteres del flujo de entrada estándar hasta que encuentra un carácter de línea nueva o un fin de archivo. En lugar de guardar el carácter de línea nueva, la función `gets()` agrega un carácter nulo a la cadena y la almacena en el arreglo al que hace referencia el argumento de la función `gets()`.

P ¿Qué tipos de datos puede leer la función `scanf()`?

R Dependiendo de los especificadores de formato del estilo `printf()` que pase a la función, `scanf()` puede leer varios tipos de datos, como una serie de caracteres, enteros o números de punto flotante. A diferencia de `gets()`, `scanf()` detiene la lectura del elemento de entrada actual (y pasa al siguiente elemento, si lo hay) cuando encuentra un carácter de espacio, de línea nueva, de tabulador, de tabulador vertical o de avance de página.

Materiał chroniony prawem autorskim

Después, la función `scanf()` de la línea 11 lee dos enteros que introduce el usuario y los guarda en las ubicaciones de memoria reservadas para las variables enteras `x` y `y`. El operador de dirección se utiliza para obtener las direcciones de memoria de las variables. La instrucción de la línea 13 lee un número de punto flotante y lo guarda en `z`. Observe que se utilizan los especificadores de formato, `%d` y `%f`, para especificar los formatos adecuados para los números introducidos en las líneas 11 y 13.

La línea 15 usa la función `scanf()` y el especificador de formato `%s` para leer una serie de caracteres que introduce el usuario, y después guarda los caracteres (más un carácter nulo como terminador) dentro del arreglo al que apunta `cadena`. Aquí no se usa el operador de dirección, ya que la misma `cadena` apunta a la dirección de inicio del arreglo.

Para demostrar que la función `scanf()` lee todos los números y caracteres que introdujo el usuario, la función `printf()` de la línea 17 despliega en la pantalla los contenidos de `x`, `y`, `z` y `cadena`. Los resultados muestran que `scanf()` sí hizo su trabajo.

Algo que debe tener presente es que la función `scanf()` no comienza a leer la entrada hasta que se presiona la tecla Entrar. Los datos que se introducen desde el teclado se colocan en un búfer de entrada. Cuando se oprime Entrar, la función `scanf()` busca su entrada en el búfer. Aprenderá más acerca de la entrada y salida con búferes en la hora 21, "Lectura y escritura de archivos".

Resumen

En esta lección aprendió acerca de las siguientes funciones y conceptos importantes acerca de las cadenas de C:

- Una cadena es un arreglo de caracteres con un carácter nulo que marca el final de la cadena.
- Una constante de cadena es una serie de caracteres encerrados entre comillas dobles.
- El compilador de C agrega de manera automática un carácter nulo a una constante de cadena que se usó para inicializar un arreglo.
- No se puede asignar una constante de cadena a un apuntador de tipo `char` indireccionado (con el operador de indirección).
- La función `strlen()` se puede utilizar para medir la longitud de una cadena. Esta función no cuenta el carácter nulo.
- Es posible copiar una cadena de un arreglo a otro, llamando a la función `strcpy()` de C.
- La función `gets()` se puede utilizar para leer una serie de caracteres. Esta función deja de leer cuando encuentra un carácter de línea nueva o un fin de archivo (`EOF`). La función agrega un carácter nulo al final de la cadena.

Después de leer la cadena, se agrega en forma automática un carácter nulo al arreglo.

Observe que con `scanf()`, a diferencia de `printf()`, usted debe pasar apuntadores a sus argumentos para que la función `scanf()` pueda modificar sus valores.

El programa del listado 13.5 muestra cómo usar varios especificadores de formato con la función `scanf()`.

LISTADO 13.5 Uso de la función `scanf()` con varios especificadores de formato

```
1:  /* 13L05.c: Uso de scanf() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char cadena[80];
7:     int x, y;
8:     float z;
9:
10:    printf("Introduzca dos enteros separados por un espacio:\n");
11:    scanf("%d  %d", &x, &y);
12:    printf("Introduzca un real (punto flotante):\n");
13:    scanf("%f", &z);
14:    printf("Introduzca una cadena:\n");
15:    scanf("%s", cadena);
16:    printf("Usted introdujo:\n");
17:    printf("%d  %d\n%f\n%s\n", x, y, z, cadena);
18:    return 0;
19: }
```

Después de ejecutar el archivo `13L05.exe`, e introducir datos (los cuales aparecen en negritas) desde mi teclado, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
Introduzca dos enteros separados por un espacio:
10 12345
Introduzca un real (punto flotante):
1.234567
Introduzca una cadena:
Prueba
Usted introdujo:
10 12345
1.234567
Prueba
```

ANÁLISIS En las líneas 6 a 8 del listado 13.5 se declaran un arreglo de tipo `int` (`cadena`), dos variables `int` (`x` y `y`) y una variable `float` (`z`), respectivamente.

Materiał chroniony prawem autorskim

variable entera, `delt`. En la línea 7 se inicializa la variable `delt` con el valor de la expresión `'a' - 'A'`, que es la diferencia entre el valor numérico del carácter a en minúscula y su contraparte en mayúscula. En otras palabras, al restar la diferencia de `'a'` y `'A'` del valor entero de la minúscula, obtenemos el valor entero de la mayúscula.

Después, la función `puts()` de la línea 18 imprime la cadena con todos los caracteres en mayúsculas en `stdout`, el cual va a la pantalla de manera predeterminada. La función `puts()` agrega un carácter de línea nueva cuando encuentra el carácter nulo al final de la cadena.

Uso de `%s` con la función `printf()`

Hemos usado la función `printf()` en muchos programas de ejemplo de este libro. Como usted sabe, con esta función se pueden utilizar muchos especificadores de formato para indicar diferentes formas de despliegue de los datos de diversos tipos.

Por ejemplo, puede usar el especificador de formato de cadena, `%s`, con la función `printf()` para desplegar una cadena de caracteres guardada en un arreglo. (Consulte el ejemplo del listado 13.3.)

En la siguiente sección se presenta la función `scanf()` como una forma de leer valores de distintos tipos de datos con diferentes especificadores de formato, incluyendo al especificador de formato `%s`.

La función `scanf()`

La función `scanf()` proporciona otra forma de leer cadenas desde el flujo de entrada estándar. Es más, esta función se puede utilizar para leer varios tipos de datos de entrada. Los formatos de los argumentos para la función `scanf()` son muy similares a los que emplea la función `printf()`.

SINTAXIS

La sintaxis de la función `scanf()` es

```
#include <stdio.h>
int scanf(const char *formato,...);
```

Aquí se pueden incluir varios especificadores de formato dentro de la cadena de formato a la que hace referencia la variable de apuntador *`formato`* de tipo `char`. Si la función `scanf()` termina con éxito, devuelve el número de elementos de datos leídos desde `stdin`. Si ocurre un error, la función `scanf()` devuelve `EOF` (fin de archivo).

El uso del especificador de formato de cadena `%s` le indica a la función `scanf()` que continúe leyendo hasta que encuentre un carácter de espacio, de línea nueva, de tabulador, de tabulador vertical, o de avance de página. Los caracteres que lee la función `scanf()` se almacenan en el arreglo al que hace referencia el argumento correspondiente. El arreglo debe ser lo suficientemente grande para almacenar los caracteres de entrada.

LISTADO 13.4 Uso de las funciones `gets()` y `puts()`

```
 1:  /* 13L04.c: Uso de gets() y puts() */
 2:  #include <stdio.h>
 3:
 4:  main()
 5:  {
 6:     char cadena[80];
 7:     int i, delt = 'a' - 'A';
 8:
 9:     printf("Introduzca una cadena que tenga menos de 80 caracteres:\n");
10:     gets( cadena );
11:     i = 0;
12:     while (cadena[i]){
13:       if ((cadena [i] >= 'a') && (cadena [i] <= 'z'))
14:          cadena [i] -= delt;  /* convierte a mayúsculas */
15:       ++i;
16:     }
17:     printf("La cadena introducida (en mayúsculas) es:\n");
18:     puts( cadena );
19:     return 0;
20: }
```

Durante la ejecución del archivo `13L04.exe`, introduje desde el teclado una línea de caracteres (abajo en negritas), y obtuve los siguientes caracteres en la pantalla (todos en mayúsculas).

SALIDA

```
Introduzca una cadena que tenga menos de 80 caracteres:
Esto es una prueba.
La cadena introducida (en mayúsculas) es:
ESTO ES UNA PRUEBA.
```

ANÁLISIS El programa del listado 13.4 acepta una cadena de caracteres que se introduce desde el teclado (es decir, `stdin`), y luego convierte a mayúsculas los caracteres que están en minúsculas. Por último, la cadena modificada se muestra en la pantalla.

En la línea 6 se declara un arreglo de caracteres (`cadena`) que puede contener hasta `80` caracteres. La función `gets()` de la línea 10 lee los caracteres que introduzce el usuario desde el teclado, hasta que éste oprime la tecla Entrar, lo que se interpreta como un carácter de línea nueva. Los caracteres leídos por la función `gets()` se almacenan en el arreglo de caracteres indicado por `cadena`. El carácter de línea nueva no se almacena en `cadena`. En su lugar, se agrega un carácter nulo al arreglo como terminador.

El ciclo `while` de las líneas 12 a 16 tiene una expresión condicional, `cadena[i]`. Dicho ciclo continúa iterando mientras `cadena[i]` se evalúe como diferente de cero. Dentro del ciclo, el valor de cada carácter representado por `cadena[i]` se evalúa en la línea 13, para determinar si es un carácter en minúscula dentro del rango de la `a` a la `z`. Si éste es el caso, se convierte a mayúsculas en la línea 14, restando a su valor actual el valor de una

Materiał chroniony prawem autorskim

pantalla. Observe que el especificador de formato %s y la dirección de inicio de cadena2 y cadena3 se pasan a la llamada a printf() de las líneas 19 y 20, para que ésta imprima todos los caracteres almacenados en cadena2 y cadena3, excepto el carácter nulo. Los resultados que se despliegan en la pantalla muestran que cadena2 y cadena3 tienen exactamente el mismo contenido que cadena1.

Lectura y escritura de cadenas

Ahora nos vamos a concentrar en cómo leer o escribir cadenas con los flujos de entrada y salida estándar; es decir, stdin y stdout. Hay varias funciones en C que usted puede utilizar para la lectura o escritura de cadenas. Las subsecciones que siguen presentan algunas de estas funciones.

Las funciones gets() y puts()

La función gets() se puede usar para leer caracteres desde el flujo de entrada estándar.

SINTAXIS

La sintaxis de la función gets() es

```
#include <stdio.h>
char *gets(char *s);
```

Aquí, los caracteres que se leen del flujo de entrada estándar se almacenan en el arreglo de caracteres identificado por s. Cuando la función gets() encuentra un carácter de línea nueva o de fin de archivo (EOF), deja de leer y agrega un carácter nulo \0 al arreglo. Si la función concluye con éxito, devuelve s. En caso contrario, devuelve un apuntador nulo.

La función puts() se puede usar para escribir caracteres en el flujo de salida estándar (es decir, en stdout).

SINTAXIS

La sintaxis de la función puts () es

```
#include <stdio.h>
int puts(const char *s);
```

Aquí, s se refiere al arreglo de caracteres que contiene una cadena. La función puts() escribe la cadena en stdout. Si la función tiene éxito, devuelve 0. En caso contrario, devuelve un valor diferente de cero.

La función puts() agrega un carácter de línea nueva para reemplazar el carácter nulo del final del arreglo de caracteres.

Las funciones gets() y puts() necesitan el archivo de encabezado stdio.h. Puede ver en el listado 13.4 la aplicación de las dos funciones.

LISTADO 13.3 continuación

```
6:  {
7:      char cadena1[] = "Copia una cadena.";
8:      char cadena2[18];
9:      char cadena3[18];
10:     int  i;
11:
12:     /* con strcpy() */
13:     strcpy(cadena2, cadena1);
14:     /* sin strcpy() */
15:     for (i=0; cadena1[i]; i++)
16:        cadena3[i] = cadena1[i];
17:     cadena3[i] = '\0';
18:     /* muestra cadena2 y cadena3 */
19:     printf("El contenido de cadena2 usando strcpy: %s\n", cadena2);
20:     printf("El contenido de cadena3 sin usar strcpy: %s\n", cadena3);
21:     return 0;
22: }
```

Después de crear y ejecutar el archivo `13L03.exe`, se desplegaron los siguientes resultados:

SALIDA

```
El contenido de cadena2 usando strcpy: Copia una cadena.
El contenido de cadena3 sin usar strcpy: Copia una cadena.
```

ANÁLISIS En el listado 13.3 se declaran tres arreglos de tipo `char`, `cadena1`, `cadena2` y `cadena3`. Además, en la línea 7 se inicializa `cadena1` con una constante de cadena, `"Copia una cadena."`.

La instrucción de la línea 13 llama a la función `strcpy()` para copiar el contenido de `cadena1` (incluyendo el carácter nulo que agrega el compilador) al arreglo al que hace referencia `cadena2`.

Las líneas 15 a 17 muestran otra forma de copiar el contenido de `cadena1` a un arreglo al que hace referencia `cadena3`. Para ello, el ciclo `for` de las líneas 15 y 16 continúa copiando, uno tras otro, los caracteres de `cadena1` a los elementos correspondientes de `cadena3`, hasta que encuentra el carácter nulo que agrega el compilador. Cuando esto sucede, se evalúa como `0` la expresión `cadena1[i]` que se usa como la condición de la instrucción `for` de la línea 15, lo cual finaliza el ciclo.

Debido a que el ciclo `for` no copia el carácter nulo de `cadena1` a `cadena3`, la instrucción de la línea 17 agrega un carácter nulo al arreglo al que hace referencia `cadena3`. En C, es muy importante asegurarse de que cualquier arreglo que se utilice para almacenar una cadena tenga un carácter nulo al final de la misma.

Para demostrar que la constante de carácter a que hace referencia `cadena1` se copió con éxito a `cadena2` y a `cadena3`, el contenido de `cadena2` y `cadena3` se despliega en la

Materiał chroniony prawem autorskim

```
17:     return 0;
18: }
```

Obtuve los siguientes resultados al ejecutar el archivo `13L02.exe` del programa del listado 13.2.

SALIDA

```
La longitud de cadena1 es de: 23 bytes
La longitud de cadena2 es de: 24 bytes
La longitud de la cadena asignada a aptr_cadena es de: 36 bytes
```

ANÁLISIS En las líneas 7 a 11 del listado 13.2 se declaran e inicializan dos arreglos de tipo `char`, `cadena1` y `cadena2`, y una variable de apuntador, `aptr_cadena`, respectivamente.

Luego, la instrucción de la línea 13 obtiene la longitud de la cadena `cadena1`, e imprime el resultado. Puede ver en dicho resultado que la función `strlen()` no cuenta el carácter nulo (`\0`) que está al final de `cadena1`.

En las líneas 14 a 16 se miden y se muestran en la pantalla las longitudes de las constantes de cadena a las que hacen referencia `cadena2` y `aptr_cadena`. Los resultados indican que la función `strlen()` tampoco cuenta los caracteres nulos que agrega el compilador a las dos constantes de cadena.

Cómo copiar cadenas con `strcpy()`

Si desea copiar una cadena de un arreglo a otro, puede copiar cada elemento del primer arreglo al elemento correspondiente del segundo, o puede simplemente llamar a la función de C `strcpy()` para que haga el trabajo por usted.

SINTAXIS

La sintaxis de la función `strcpy()` es

```
#include <string.h>
char *strcpy(char *dest, const char *orig);
```

Aquí, el contenido de la cadena *`orig`* se copia al arreglo al que hace referencia *`dest`*. Si el copiado se realiza con éxito, la función `strcpy()` devuelve el valor de *`orig`*. Debe incluir en su programa el archivo de encabezado `string.h` antes de llamar a la función `strcpy()`.

El programa del listado 13.3 muestra cómo copiar una cadena de un arreglo a otro, ya sea llamando a la función `strcpy()` o haciéndolo usted mismo.

LISTADO 13.3 Cómo copiar cadenas

```
1:  /* 13L03.c: Cómo copiar cadenas */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main()
```

continúa

de cadena. Cuando se encuentra el carácter nulo agregado a la cadena, `*aptr_cadena` se evalúa como `0`, lo cual hace que se detenga la iteración del ciclo `for`. La expresión `*aptr_cadena++` de la línea 24 mueve el apuntador al siguiente carácter de la cadena después de extraer el carácter actual al que hace referencia el apuntador. En la hora 16, "Uso de apuntadores", aprenderá más acerca de la aritmética de los apuntadores.

¿Cuánto mide una cadena?

En ocasiones, usted necesita saber cuántos bytes ocupa una cadena, ya que su longitud puede ser menor a la de su arreglo de tipo `char`. En C, puede utilizar una función llamada `strlen()` para medir la longitud de una cadena.

La función `strlen()`

Veamos la sintaxis de la función `strlen()`.

SINTAXIS

La sintaxis de la función `strlen()` es

```
#include <string.h>
size_t strlen(const char *s);
```

Aquí, `s` es una variable de apuntador de tipo `char`. El valor de retorno de la función es el número de bytes de la cadena, sin contar el carácter nulo `'\0'`. `size_t` es un tipo de datos definido en el archivo de encabezado `string.h`. El tamaño del tipo de datos depende del sistema de cómputo en particular. Observe que debe incluir `string.h` en su programa antes de poder llamar a la función `strlen()`.

El listado 13.2 proporciona un ejemplo del uso de la función `strlen()` para medir la longitud de cadenas.

LISTADO 13.2 Cómo medir longitudes de cadenas

```
1:  /* 13L02.c: Cómo medir longitudes de cadenas */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main()
6:  {
7:     char cadena1[] = {'U', 'n', 'a', ' ',
8:                       'c', 'o', 'n', 's', 't', 'a', 'n', 't', 'e', ' ',
9:                       'd', 'e', ' ', 'c', 'a', 'd', 'e', 'n', 'a', '\0'};
10:    char cadena2[] = "Otra constante de cadena";
11:    char *aptr_cadena = "Asigna una constante a un apuntador.";
12:
13:    printf("La longitud de cadena1 es de:  %d bytes\n", strlen(cadena1));
14:    printf("La longitud de cadena2 es de: %d bytes\n", strlen(cadena2));
15:    printf("La longitud de la cadena asignada a aptr_cadena es de: %d bytes\n",
16:       strlen(aptr_cadena));
```

Materiał chroniony prawem autorskim

```
16:     printf("\n");
17:     /* imprime cadena2 */
18:     for (i=0; cadena2[i]; i++)
19:        printf("%c", cadena2[i]);
20:     printf("\n");
21:     /* asigna una cadena a un apuntador */
22:     aptr_cadena = "Asigna una cadena a un apuntador.\n";
23:     for (i=0; *aptr_cadena; i++)
24:        printf("%c", *aptr_cadena++);
25:     return 0;
26: }
```

Después de crear y ejecutar el archivo 13L01.exe del programa del listado 13.1, se desplegaron los siguientes resultados.

SALIDA

```
Una constante de cadena
Otra constante de cadena
Asigna una cadena a un apuntador.
```

ANÁLISIS Como puede ver en el listado 13.1, en las líneas 6 a 9 se declaran e inicializan dos arreglos de caracteres, cadena1 y cadena2. En la declaración de cadena1 se usa un conjunto de constantes de carácter, incluyendo un carácter nulo, para inicializar el arreglo. En la línea 9 se asigna una constante de cadena al arreglo cadena2. El compilador agregará un carácter nulo a cadena2. Observe que tanto cadena1 como cadena2 se declaran como arreglos sin especificación de tamaño, para los cuales el compilador determinará en forma automática cuánta memoria se necesita. La instrucción de la línea 10 declara una variable de apuntador de tipo char, aptr_cadena.

El ciclo for de las líneas 14 y 15 imprime entonces todos los elementos de cadena1. Debido a que el último elemento contiene un carácter nulo (\0) que se evalúa como 0, cadena1[i] se usa como la expresión condicional del ciclo for. La expresión cadena1[i] se evalúa como diferente de cero para cada elemento de cadena1, excepto para el que contiene el carácter nulo. Después de la ejecución del ciclo for, se muestra en la pantalla la cadena Una constante de cadena.

De la misma forma, el ciclo for de las líneas 18 y 19 muestra la constante de cadena asignada a cadena2 imprimiendo en la pantalla cada elemento del arreglo. La expresión cadena2[i] se evalúa en la instrucción for, debido a que el compilador agrega un carácter nulo al arreglo. El ciclo for detiene su iteración cuando cadena2[i] se evalúa como 0. Para ese entonces, el contenido de la constante de cadena, Otra constante de cadena, ya se desplegó en la pantalla.

La instrucción de la línea 22 asigna una constante de cadena, "Asigna una cadena a un apuntador.", a la variable de apuntador de tipo char, aptr_cadena. Aquí también se usa un ciclo for para imprimir la constante de cadena, el cual despliega en la pantalla cada carácter de la cadena (vea las líneas 23 y 24). Observe que se usa el operador de indirección *aptr_cadena para hacer referencia a uno de los caracteres de la constante

```
char ch = 'x';
char cadena[] = "x";
```

se reserva un byte para la variable de carácter `ch`, y se asignan dos bytes para el arreglo de caracteres `cadena`. La razón por la que se necesita un byte adicional para `cadena` es que el compilador tiene que agregar un carácter nulo al arreglo.

Otra cosa importante es que debido a que la cadena es un arreglo, en realidad es un apuntador de tipo `char`. Por lo tanto, usted puede asignar directamente una cadena de caracteres a una variable de apuntador, de la siguiente manera:

```
char *aptr_cadena;
aptr_cadena = "Una cadena de caracteres.";
```

Sin embargo, no puede asignar una constante de carácter a la variable de apuntador, como se muestra a continuación:

```
aptr_cadena = 'x';  /* Es incorrecto. */
```

En otras palabras, la constante de carácter `'x'` contiene un valor derecho, y la variable de apuntador `aptr_cadena` espera un valor izquierdo. Pero C requiere el mismo tipo de valores a ambos lados del operador de asignación =.

Es válido asignar una constante de carácter a un apuntador de tipo `char` indireccionado, de la siguiente manera:

```
char *aptr_cadena;
*aptr_cadena = 'x';
```

Ahora los valores que están a ambos lados del operador = son del mismo tipo.

El programa del listado 13.1 muestra cómo inicializar, o asignar, arreglos de caracteres con constantes de cadena.

LISTADO 13.1 Inicialización de cadenas

```
1:  /* 13L01.c: Inicialización de cadenas */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char cadena1[] = {'U', 'n', 'a', ' ',
7:                       'c', 'o', 'n', 's', 't', 'a', 'n', 't', 'e', ' ',
8:                       'd', 'e', ' ', 'c', 'a', 'd', 'e', 'n',  'a', '\0'};
9:     char cadena2[] = "Otra constante de cadena";
10:    char *aptr_cadena;
11:    int i;
12:
13:    /* imprime cadena1 */
14:    for (i=0; cadena1[i]; i++)
15:       printf("%c", cadena1[i]);
```

Materiał chroniony prawem autorskim

El compilador agregará en forma automática un carácter nulo (`\0`) al final de `"¡Hola!"`, y tratará al arreglo de caracteres como una cadena de caracteres. Observe que el tamaño del arreglo se especifica para contener hasta siete elementos, aunque la constante de cadena sólo tiene seis caracteres encerrados entre comillas dobles. El espacio extra se reserva para el carácter nulo que agregará el compilador.

Si desea que el compilador calcule el número total de elementos del arreglo, puede declarar un arreglo de caracteres sin especificar su tamaño, es decir, sin dimensionarlo. Por ejemplo, la siguiente instrucción

```
char cadena[] = "Me gusta C.";
```

inicializa con una constante de cadena el arreglo de caracteres `cadena`, sin especificar el tamaño. Más adelante, cuando el compilador vea la instrucción, determinará el espacio total de memoria necesario para contener la constante de cadena, más un carácter nulo que agrega el compilador mismo.

Si lo desea, puede también declarar un apuntador de tipo `char` y luego inicializarlo con una constante de cadena. La siguiente instrucción es un ejemplo:

```
char *aptr_cadena = "Yo aprendo C por mi cuenta.";
```

No especifique muy reducido el tamaño de un arreglo de caracteres. De hacerlo así, éste no podrá contener una constante de caracteres más un carácter nulo adicional. Por ejemplo, la siguiente declaración se considera ilegal:

```
char cadena[5] = "texto";
```

Observe que muchos compiladores de C no emitirán un mensaje de advertencia o de error sobre esta declaración incorrecta. Los errores en tiempo de ejecución que podrían surgir a la larga como resultado, podrían ser muy difíciles de depurar. Por lo tanto, es su responsabilidad asegurarse de especificar espacio suficiente para una cadena.

La siguiente instrucción es correcta, ya que especifica el tamaño del arreglo de caracteres cadena lo suficientemente grande para contener la constante de caracteres más el carácter nulo adicional:

```
char cadena[6] = "texto";
```

Constantes de cadena en comparación con constantes de carácter

Como ya sabe, una constante de cadena es una serie de caracteres encerrados entre comillas dobles (" "). Por otra parte, una constante de carácter es un carácter encerrado entre comillas sencillas (' ').

Declaración de cadenas

Esta sección le enseña cómo declarar e inicializar cadenas, así como la diferencia entre constantes de cadena y constantes de carácter. Revisemos primero la definición de una cadena.

Qué es una cadena

Como vimos en la hora 12, "Arreglos", una *cadena* es un arreglo de caracteres, en el cual se utiliza un carácter nulo (`\0`) para marcar el final de la cadena. (La longitud de una cadena puede ser más corta que su arreglo de caracteres.)

Por ejemplo, el arreglo de caracteres, `arreglo_ch`, declarado en la siguiente instrucción es considerado como una cadena de caracteres:

```
char arreglo_ch[7] = {'¡', 'H', 'o', 'l', 'a', '!', '\0'};
```

En C, el carácter nulo se emplea para marcar el final de una cadena, y siempre se evalúa como `0`. C trata al `\0` como un carácter. Cada carácter de una cadena ocupa sólo 1 byte.

Una serie de caracteres encerrados entre comillas dobles (`""`) se llama *constante de cadena*. El compilador de C agregará en forma automática un carácter nulo (`\0`) al final de una constante de cadena, para indicar el fin de la cadena.

Por ejemplo, la cadena de caracteres `"Una cadena de caracteres."` se considera una constante de cadena, lo mismo que `"¡Hola!"`.

Inicialización de cadenas

Como vimos en la lección anterior, un arreglo de caracteres se puede declarar e inicializar de la siguiente manera:

```
char arr_cadena[6] = {'¡', 'H', 'o', 'l', 'a', '!'};
```

Aquí, el arreglo `arr_cadena` es tratado como un arreglo de caracteres. Sin embargo, si le agrega un carácter nulo (`\0`) al arreglo, puede tener la siguiente instrucción:

```
char arr_cadena[7] = {'¡', 'H', 'o', 'l', 'a', '!', '\0'};
```

Aquí, el arreglo `arr_cadena` se amplía para contener siete elementos; el último elemento es un carácter nulo. Ahora, el arreglo de caracteres `arr_cadena` es considerado una cadena de caracteres debido al carácter nulo que se agregó al final.

También puede inicializar un arreglo de caracteres con una constante de cadena, en lugar de una lista de constantes de carácter. Por ejemplo, la siguiente instrucción inicializa un arreglo de caracteres, `cadena`, con una constante de cadena, `"¡Hola!"`:

```
char cadena[7] = "¡Hola!";
```

Materiał chroniony prawem autorskim

Hora 13

Cadenas

Hice esta carta más larga que de costumbre, porque me faltó tiempo para hacerla más breve.

—B. Pascal

En la hora anterior aprendió a usar arreglos para reunir variables del mismo tipo. También aprendió que una cadena de caracteres es en realidad un arreglo de caracteres, con un carácter nulo `\0` que marca el final de la cadena. En esta lección aprenderá más acerca de cadenas y funciones de C que se pueden utilizar para manipularlas. Así pues, trataremos los siguientes temas:

- Declaración de una cadena
- La longitud de una cadena
- Cómo copiar cadenas
- Lectura de cadenas con `scanf()`
- Las funciones `gets()` y `puts()`

3. ¿Cuántas dimensiones tienen los siguientes arreglos?

   - `char arreglo1[3][19];`
   - `int arreglo2[];`
   - `float arreglo3[][8][16];`
   - `char arreglo4[][80];`

4. ¿Qué está mal en la siguiente declaración?

```
char lista_ch[][] = {
          'A', 'a',
          'B', 'b',
          'C', 'c',
          'D', 'd',
```

Ejercicios

1. Dado el arreglo de caracteres:

```
char arreglo_ch[5] = {'A', 'B', 'C', 'D', 'E'};
```

   escriba un programa que despliegue en la pantalla cada elemento del arreglo.

2. Reescriba el programa del ejercicio 1, pero esta vez utilice un ciclo `for` que inicialice el arreglo de caracteres con `'a'`, `'b'`, `'c'`, `'d'` y `'e'`, e imprima después el valor de cada elemento del arreglo.

3. Dado el siguiente arreglo bidimensional sin especificación del tamaño de la primera dimensión:

```
char lista_ch[][2] = {
          '1', 'a',
          '2', 'b',
          '3', 'c',
          '4', 'd',
          '5', 'e',
          '6', 'f'};
```

   escriba un programa que mida el total de bytes que ocupa el arreglo, y que imprima después todos los elementos del mismo.

4. Reescriba el programa del listado 12.5. Esta vez ponga en la pantalla la cadena de caracteres `¡Me gusta C!`

5. Dado el siguiente arreglo:

```
double lista_datos[6] = {
          1.12345,
          2.12345,
          3.12345,
          4.12345,
          5.12345};
```

   utilice las dos formas equivalentes que vimos en esta lección para medir el espacio total de memoria que ocupa el arreglo, y despliegue los resultados en la pantalla.

P ¿Cuál es el índice mínimo en un arreglo?

R En C, el índice mínimo de un arreglo de una sola dimensión es 0, el cual marca el primer elemento del arreglo. Por ejemplo, dado el arreglo entero

```
int arreglo_ent[8];
```

el primer elemento del arreglo es `arreglo_ent[0]`.

De la misma manera, el índice mínimo de cada dimensión de un arreglo multidimensional comienza en 0.

P ¿Cómo hace referencia a un arreglo usando un apuntador?

R Puede emplear un apuntador para hacer referencia a un arreglo, asignándole al apuntador la dirección de inicio del arreglo. Por ejemplo, dada una variable de apuntador `aptr_ch` y un arreglo de caracteres `arreglo_ch`, puede usar una de las siguientes instrucciones para hacer referencia al arreglo mediante el apuntador:

```
aptr_ch = arreglo_ch;
```

o bien,

```
aptr_ch = &arreglo_ch[0];
```

P ¿Qué hace el carácter nulo?

R En C, el carácter nulo (`'\0'`) se puede utilizar para marcar el final de una cadena. Por ejemplo, la función `printf()` continúa poniendo en la pantalla el siguiente carácter hasta que encuentra el carácter nulo. Además, el carácter nulo siempre se evalúa como un valor de cero.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Qué hace la siguiente instrucción?

```
int arreglo_ent[4] = {12, 23, 9, 56};
```

2. Dado el arreglo `int datos[3]`, ¿qué está mal en la siguiente inicialización?

```
datos[1] = 1;
datos[2] = 2;
datos[3] = 3;
```

Materiał chroniony prawem autorskim

(Si el tamaño de `int` es diferente en su máquina, podría obtener valores diferentes a los del tamaño del arreglo `lista_ent` del programa del listado 12.7.)

Resumen

En esta lección aprendió los siguientes conceptos importantes acerca de los arreglos en C:

- Un arreglo es una colección de variables que son del mismo tipo de datos.
- En C, el índice para un arreglo comienza en 0.
- Puede inicializar cada elemento de un arreglo después de declararlo, o puede poner todos los valores iniciales, separados por comas, dentro de un bloque de datos encerrado entre llaves (`{` y `}`) durante la declaración de un arreglo.
- El espacio de memoria que ocupa un arreglo está determinado por la multiplicación del tamaño del tipo de datos por las dimensiones del arreglo.
- Se dice que un apuntador *hace referencia* a un arreglo cuando la dirección del primer elemento del arreglo se asigna al apuntador. La dirección del primer elemento de un arreglo también se conoce como dirección de inicio del arreglo.
- Para asignar la dirección de inicio de un arreglo a un apuntador, puede poner la combinación del operador de dirección (`&`) y el nombre del primer elemento del arreglo, o simplemente usar el nombre del arreglo a la derecha del operador de asignación (`=`).
- El carácter nulo (`'\0'`) marca el final de una cadena. Las funciones de C, como `printf()`, detendrán el procesamiento de la cadena al encontrar el carácter nulo.
- C también maneja arreglos multidimensionales. Un par de corchetes vacío (el operador de subíndices de arreglo, `[` y `]`) indica una dimensión.
- El compilador puede calcular en forma automática el espacio de memoria necesario para un arreglo sin especificación de tamaño.

En la siguiente lección aprenderá más acerca de las cadenas de C.

Preguntas y respuestas

P ¿Por qué necesita usar los arreglos?

R En muchos casos necesita declarar un conjunto de variables que son del mismo tipo de datos. En lugar de declarar cada variable por separado, puede declararlas todas en forma colectiva en el formato de un arreglo. Se puede acceder a cada variable, como un elemento del arreglo, ya sea a través de una referencia al elemento del arreglo, o por medio de un apuntador que haga referencia al arreglo.

TECLEE **LISTADO 12.7** Inicialización de arreglos sin especificación de tamaño

```
1:  /* 12L07.c: Inicialización de arreglos sin especificación de tamaño*/
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char arreglo_ch[] = {'I', 'C', ' ',
7:                          'e', 's', ' ',
8:                          'm', 'u', 'y', ' ',
9:                          'p', 'o', 't', 'e', 'n', 't', 'e', '!', '\0'};
10:    int lista_ent[][3] = {
11:           1, 1, 1,
12:           2, 2, 8,
13:           3, 9, 27,
14:           4, 16, 64,
15:           5, 25, 125,
16:           6, 36, 216,
17:           7, 49, 343};
18:
19:    printf("El tamaño de arreglo_ch[] es: %d bytes.\n", sizeof (arreglo_ch));
20:    printf("El tamaño de lista_ent[][3] es: %d bytes.\n", sizeof (lista_ent));
21:    return 0;
22: }
```

Obtuve los siguientes resultados al ejecutar en mi computadora el archivo `12L07.exe`:

SALIDA

```
El tamaño de arreglo_ch[] es: 19 bytes.
El tamaño de lista_ent[][3] es: 42 bytes.
```

ANÁLISIS En las líneas 6 a 9 se declara un arreglo de caracteres, `arreglo_ch`, sin especificación de tamaño. En las líneas 10 a 17 se declara e inicializa un arreglo bidimensional entero, `lista_ent`, sin especificación del tamaño de la primera dimensión.

La instrucción de la línea 19 mide e imprime el espacio total de memoria (en bytes) que ocupa el arreglo `arreglo_ch`. Los resultados muestran que a dicho arreglo se le asignan `19` bytes de memoria para contener todos los elementos después de compilar. Al calcular en forma manual el número total de elementos del arreglo de caracteres, se encuentra con que, en efecto, hay `19` elementos. Debido a que cada carácter ocupa un byte de memoria, el arreglo de caracteres `arreglo_ch` ocupa, por consiguiente, `19` bytes.

Asimismo, la instrucción de la línea 20 da el número total de bytes reservados en la memoria para el arreglo bidimensional entero sin especificación del tamaño de la primera dimensión, `lista_ent`. Ya que hay un total de `21` elementos enteros en el arreglo, y como en mi máquina un entero ocupa `2` bytes, el compilador debe asignar `42` bytes para el arreglo entero `lista_ent`. El resultado que imprime la función `printf()` de la línea 20 demuestra que hay `42` bytes reservados en la memoria para el mencionado arreglo.

Materiał chroniony prawem autorskim

ANÁLISIS Como puede ver en el listado 12.6, en las líneas 6 a 8 se declara e inicializa un arreglo bidimensional, `dos_dim`.

En las líneas 11 a 15 se anidan dos ciclos `for`. El ciclo `for` externo incrementa la variable entera `i` e imprime el carácter de línea nueva `'\n'` en cada iteración. Aquí, la variable entera `i` se usa como índice para la primera dimensión del arreglo `dos_dim`.

El ciclo `for` interno de las líneas 13 y 14 imprime cada elemento, representado por la expresión `dos_dim[i][j]`, incrementando el índice de la segunda dimensión del arreglo. Por lo tanto, obtuve los siguientes resultados después de ejecutar con éxito los dos ciclos `for` anidados:

```
     1       2       3       4       5
    10      20      30      40      50
   100     200     300     400     500
```

Arreglos sin especificación de tamaño

Como ha visto, el tamaño de una dimensión se da normalmente durante la declaración de un arreglo. Esto significa que debe contar cada elemento de un arreglo para determinar el tamaño. Aunque sería tedioso hacerlo, en especial si el arreglo tiene muchos elementos.

La buena noticia es que el compilador de C puede calcular en forma automática el tamaño de una dimensión de un arreglo si éste se declara como *arreglo sin especificación de tamaño*. Por ejemplo, cuando el compilador ve el siguiente arreglo sin especificación de tamaño:

```
int  lista_ent[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90};
```

creará un arreglo lo suficientemente grande para almacenar todos los elementos.

Asimismo, puede declarar un arreglo multidimensional sin especificación de tamaño. Sin embargo, tiene que especificar el tamaño de todas las dimensiones, salvo la que está más a la izquierda (es decir, la primera). Por ejemplo, el compilador puede reservar espacio de memoria suficiente para contener todos los elementos del siguiente arreglo bidimensional sin especificación del tamaño de la primera dimensión:

```
char lista_ch[][2] = {
          'a', 'A',
          'b', 'B',
          'c', 'C',
          'd', 'D',
          'e', 'E',
          'f', 'F',
          'g', 'G'};
```

El programa del listado 12.7 inicializa un arreglo de caracteres de una sola dimensión y un arreglo bidimensional entero sin especificación de tamaño, y después mide los espacios de memoria ocupados para almacenar los dos arreglos.

```
arreglo_ent[0][0] = 1;
arreglo_ent[0][1] = 2;
arreglo_ent[0][2] = 3;
arreglo_ent[1][0] = 4;
arreglo_ent[1][1] = 5;
arreglo_ent[1][2] = 6;
```

lo cual es equivalente a la instrucción

```
int arreglo_ent[2][3] = {1, 2, 3, 4, 5, 6};
```

También puede inicializar el arreglo `arreglo_ent` de la siguiente manera:

```
int arreglo_ent[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Observe que `arreglo_ent[0][0]` es el primer elemento del arreglo bidimensional `arreglo_ent`; `arreglo_ent[0][1]` es el segundo elemento del arreglo; `arreglo_ent[0][2]` es el tercer elemento; `arreglo_ent[1][0]` es el cuarto elemento; `arreglo_ent[1][1]` es el quinto elemento y `arreglo_ent[1][2]` es el sexto elemento del arreglo.

El programa del listado 12.6 muestra un arreglo bidimensional entero que se inicializa y se despliega en la pantalla.

TECLEE **LISTADO 12.6** Impresión de un arreglo bidimensional

```
1:  /* 12L06.c: Impresión de un arreglo 2-D */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int dos_dim[3][5] = {1, 2, 3, 4, 5,
7:                          10, 20, 30, 40, 50,
8:                          100, 200, 300, 400, 500};
9:     int i, j;
10:
11:    for (i=0; i<3; i++){
12:       printf("\n");
13:       for (j=0; j<5; j++)
14:          printf("%6d", dos_dim[i][j]);
15:    }
16:    printf("\n");
17:    return 0;
18: }
```

Obtuve los siguientes resultados al ejecutar el archivo `12L06.exe`:

SALIDA

```
     1     2     3     4     5
    10    20    30    40    50
   100   200   300   400   500
```

Materiał chroniony prawem autorskim

ANÁLISIS En las líneas 6 a 9 del listado 12.5 se declara un arreglo de caracteres, `arreglo_ch`, y se inicializa con los caracteres de la cadena `¡C es muy potente!` (incluyendo los caracteres de espacios).

Observe que el último elemento del arreglo contiene el carácter nulo (`'\0'`), el cual es necesario para terminar la cadena.

El ciclo `for` de las líneas 12 y 13 imprime cada uno de los elementos del arreglo `arreglo_ch` para mostrar en la pantalla la cadena `¡C es muy potente!` Por consiguiente, en la primera expresión de la instrucción `for` se inicializa a `0` la variable entera `i`, la cual se usa como índice del arreglo.

Después se evalúa la expresión condicional `arreglo_ch[i]`. Si la expresión se evalúa como diferente de cero, el ciclo `for` realiza una iteración; en caso contrario, el ciclo se detiene. Comenzando con el primer elemento del arreglo, la expresión `arreglo_ch[i]` continúa produciendo un valor diferente de cero hasta que encuentra el carácter nulo. Por lo tanto, el ciclo `for` puede desplegar en la pantalla todos los caracteres del arreglo, y dejar de imprimir justo después de que la expresión `arreglo_ch[i]` produce un valor de cero, al encontrar el carácter nulo.

Arreglos multidimensionales

Hasta ahora, todos los arreglos que ha visto han sido arreglos de una sola dimensión, en los que los tamaños de las dimensiones se colocan entre corchetes (`[` y `]`).

Además de los arreglos de una sola dimensión, el lenguaje C también maneja arreglos multidimensionales. Puede declarar arreglos con tantas dimensiones como permita su compilador.

La forma general de declarar un arreglo de *N* dimensiones es

```
tipo_de_datos Nombre-arreglo [Tamaño-arreglo1][Tamaño-arreglo2]. . . [Tamaño-
arregloN];
```

en donde *N* puede ser cualquier entero positivo.

Debido a que el arreglo de dos dimensiones, el cual es muy utilizado, es la forma más simple de arreglo multidimensional, en esta sección nos concentraremos en los arreglos de dos dimensiones. Sin embargo, puede aplicar en arreglos de más de dos dimensiones todo lo que aprenda en esta sección.

Por ejemplo, la siguiente instrucción declara un arreglo bidimensional entero:

```
int  arreglo_ent[2][3];
```

Aquí, los dos pares de corchetes representan las dos dimensiones con un tamaño de 2 y 3 elementos enteros, respectivamente.

Puede inicializar el arreglo bidimensional `arreglo_ent` de la siguiente manera:

Quizá se pregunte cómo sabe la función `printf()` dónde está el final del arreglo de caracteres. ¿Recuerda que el último elemento del arreglo de caracteres `arreglo_ch` es un carácter `'\0'`? Es este carácter nulo el que marca el fin de la cadena. Como ya mencioné, en C, a una secuencia de caracteres contiguos que termina con un carácter nulo se le llama cadena de caracteres. Nosotros no le indicamos a `printf()` cuántos caracteres hay en el arreglo, por lo que el especificador de formato `%s` le indica que siga imprimiendo caracteres hasta encontrar un carácter nulo, tal como hicimos nosotros en el primer método.

El carácter nulo ('\0')

En C, al carácter nulo (`'\0'`) se le trata como a un carácter; éste es un carácter especial que marca el final de una cadena. Por lo tanto, cuando funciones como `printf()` actúan sobre una cadena de caracteres, procesan un carácter tras otro hasta encontrar el carácter nulo. (En la hora 13 aprenderá más acerca de las cadenas.)

El carácter nulo (`'\0'`), el cual siempre se evalúa a un valor de cero, también se puede usar para una prueba lógica en una instrucción de control de flujo. El listado 12.5 muestra un ejemplo de cómo usar el carácter nulo en un ciclo `for`.

TECLEE **LISTADO 12.5** Terminación de la salida en el carácter nulo

```
1:  /* 12L05.c: Detener en el carácter nulo */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char arreglo_ch[19] = {'¡', 'C', ' ',
7:                            'e', 's', ' ',
8:                            'm', 'u', 'y', ' ',
9:                            'p', 'o', 't', 'e', 'n', 't', 'e', '!', '\0'};
10:    int i;
11:    /*  arreglo_ch[i] en prueba lógica */
12:    for (i=0; arreglo_ch[i]; i++)
13:       printf("%c", arreglo_ch[i]);
14:
15:    printf("\n");
16:    return 0;
17: }
```

Al ejecutar el archivo `12L05.exe` obtuve el siguiente resultado:

SALIDA

```
¡C es muy potente!
```

Materiał chroniony prawem autorskim

```
15:     /*-- método II --*/
16:     printf( "\nJuntar todos los elementos (Método II):\n");
17:     printf( "%s\n", arreglo_ch);
18:
19:     return 0;
20: }
```

Cuando ejecuté el archivo `12L04.exe` se desplegaron los siguientes resultados:

SALIDA

```
arreglo_ch[0] contiene: ¡
arreglo_ch[1] contiene: H
arreglo_ch[2] contiene: o
arreglo_ch[3] contiene: l
arreglo_ch[4] contiene: a
arreglo_ch[5] contiene: !
arreglo_ch[6] contiene:
Juntar todos los elementos (Método I):
¡Hola!
Juntar todos los elementos (Método II):
¡Hola!
```

ANÁLISIS Como puede ver en el listado 12.4, en la línea 6 se declara e inicializa un arreglo de caracteres, `arreglo_ch`. Cada uno de los elementos del arreglo se imprime por medio de la llamada a `printf()` del ciclo `for` que se muestra en las líneas 9 y 10. En el arreglo hay un total de siete elementos, los cuales contienen las siguientes constantes de carácter: `'¡'`, `'H'`, `'o'`, `'l'`, `'a'`, `'!'` y `'\0'`.

Hay dos formas de desplegar este arreglo: mostrar todos los caracteres en forma individual, o tratarlos como una cadena de caracteres.

Las líneas 12 a 14 muestran la primera forma, `arreglo_ch[i]`, la cual extrae cada elemento de manera consecutiva en un ciclo, e imprime un carácter junto al otro por medio del especificador de formato `%c` que está en la llamada a `printf()` de la línea 14.

Como mencioné antes, siempre que trate con un arreglo de caracteres, el carácter nulo `'\0'` señala el final de la cadena (aunque podría no ser el final del arreglo). Es una buena idea estar pendiente del carácter nulo para que sepa cuándo dejar de imprimir; por lo tanto, la expresión condicional de la línea 13 terminará el ciclo `for` si el elemento actual es un carácter nulo.

La segunda forma es más sencilla. Simplemente le tiene que indicar a la función `printf()` dónde encontrar el primer elemento del arreglo (la dirección del primer elemento). También necesita emplear el especificador de formato de cadena `%s` en la llamada a `printf()`, como se muestra en la línea 17. Observe que la expresión `arreglo_ch` de la línea 17 contiene la dirección del primer elemento del arreglo, es decir, la dirección inicial del arreglo. El nombre del arreglo, por sí mismo, es una forma abreviada de decir `arreglo_ch[0]`; ambos significan lo mismo.

La instrucción de la línea 12 asigna la dirección del primer elemento del arreglo a la variable de apuntador aptr_int. Para hacerlo, simplemente se coloca el nombre del arreglo lista_ent a la derecha del operador de asignación (=).

La línea 13 despliega la dirección asignada a la variable de apuntador aptr_int. Los resultados muestran que la dirección inicial del arreglo es 0x1802. (Usted podría obtener una dirección diferente en su máquina.) La expresión *aptr_int de la línea 14 da el valor al que hace referencia el apuntador. El primer elemento del arreglo contiene este mismo valor , el cual es el valor inicial, 1, asignado en el ciclo for. Puede ver que el resultado de la instrucción de la línea 14 muestra correctamente el valor.

La instrucción de la línea 15 equivale a la de la línea 12, la cual asigna la dirección del primer elemento a la variable de apuntador. Las líneas 16 y 17 imprimen entonces la dirección y el valor que guarda el primer elemento, 0x1802 y 1, respectivamente.

En la hora 16, "Uso de apuntadores", aprenderá a acceder a un elemento de un arreglo incrementando o decrementando un apuntador.

Cómo desplegar arreglos de caracteres

Esta subsección se enfoca en los arreglos de caracteres. El tipo de datos char ocupa un byte. Por lo tanto, cada elemento de un arreglo de caracteres tiene un byte de longitud. El número total de elementos de un arreglo de caracteres es el número de bytes que ocupa el arreglo en la memoria.

Algo muy importante en C es que una *cadena de caracteres* se define como una secuencia continua de caracteres que termina en el carácter nulo ('\0'). La hora 13, "Cadenas", presenta más detalles acerca de las cadenas.

El listado 12.4 presenta varias formas de desplegar un arreglo de caracteres en la pantalla.

TECLEE **LISTADO 12.4** Cómo imprimir un arreglo de caracteres

```
1:  /* 12L04.c: Cómo imprimir un arreglo de caracteres */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char arreglo_ch[7] = {'¡', 'H', 'o', 'l', 'a', '!', '\0'};
7:     int i;
8:
9:     for (i=0; i<7; i++)
10:       printf("arreglo_ch[%d] contiene: %c\n", i, arreglo_ch[i]);
11:    /*---  método I ---*/
12:    printf( "Juntar todos los elementos (Método I):\n");
13:    for (i=0; arreglo_ch[i] != '\0' && i<7; i++)
14:       printf("%c", arreglo_ch[i]);
```

Materiał chroniony prawem autorskim

apuntador. Si se hace referencia a un arreglo mediante un apuntador, se puede acceder a los elementos del arreglo con ayuda del apuntador.

Por ejemplo, las siguientes instrucciones declaran un apuntador y un arreglo, y asignan la dirección del primer elemento a la variable de apuntador:

```
char  *aptr_c;
char  lista_c[10];
aptr_c = lista_c;
```

Debido a que la dirección del primer elemento del arreglo `lista_c` es la dirección de inicio del arreglo, en realidad la variable de apuntador `aptr_c` ahora está haciendo referencia al arreglo a través de su dirección inicial.

El listado 12.3 muestra cómo hacer referencia a un arreglo con un apuntador.

TECLEE **LISTADO 12.3** Cómo hacer referencia a un arreglo con un apuntador

```
1:  /* 12L03.c: Cómo hacer referencia a un arreglo con un apuntador */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int *aptr_int;
7:     int lista_ent[10];
8:     int i;
9:
10:    for (i=0; i<10; i++)
11:       lista_ent[i] = i + 1;
12:    aptr_int = lista_ent;
13:    printf( "La dirección de inicio del arreglo es: %p\n", aptr_int);
14:    printf( "El valor del primer elemento es: %d\n", *aptr_int);
15:    aptr_int = &lista_ent[0];
16:    printf( "La dirección del primer elemento es: %p\n", aptr_int);
17:    printf( "El valor del primer elemento es: %d\n", *aptr_int);
18:    return 0;
19: }
```

Después de ejecutar en mi computadora el archivo `12L03.exe`, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
La dirección de inicio del arreglo es: 0x1802
El valor del primer elemento es: 1
La dirección del primer elemento es: 0x1802
El valor del primer elemento es: 1
```

ANÁLISIS En la línea 6 del listado 12.3 se declara una variable de apuntador entera, `*aptr_int`. Luego, en la línea 7 se declara un arreglo entero, el cual se inicializa en el ciclo `for` de las líneas 10 y 11 mediante la expresión `lista_ent[i] = i + 1`.

arreglo, `lista_ent[0]`. Aquí, la dirección del primer elemento es la dirección inicial del arreglo. Puede ver en los resultados que, en mi máquina, la dirección del elemento `lista_ent[0]` es `0x1806`.

Después, la expresión `&lista_ent[9]` de la línea 13 da la dirección del último elemento del arreglo, que en mi máquina es `0x1818`. Por lo tanto, la distancia entre el último elemento y el primero es `0x1818–0x1806`, o `18` bytes.

Como mencioné anteriormente, el hexadecimal es un sistema de numeración de base 16. Sabemos que `0x1818` menos `0x1806` produce `0x0012` (es decir, `0x12`). Entonces, `0x12` en hexadecimal es igual a `1*16 + 2`, lo que produce `18` en decimal.

Debido a que cada elemento ocupa 2 bytes, y a que la dirección del último elemento es el principio de ese elemento de 2 bytes, el número total de bytes que ocupa el arreglo `lista_ent` es en realidad 20 bytes. Puede calcularlo de otra forma: la distancia entre el último elemento y el primero es de 18 bytes. El número total de bytes que ocupa el arreglo se debe contar desde el primer byte del primer elemento hasta el último byte del último elemento. Por lo tanto, el número total de bytes que ocupa el arreglo es igual a `18` más `2`, es decir `20` bytes.

La figura 12.1 le muestra el espacio de memoria que ocupa el arreglo `lista_ent`.

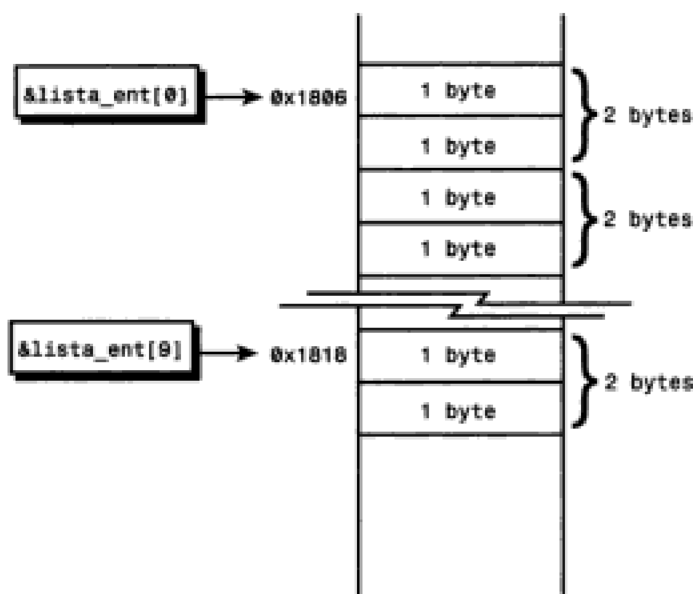


FIGURA 12.1
El espacio de memoria que ocupa el arreglo lista_ent.

Arreglos y apuntadores

Como mencioné antes en esta hora, los apuntadores y arreglos tienen una estrecha relación en C. De hecho, usted puede hacer que un apuntador haga referencia al primer elemento de un arreglo simplemente asignando el nombre del arreglo a la variable de

Materiał chroniony prawem autorskim

TECLEE **LISTADO 12.2** Cálculo del tamaño de un arreglo

```
1:  /* 12L02.c: Bytes totales de un arreglo */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int bytes_totales;
7:     int lista_ent[10];
8:
9:     bytes_totales = sizeof (int) * 10;
10:    printf( "El tamano del tipo int es de %d bytes.\n", sizeof (int));
11:    printf( "El arreglo de 10 ints tiene en total %d bytes.\n", bytes_totales);
12:    printf( "La dirección del primer elemento es: %p\n", &lista_ent[0]);
13:    printf( "La dirección del último elemento es:  %p\n", &lista_ent[9]);
14:    return 0;
15: }
```

Obtuve los siguientes resultados desplegados en la pantalla de mi computadora, después de ejecutar el archivo `12L02.exe`:

SALIDA

```
El tamaño del tipo int es de 2 bytes.
El arreglo de 10 ints tiene en total 20 bytes
La dirección del primer elemento es: 0x1806
La dirección del último elemento es: 0x1818
```

ANÁLISIS Debo decirle que al ejecutar en su máquina el programa del listado 12.2 podría obtener diferentes valores de direcciones. Sin embargo, la diferencia entre la dirección del primer elemento y la del último elemento debe ser igual al número total de bytes del arreglo.

En el listado 12.2 hay un arreglo entero, `lista_ent`, el cual se declara en la línea 7. El espacio total de memoria que ocupa el arreglo es el resultado de multiplicar el tamaño del tipo `int` por el número total de elementos del arreglo. Como se declaró en este ejemplo, hay un total de 10 elementos en el arreglo `lista_ent`.

La instrucción de la línea 10 imprime el tamaño que tiene en mi máquina el tipo `int`. Puede ver en los resultados que cada elemento entero del arreglo ocupa 2 bytes. Por lo tanto, el espacio total de memoria (en bytes) que ocupa el arreglo es `10 * 2`. En otras palabras, la instrucción de la línea 9 asigna el valor de `20`, producido por la expresión `sizeof (int) * 10`, a la variable entera `bytes_totales`. La línea 11 despliega en la pantalla el valor contenido por la variable `bytes_totales`.

Para demostrar que este arreglo sí ocupa un espacio consecutivo de memoria de 20 bytes, la instrucción de la línea 12 imprime la dirección del primer elemento del arreglo. Observe que el ampersand (`&`), el cual se presentó en la hora 11, "Apuntadores", como el operador de dirección, se usa en la línea 12 para obtener la dirección del primer elemento del

Después de crear y ejecutar en mi computadora el archivo `12L01.exe`, se desplegaron los siguientes resultados en la pantalla:

SALIDA

```
lista_ent[0] se inicializa con 1.
lista_ent[1] se inicializa con 2.
lista_ent[2] se inicializa con 3.
lista_ent[3] se inicializa con 4.
lista_ent[4] se inicializa con 5.
lista_ent[5] se inicializa con 6.
lista_ent[6] se inicializa con 7.
lista_ent[7] se inicializa con 8.
lista_ent[8] se inicializa con 9.
lista_ent[9] se inicializa con 10.
```

ANÁLISIS Como puede ver en el listado 12.1, hay un arreglo de enteros, llamado `lista_ent`, declarado en la línea 7. El arreglo `lista_ent` contiene `10` elementos.

Las líneas 9 a 12 forman un ciclo `for` que itera `10` veces. La instrucción de la línea 10 inicializa `lista_ent[i]`, el iésimo elemento del arreglo `lista_ent`, con el resultado de la expresión `i + 1`.

La línea 11 imprime luego el nombre del elemento, `lista_ent[i]`, y el valor asignado al mismo.

El tamaño de un arreglo

Como mencioné antes en esta lección, un arreglo consta de ubicaciones consecutivas de memoria. Dado un arreglo como el siguiente:

```
tipo_de_datos Nombre-arreglo [Tamaño-arreglo];
```

usted puede calcular el total de bytes del arreglo mediante la siguiente expresión:

```
sizeof(tipo_de_datos) * Tamaño-arreglo
```

Aquí, *`tipo_de_datos`* es el tipo de datos del arreglo; *`Tamaño-arreglo`* especifica el número total de elementos que puede tomar el arreglo. Esta expresión da como resultado el número total de bytes que ocupa el arreglo.

Otra forma más simple de calcular el total de bytes de un arreglo es por medio de la siguiente expresión:

```
sizeof(Nombre-arreglo)
```

Aquí, *`Nombre-arreglo`* es el nombre del arreglo.

El programa del listado 12.2 muestra cómo calcular el espacio de memoria que ocupa un arreglo.

Materiał chroniony prawem autorskim

Los siete elementos del arreglo tienen las siguientes expresiones: `dia[0]`, `dia[1]`, `dia[2]`, `dia[3]`, `dia[4]`, `dia[5]` y `dia[6]`.

Debido a que estas expresiones hacen referencia a los elementos del arreglo, en ocasiones se les llama *referencias a elementos de un arreglo*.

Inicialización de arreglos

Con la ayuda de las referencias a elementos de un arreglo, usted puede inicializar cada uno de los elementos del mismo.

Por ejemplo, puede inicializar el primer elemento del arreglo `dia`, el cual se declaró en la sección anterior, de la siguiente manera:

```
dia[0] = 'D';
```

Aquí se asigna el valor numérico de `D` al primer elemento de `dia`, `dia[0]`.

De la misma manera, la instrucción `dia[1] = 'L';` asigna `'L'` al segundo elemento del arreglo, `dia[1]`.

La segunda forma de inicializar un arreglo consiste en inicializar al mismo tiempo todos sus elementos. Por ejemplo, la siguiente instrucción inicializa un arreglo de enteros, `arEnteros`:

```
int arEnteros[5] = {100, 8, 3, 365, 16};
```

Aquí, los enteros que están dentro de las llaves (`{` y `}`) se asignan respectivamente a los elementos del arreglo `arEnteros`. Es decir, `100` se asigna al primer elemento (`arEnteros[0]`), `8` al segundo elemento (`arEnteros[1]`), `3` al tercero (`arEnteros[2]`), y así sucesivamente.

El listado 12.1 proporciona otro ejemplo de la inicialización de arreglos.

TECLEE **LISTADO 12.1** Inicialización de un arreglo

```
1:  /* 12L01.c: Inicialización de un arreglo */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:     int lista_ent[10];
8:
9:     for (i=0; i<10; i++){
10:       lista_ent[i] = i + 1;
11:       printf( "lista_ent[%d] se inicializa con %d.\n", i, lista_ent[i]);
12:    }
13:    return 0;
14: }
```

Qué es un arreglo

Ahora sabe cómo declarar una variable con un tipo de datos específico, como `char`, `int`, `float` o `double`. En muchos casos es necesario declarar un conjunto de variables que tengan el mismo tipo de datos. En lugar de hacerlo en forma individual, C le permite declarar como un arreglo a un conjunto de variables del mismo tipo de datos.

Un *arreglo* es un conjunto de variables que son del mismo tipo de datos. A cada parte de un arreglo se le denomina *elemento*. Todos los elementos de un arreglo se referencian con el mismo nombre del arreglo y se almacenan en un conjunto de posiciones consecutivas de memoria.

Declaración de arreglos

La forma general para declarar un arreglo es la siguiente:

```
tipo_de_datos Nombre-arreglo [Tamaño-arreglo];
```

Aquí, *`tipo_de_datos`* es el especificador de tipo que indica de qué tipo de datos será el arreglo declarado. *`Nombre-arreglo`* es el nombre del arreglo declarado. *`Tamaño-arreglo`* define cuántos elementos puede contener el arreglo. Observe que en la declaración de un arreglo se requieren los corchetes (`[` y `]`). Este par de corchetes (`[` y `]`) se conoce como el *operador de subíndices del arreglo*.

Por ejemplo, en la siguiente instrucción se declara un arreglo de enteros,

```
int arreglo_ent[8];
```

en donde `int` especifica el tipo de datos del arreglo cuyo nombre es `arreglo_ent`. El tamaño del arreglo es `8`, lo que significa que puede almacenar ocho elementos (en este caso, enteros).

En C, debe declarar un arreglo de manera explícita, antes de poder utilizarlo, como lo hace con otras variables.

Indexación de arreglos

Después de declarar un arreglo, puede acceder por separado a cada uno de sus elementos.

Por ejemplo, la siguiente es una declaración de un arreglo de caracteres:

```
char dia[7];
```

Puede acceder a los elementos del arreglo `dia` uno tras otro. Para ello, debe utilizar el nombre del arreglo en una expresión, seguido de un número, llamado *índice*, encerrado entre corchetes.

Lo que es importante que recuerde es que en C todos los arreglos se indexan comenzando en `0`. En otras palabras, el índice del primer elemento del arreglo es `0`, no `1`. Por lo tanto, el primer elemento del arreglo `dia` es `dia[0]`. Debido a que hay 7 elementos en el arreglo, el último elemento es `dia[6]`, no `dia[7]`.

Materiał chroniony prawem autorskim

HORA 12

Arreglos

Recoged los pedazos que han sobrado, para que no se pierda nada.

—Juan 6:12

En la hora anterior aprendió acerca de los apuntadores y del concepto de indirección. En esta lección aprenderá acerca de los arreglos, que son colecciones de elementos de datos similares y están estrechamente relacionados con los apuntadores. Los temas principales que se abordan en esta lección son los siguientes:

- Arreglos de una sola dimensión
- Indexación de arreglos
- Apuntadores y arreglos
- Arreglos de caracteres
- Arreglos multidimensionales
- Arreglos sin dimensionar

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Cómo puede obtener el valor izquierdo de una variable de tipo carácter `ch`?
2. ¿Cuáles asteriscos (`*`) son operadores de indirección y cuáles son operadores de multiplicación en las siguientes expresiones?
   - `*aptr`
   - `x * y`
   - `y *= x + 5`
   - `*y *= *x + 5`
3. Dado que `x = 10`, la dirección de `x` es `0x1A38`, y `aptr_int = &x`, ¿qué producirán `aptr_int` y `*aptr_int`, respectivamente?
4. ¿Qué contiene `x` si `x = 123`, y `aptr_int = &x` después de la ejecución de `*aptr_int = 456`?

Ejercicios

1. Dadas tres variables enteras, `x = 512`, `y = 1024` y `z = 2048`, escriba un programa que imprima los valores izquierdos y derechos de dichas variables.
2. Escriba un programa que actualice el valor de la variable `flt_num` de tipo `double`, de `123.45` a `543.21`, por medio de un apuntador `double`.
3. Dada una variable de tipo carácter `ch`, y `ch = 'A'`, escriba un programa que actualice el valor de `ch` al decimal `66` por medio de un apuntador.
4. Dado que `x=5` y `y=6`, escriba un programa que multiplique los dos enteros e imprima el resultado, el cual se guarda en `x`, todo esto en forma de indirección (es decir, usando apuntadores).

Materiał chroniony prawem autorskim

- Puede actualizar el valor de una variable a la que hace referencia una variable de apuntador.
- Varios apuntadores pueden apuntar a la misma ubicación de una variable en la memoria.

En el resto del libro verá más ejemplos del uso de apuntadores.

En la siguiente lección aprenderá acerca de un tipo agregado, un arreglo, que está relacionado estrechamente con los apuntadores de C.

Preguntas y respuestas

P ¿Qué son los valores izquierdo y derecho?

R El valor izquierdo es la dirección de una variable, y el valor derecho es el contenido de la variable almacenado en la memoria. Hay dos formas de obtener el valor derecho de una variable: usar directamente el nombre de la variable, o usar el valor izquierdo de ésta y el operador de indirección para referirse al lugar en que reside el valor derecho. A la segunda forma se le llama también forma indirecta.

P ¿Cómo puede obtener la dirección de una variable?

R Utilizando el operador de dirección, `&`. Por ejemplo, dada una variable entera `x`, la expresión `&x` se evalúa como la dirección de `x`. Para imprimir la dirección de `x`, puede emplear el especificador de formato `%p` en la función `printf()`.

P ¿Qué es el concepto de indirección en términos del uso de apuntadores?

R Antes de esta hora, la única forma que usted conocía para leer o escribir en una variable era invocarla directamente. Por ejemplo, si quería escribir un decimal `16` en una variable entera `x`, podía usar la instrucción `x = 16;`.

Como aprendió en esta hora, C le permite acceder de otra forma a una variable, utilizando apuntadores. Por lo tanto, para escribir `16` en `x`, puede declarar primero un apuntador entero (`aptr`) y asignarle el valor izquierdo (dirección) de `x`, es decir, `aptr = &x;`. Luego, en lugar de ejecutar la instrucción `x = 16;`, puede emplear otra instrucción:

```
*aptr = 16;
```

Aquí, el apuntador `*aptr` hace referencia a la ubicación de memoria reservada por `x`, y el contenido almacenado en la ubicación de memoria se actualiza a `16` después de ejecutar la instrucción. De modo que, como ve, usar apuntadores para acceder las ubicaciones de memoria de variables es una forma de indirección.

P ¿Puede un apuntador nulo apuntar a datos válidos?

R No. Un apuntador nulo no puede apuntar a datos válidos. Esto se debe a que el valor contenido por un apuntador nulo se establece a `0`. Más adelante verá ejemplos del uso de apuntadores nulos en arreglos, cadenas y asignación de memoria.

La línea 11 asigna el valor izquierdo de x a la variable de apuntador aptr_1 para que se pueda utilizar a aptr_1 para hacer referencia al valor derecho de x. Para asegurarnos de que la variable de apuntador aptr_1 ya contenga la dirección de x, la línea 12 imprime el valor derecho de aptr_1, junto con su valor izquierdo. Los resultados muestran que aptr_1 sí contiene la dirección de x, 0x1838. Después, la línea 13 imprime el valor 1234, que es al que hace referencia la expresión *aptr_1. Observe que el asterisco * de esta expresión es el operador de indirección.

La expresión *aptr_2 = &x de la línea 14 asigna el valor izquierdo de x a otra variable de apuntador, aptr_2; es decir, ahora la variable de apuntador aptr_2 está enlazada a la dirección de x. La instrucción de la línea 16 muestra en la pantalla el entero 1234 mediante el operador de indirección * y su operando, aptr_2. En otras palabras, el segundo apuntador *aptr_2 hace referencia a la ubicación de memoria de x.

En la línea 17 se le asigna el valor derecho de aptr_1 a la variable de apuntador aptr_3. Debido a que aptr_1 contiene ahora la dirección de x, la expresión aptr_3 = aptr_1 equivale a aptr_3 = &x. Entonces, en los resultados producidos por las instrucciones de las líneas 18 y 19, usted ve de nuevo en la pantalla el entero 1234. Esta vez, el tercer apuntador, aptr_3, hace referencia al entero.

Resumen

En esta lección aprendió los siguientes conceptos importantes acerca de los apuntadores en C:

- Un apuntador es una variable cuyo valor apunta a otra variable.
- Una variable declarada en C tiene dos valores: el valor izquierdo y el valor derecho.
- El valor izquierdo de una variable es la dirección; el valor derecho es el contenido de la variable.
- El operador de dirección (&) se puede usar para obtener el valor izquierdo (dirección) de una variable.
- En una declaración de apuntador, el asterisco (*) le indica al compilador que la variable es una variable de apuntador.
- El operador de indirección (*) es un operador unario; como tal, sólo requiere de un operando.
- La expresión *nom_apuntador se evalúa como el valor al que apunta la variable de apuntador nom_apuntador, en donde nom_apuntador puede ser cualquier nombre de variable válido de C.
- Si se le asigna el valor 0 al valor derecho de una variable de apuntador, entonces el apuntador es nulo. Un apuntador nulo no puede apuntar a datos válidos.

Materiał chroniony prawem autorskim

tipo char, aptr_c1 = &c y aptr_c2 = &c establecen que las dos variables de apuntador apunten a la misma ubicación de memoria.

El programa del listado 11.4 muestra otro ejemplo de cómo apuntar a lo mismo con varios apuntadores.

TECLEE

LISTADO 11.4 Cómo apuntar a la misma ubicación de memoria con más de un apuntador

```
1:  /* 11L04.c: Cómo apuntar a la misma ubicación */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int x;
7:      int *aptr_1, *aptr_2, *aptr_3;
8:
9:      x = 1234;
10:     printf("x: dirección=%p, contenido=%d\n", &x, x);
11:     aptr_1 = &x;
12:     printf("aptr_1: dirección=%p, contenido=%p\n", &aptr_1, aptr_1);
13:        printf("*aptr_1 => %d\n", *aptr_1);
14:     aptr_2 = &x;
15:     printf("aptr_2: dirección=%p, contenido=%p\n", &aptr_2, aptr_2);
16:        printf("*aptr_2 => %d\n", *aptr_2);
17:     aptr_3 = aptr_1;
18:     printf("aptr_3: dirección=%p, contenido=%p\n", &aptr_3, aptr_3);
19:        printf("*aptr_3 => %d\n", *aptr_3);
20:     return 0;
21: }
```

Al ejecutar el archivo 11L04.exe en mi máquina, se desplegaron en la pantalla los siguientes resultados (dependiendo de su sistema, usted podría obtener diferentes valores de direcciones):

SALIDA

```
x: dirección=0x1838, contenido=1234
aptr_1: dirección=0x1834, contenido=0x1838
*aptr_1 => 1234
aptr_2: dirección=0x1836, contenido=0x1838
*aptr_2 => 1234
aptr_3: dirección=0x1832, contenido=0x1838
*aptr_3 => 1234
```

ANÁLISIS Como se muestra en el listado 11.4, la línea 6 declara una variable entera, x, y la línea 7 declara tres variables de apuntador enteras, aptr_1, aptr_2 y aptr_3.

La instrucción de la línea 10 imprime los valores izquierdo y derecho de x. En mi máquina, el valor izquierdo (dirección) de x es 0x1838. El valor derecho (contenido) de x es 1234, que es el valor inicial asignado a x en la línea 9.

LISTADO 11.3 continuación

```
15:         printf("*aptr_c => %c\n", *aptr_c);
16:      printf("c: dirección=%p, contenido=%c\n", &c, c);
17:      return 0;
18: }
```

Después de ejecutar el archivo 11L03.exe en mi máquina, obtuve los siguientes resultados desplegados en la pantalla:

SALIDA

```
c: dirección=0x1828, contenido=A
aptr_c: dirección=0x1826, contenido=0x1828
*aptr_c => A
aptr_c: dirección=0x1826, contenido=0x1828
*aptr_c => B
c: dirección=0x1828, contenido=B
```

ANÁLISIS En la línea 6 del listado 11.3 se declaran una variable de tipo `char`, `c`, y una variable de apuntador del mismo tipo, `aptr_c`.

En la línea 8 se inicializa la variable `c` con `'A'`, la cual es impresa junto con su dirección por la función `printf()` de la línea 9.

En la línea 10 se le asigna a la variable de apuntador `aptr_c` el valor izquierdo (dirección) de `c`. No es una sorpresa ver en la salida impresa por las instrucciones de las líneas 11 y 12 que el valor derecho de `aptr_c` es el valor izquierdo de `c`, y que el apuntador `*aptr_c` apunta al valor derecho de `c`.

La expresión `*aptr_c = 'B'` de la línea 13 pide a la computadora que escriba `'B'` en la ubicación a la que apunta el apuntador `aptr_c`. El resultado que imprime la instrucción de la línea 15 demuestra que el contenido de la ubicación de memoria a la que apunta `aptr_c` se actualiza. La instrucción de la línea 14 imprime los valores izquierdo y derecho de la variable de apuntador `aptr_c` y muestra que estos valores permanecen iguales. Como usted sabe, la ubicación a la que apunta `aptr_c` es en donde reside la variable de carácter `c`. Por lo tanto, la expresión `*aptr_c = 'B'` en realidad actualiza el contenido (es decir, el valor derecho) de la variable `c` en `'B'`. Para demostrarlo, la instrucción de la línea 16 despliega en la pantalla los valores izquierdo y derecho de `c`. El resultado muestra que, en efecto, se modificó el valor derecho de `c`.

Cómo apuntar a la misma ubicación de memoria

Es posible apuntar a una ubicación de memoria por medio de más de un apuntador. Por ejemplo, dado que `c = 'A'` y que `aptr_c1` y `aptr_c2` son dos variables de apuntador de

Materiał chroniony prawem autorskim

Apuntadores nulos

Se dice que un *apuntador es nulo* cuando su valor derecho es `0`. Recuerde, un apuntador nulo nunca puede apuntar a datos válidos. Por esta razón, puede verificar un apuntador para ver si tiene asignado `0`; si es así, usted sabe que se trata de un apuntador nulo y que no es válido.

Para establecer un apuntador nulo, simplemente asigne `0` a la variable de apuntador. Por ejemplo:

```
char *aptr_c;
int  *aptr_int;

aptr_c = aptr_int = 0;
```

Aquí, `aptr_c` y `aptr_int` se convierten en apuntadores nulos después de asignarles el valor entero `0`.

Más adelante verá aplicaciones de los apuntadores nulos utilizados en instrucciones de control de flujo y en arreglos.

Actualización de variables por medio de apuntadores

Como aprendió en la sección anterior, puede obtener el valor de una variable utilizando una variable de apuntador, siempre y cuando enlace la variable con dicha variable de apuntador. En otras palabras, puede leer el valor apuntando a la ubicación de memoria de la variable y utilizando el operador de indirección.

Esta sección le muestra que puede escribir un nuevo valor en la ubicación de memoria de una variable empleando un apuntador que contenga el valor izquierdo de dicha variable. El listado 11.3 presenta un ejemplo.

TECLEE **LISTADO 11.3** Modificación de valores de variables por medio de apuntadores

```
1:  /* 11L03.c: Modificación de valores por medio de apuntadores */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c,  *aptr_c;
7:
8:     c = 'A';
9:     printf("c: dirección=%p, contenido=%c\n", &c, c);
10:    aptr_c = &c;
11:       printf("aptr_c: dirección=%p, contenido=%p\n", &aptr_c, aptr_c);
12:       printf("*aptr_c => %c\n", *aptr_c);
13:    *aptr_c = 'B';
14:       printf("aptr_c: dirección=%p, contenido=%p\n", &aptr_c, aptr_c);
```

continúa

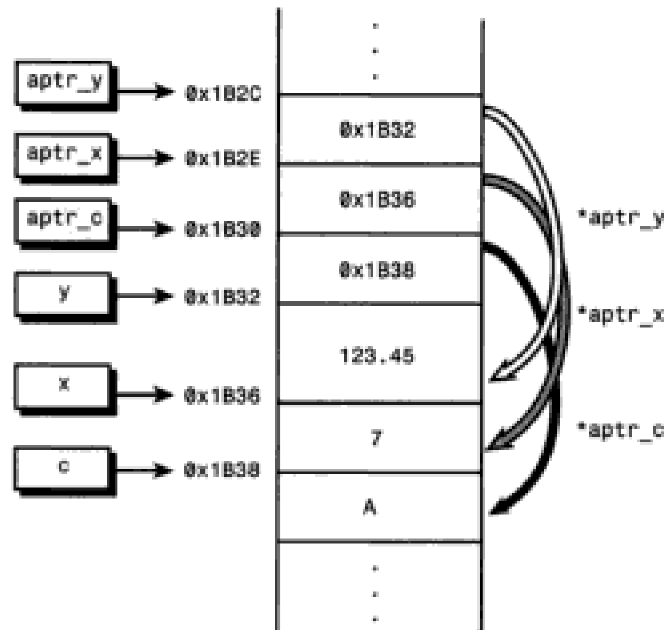


FIGURA 11.1
La imagen de memoria de las variables y sus apuntadores.

El operador de indirección (*)

Usted ha visto el asterisco (*) en la declaración de un apuntador. En C, el asterisco se llama *operador de indirección* cuando se usa como operador unario. (En ocasiones también se le llama operador de desreferenciación). Se puede hacer referencia al valor de una variable mediante la combinación del operador * y su operando, el cual contiene la dirección de la variable.

Por ejemplo, en el programa que se muestra en el listado 11.2, después de asignar la dirección de la variable de carácter `c` a la variable de apuntador `aptr_c`, la expresión `*aptr_c` hace referencia al valor contenido por `c`. Por lo tanto, para obtener el valor de `c` puede usar la expresión `*aptr_c`, en vez de usar directamente la variable `c`.

De la misma manera, dada una variable entera `x` y siendo `x = 1234`, puede, por ejemplo, declarar una variable de apuntador entera, `aptr_x`, y asignarle el valor izquierdo (dirección) de `x`, es decir, `aptr_x = &x`. Entonces, la expresión `*aptr_x` produce `1234`, que es el valor derecho (contenido) de `x`.

No confunda el operador de indirección con el operador de multiplicación, aunque ambos compartan el mismo símbolo, *.

El *operador de indirección* es un operador unario que toma sólo un operando. Dicho operando contiene la dirección (es decir, el valor izquierdo) de una variable.

Por otra parte, el *operador de multiplicación* es un operador binario que necesita dos operandos para realizar la operación de multiplicación.

El significado del símbolo * lo determina el contexto en el que lo utilice.

Materiał chroniony prawem autorskim

Después de ejecutar en mi máquina el archivo `11L02.exe`, obtuve los siguientes resultados desplegados en la pantalla:

SALIDA

```
c: dirección=0x1B38, contenido=A
x: dirección=0x1B36, contenido=7
y: dirección=0x1B32, contenido=123.45
aptr_c: dirección=0x1B30, contenido=0x1B38
*aptr_c => A
aptr_x: dirección=0x1B2E, contenido=0x1B36
*aptr_x => 7
aptr_y: dirección=0x1B2C, contenido=0x1B32
*aptr_y => 123.45
```

ANÁLISIS En el listado 11.2, hay tres variables `c`, `x` y `y`, y tres variables de apuntador, `aptr_c`, `aptr_x` y `aptr_y`, declaradas en las líneas 6 a 8, respectivamente.

Las instrucciones de las líneas 10 a 12 inicializan las variables `c`, `x` y `y`. Después, las líneas 13 a 15 imprimen las direcciones y los contenidos de las tres variables.

En la línea 16 se asigna el valor izquierdo de la variable de carácter `c` a la variable de apuntador `aptr_c`. La salida que genera la instrucción de la línea 17 muestra que la variable de apuntador `aptr_c` contiene la dirección de `c`. En otras palabras, el contenido (es decir, el valor derecho) de `aptr_c` es la dirección (es decir, el valor izquierdo) de `c`.

Luego, en la línea 18 se imprime el valor al que hace referencia el apuntador `*aptr_c`. Los resultados demuestran que el apuntador `*aptr_c` sí apunta a la ubicación de memoria de `c`.

La línea 19 asigna el valor izquierdo de la variable de tipo entero `x` a la variable de apuntador entera `aptr_x`. Las instrucciones de las líneas 20 y 21 imprimen los valores izquierdo y derecho de la variable de apuntador `aptr_x`, así como el valor al que hace referencia el apuntador `*aptr_x`.

Asimismo, en la línea 22 se asigna el valor izquierdo de la variable `y` de tipo `float` a la variable de apuntador `aptr_y` de tipo `float`. Para demostrar que `aptr_y` contiene la dirección de `y`, y que `*aptr_y` da el contenido de `y`, las líneas 23 y 24 imprimen los valores derechos de `aptr_y` y `*aptr_y`, respectivamente.

Las instrucciones de las líneas 16, 19 y 22 le muestran cómo asignar el valor de una variable a otra, en una forma indirecta. En otras palabras, el valor izquierdo de una variable se puede asignar a otra variable, para que la segunda se pueda usar como una variable de apuntador para obtener el valor derecho de la primera. En este caso, el nombre de la variable y el apuntador hacen referencia a la misma ubicación de memoria. De acuerdo con esto, si en una expresión se usa el nombre de la variable o el apuntador para modificar el contenido de la ubicación de memoria, también cambia para el otro el contenido de la ubicación de memoria.

Para ayudarle a entender la indirección de la asignación de valores, la figura 11.1 muestra la imagen de memoria de las relaciones entre `c` y `aptr_c`, `x` y `aptr_x`, así como entre `y` y `aptr_y`, con base en los resultados obtenidos en mi máquina.

La forma general de la declaración de un apuntador es

*tipo_de_datos *nombre_del_apuntador*;

Aquí, *tipo_de_datos* especifica el tipo de datos al que apunta el apuntador. *nombre_del_apuntador* es el nombre de la variable de apuntador, el cual puede ser cualquier nombre válido de variable en C.

Observe que justo antes del nombre del apuntador hay un asterisco *, lo que indica que la variable es un apuntador. Cuando el compilador ve un asterisco en la declaración, toma nota de que la variable se puede usar como un apuntador.

A continuación se muestran diferentes tipos de apuntadores:

```
char *aptr_c;     /* declara un apuntador a un carácter */

int  *aptr_int;  /* declara un apuntador a un entero */

float *aptr_flt; /* declara un apuntador a un punto flotante */
```

El programa del listado 11.2 muestra cómo declarar apuntadores y cómo asignarles valores.

TECLEE **LISTADO 11.2** Cómo declarar apuntadores y cómo asignarles valores

```
1:  /* 11L02.c: Cómo declarar apuntadores y cómo asignarles valores */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c, *aptr_c;
7:     int   x, *aptr_x;
8:     float y, *aptr_y;
9:
10:    c = 'A';
11:    x = 7;
12:    y = 123.45;
13:    printf("c: dirección=%p, contenido=%c\n", &c, c);
14:    printf("x: dirección=%p, contenido=%d\n", &x, x);
15:    printf("y: dirección=%p, contenido=%5.2f\n", &y, y);
16:    aptr_c = &c;
17:       printf("aptr_c: dirección=%p, contenido=%p\n", &aptr_c, aptr_c);
18:       printf("*aptr_c => %c\n", *aptr_c);
19:    aptr_x = &x;
20:       printf("aptr_x: dirección=%p, contenido=%p\n", &aptr_x, aptr_x);
21:       printf("*aptr_x => %d\n", *aptr_x);
22:    aptr_y = &y;
23:       printf("aptr_y: dirección=%p, contenido=%p\n", &aptr_y, aptr_y);
24:       printf("*aptr_y => %5.2f\n", *aptr_y);
25:    return 0;
26: }
```

Materiał chroniony prawem autorskim

La instrucción de la línea 10 despliega en la pantalla la dirección (el valor izquierdo) y el contenido (el valor derecho) de la variable de carácter `c`. Aquí, la expresión `&c` produce la dirección de `c`.

Observe que en la función `printf()` de la línea 10 se utiliza el especificador de formato `%p` para desplegar la dirección producida por `&c`.

De la misma manera, las líneas 11 y 12 imprimen las direcciones de `x` y `y`, así como su contenido. Puede ver en la primera parte de los resultados que las direcciones de `c`, `x` y `y` son `0x1AF4`, `0x1AF2` y `0x1AF6`. Mi computadora imprimió estas direcciones en formato hexadecimal. Sin embargo, el especificador de formato `%p` no garantiza la impresión de direcciones en formato hexadecimal, sino sólo la conversión de las direcciones a una secuencia de caracteres imprimibles. Debe consultar el manual de su compilador de C para ver qué formato esperar. Debido a que estas tres variables aún no han sido inicializadas, el contenido de sus ubicaciones de memoria corresponde a lo que había ahí como resultado de la última escritura de memoria.

Sin embargo, después de la inicialización que se lleva a cabo en las líneas 13 a 15, los espacios de memoria reservados para las tres variables contienen ahora los valores iniciales. Las líneas 16 a 18 despliegan las direcciones y contenidos de `c`, `x` y `y` después de la inicialización.

Puede ver en la segunda parte de los resultados que los contenidos de `c`, `x` y `y` ahora son `'A'`, `7` y `123.45`, respectivamente, con las mismas direcciones de memoria.

El estándar ANSI maneja el especificador de formato `%p` utilizado en la función `printf()`. Si por algún motivo su compilador no maneja `%p`, puede intentar utilizar `%u` o `%lu` en la función `printf()` para convertir e imprimir un valor izquierdo (es decir, una dirección).

Además, obtuve las direcciones impresas por los ejemplos de esta lección al ejecutar los ejemplos en mi máquina. Los valores podrían variar al ejecutar los ejemplos en su máquina. Esto se debe a que la dirección de una variable puede variar de un tipo de computadora a otro.

Declaración de apuntadores

Como mencioné al principio de esta lección, un apuntador es una variable, lo que significa que un apuntador también tiene un valor izquierdo y un valor derecho. Sin embargo, ambos valores son direcciones. El valor izquierdo de un apuntador se usa para hacer referencia al apuntador mismo, mientras que el valor derecho, que es su contenido, es la dirección de otra variable.

El listado 11.1 muestra otro ejemplo de la obtención de direcciones (es decir, valores izquierdos) de variables.

TECLEE **LISTADO 11.1** Obtención de los valores izquierdos de variables

```
1:  /* 11L01.c: Obtención de direcciones */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c;
7:     int   x;
8:     float y;
9:
10:    printf("c: dirección=%p, contenido=%c\n", &c, c);
11:    printf("x: dirección=%p, contenido=%d\n", &x, x);
12:    printf("y: dirección=%p, contenido=%5.2f\n", &y, y);
13:    c = 'A';
14:    x = 7;
15:    y = 123.45;
16:    printf("c: dirección=%p, contenido=%c\n", &c, c);
17:    printf("x: dirección=%p, contenido =%d\n", &x, x);
18:    printf("y: dirección=%p, contenido=%5.2f\n", &y, y);
19:    return 0;
20: }
```

Después de crear y ejecutar en mi computadora el archivo `11L01.exe`, se exhibieron los siguientes resultados en la pantalla:

Usted podría obtener un resultado diferente, dependiendo de su computadora y sistema operativo y, en especial, de la situación de la memoria de su computadora al momento de ejecutar el programa.

SALIDA

```
c: dirección=0x1AF4, contenido=0
x: dirección=0x1AF2, contenido=-32557
y: dirección=0x1AF6, contenido=0.00
c: dirección=0x1AF4, contenido=A
x: dirección=0x1AF2, contenido=7
y: dirección=0x1AF6, contenido=123.45
```

ANÁLISIS Como puede ver en el listado 11.1, en las líneas 6 a 8 se declaran las variables `c`, `x` y `y`, respectivamente.

Materiał chroniony prawem autorskim

Luego, cuando se le asigna un valor a la variable, éste se almacena como contenido en la ubicación reservada de memoria. El contenido también se conoce como *valor derecho* de la variable.

Por ejemplo, después de declarar la variable entera x y asignarle un valor como el siguiente

```
int x;
x = 7;
```

la variable x tiene ahora dos valores:

el valor izquierdo: 1000

el valor derecho: 7

Aquí, el valor izquierdo, 1000, es la dirección de la ubicación de memoria reservada para x. El valor derecho, 7, es el contenido almacenado en esa ubicación de memoria. Observe que, dependiendo de las computadoras y los sistemas operativos, el valor izquierdo de x puede ser diferente de una máquina a otra.

Puede imaginar que la variable x es el buzón de su casa, la cual tiene la dirección (normalmente el número de la calle) 1000. Imagine que el valor derecho, 7, es una carta entregada en el buzón.

Observe que cuando se está compilando su programa de C y se está asignando un valor a una variable, el compilador de C tiene que verificar el valor izquierdo de la variable. Si el compilador no puede encontrar el valor izquierdo, emitirá un mensaje de error que dice que la variable es una variable indefinida en su programa. Es por ello que en C tiene que declarar una variable antes de poder utilizarla. (Imagine a un cartero quejándose de que no puede depositar las cartas dirigidas a usted porque todavía no tiene un buzón.)

Utilizando el valor izquierdo de una variable, el compilador de C puede localizar con facilidad el almacenamiento adecuado de memoria reservado para una variable, y luego escribir o leer el valor derecho de la variable

El operador de dirección (&)

El lenguaje C también le proporciona un operador, &, que puede utilizar en caso de que desee saber el valor izquierdo de una variable. Este operador se llama *operador de dirección* debido a que se evalúa como la dirección (es decir, el valor izquierdo) de una variable.

Por ejemplo, el siguiente código

```
long int x;
long int *y;
y = &x;
```

asigna la dirección de la variable x de tipo long int a la variable de apuntador y. (Más adelante en este capítulo abundaremos sobre esto y sobre el significado de *y.)

Qué es un apuntador

Entender los apuntadores es vital para ser un eficaz programador en C. Hasta ahora, sólo ha tratado directamente con variables, asignándoles valores. Esta hora le presenta un nuevo concepto conocido como *indirección*.

En lugar de asignar los valores en forma directa a las variables, puede manipular indirectamente una variable creando otra denominada *apuntador*, la cual contiene la dirección de memoria de otra variable.

Pero, ¿por qué es tan importante esto? Para los principiantes, utilizar la dirección de memoria de sus datos es a menudo la forma más rápida y simple de acceder a ellos. Hay muchas cosas que sin los apuntadores son difíciles, si no es que imposibles, de realizar, como la asignación dinámica de memoria, la transferencia de grandes estructuras de datos entre funciones, e incluso hablar con el hardware de su computadora.

De hecho ya usó un apuntador en la hora 1 de este libro, "El primer paso". ¿Recuerda la cadena `"Hola, amigo. Éste es mi primer programa de C.\n"`? Una cadena es en realidad un arreglo, el cual es, por sí mismo, una clase especial de apuntador.

Más adelante le presentaré más acerca de los arreglos, la asignación de memoria y las cosas maravillosas que puede hacer con los apuntadores. Por ahora, el primer paso es entender qué es un apuntador y cómo trabajar con uno en sus programas.

A partir de la definición de un apuntador, usted sabe dos cosas: primera, que un apuntador es una variable, así que puede asignar diferentes valores a una variable de apuntador, y segunda, que el valor contenido en un apuntador debe ser una dirección que indique la ubicación de otra variable en la memoria. Es por ello que un apuntador también se conoce como *variable de dirección*.

Dirección (valor izquierdo) contra contenido (valor derecho)

Como quizá sepa, la memoria de su computadora se usa para contener el código binario de su programa, el cual consta de instrucciones y datos, así como el código binario del sistema operativo de su máquina.

Cada ubicación de memoria debe tener una dirección única para que la computadora pueda leer o escribir en ella sin ninguna confusión. Esto es parecido al concepto de que cada casa de una ciudad debe tener una dirección única.

Cuando se declara una variable, se reserva para ella una parte de memoria no utilizada, y la dirección única de la memoria se asocia al nombre de la variable. La dirección asociada con el nombre de la variable se conoce normalmente como *valor izquierdo* de la variable

Materiał chroniony prawem autorskim

HORA 11

Apuntadores

Los deberes del apuntador fueron señalar, llamando por sus nombres, a aquellos en la congregación que debían tomar nota de alguna idea expuesta en el sermón.

—H. B. Otis, Simple Truth

En las últimas 10 horas ha aprendido acerca de tipos de datos, operadores, funciones y ciclos de C, todos ellos muy importantes. En esta lección aprenderá acerca de una de las características más importantes y poderosas de C: los apuntadores. Los temas que trataremos en esta hora son los siguientes:

- Variables de apuntador
- Direcciones de memoria
- El concepto de indirección
- Declaración de un apuntador
- El operador de dirección
- El operador de indirección

En las siguientes horas se mostrarán más ejemplos de la aplicación de los apuntadores, en especial en la hora 16, "Uso de apuntadores".

Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

PARTE III

Apuntadores y arreglos

Hora

11 Apuntadores

12 Arreglos

13 Cadenas

14 Alcance y clases de almacenamiento

```
    case '/':  x /= y;
    default:   break;
}
```

3. Igual que en la pregunta 2, usando `x = 4`, `y = 2`, y `operador = '-'`, ¿cuál es el valor final de `x` después de ejecutar la siguiente instrucción `switch`?

```
switch (operador){
    case '+':  x += y; break;
    case '-':  x -= y; break;
    case '*':  x *= y; break;
    case '/':  x /= y; break;
    default:   break;
}
```

4. ¿Cuál es el valor de la variable entera `x` después de ejecutar el siguiente código?

```
x = 1;
for (i=2; i<10; i++){
    if (i%3 == 0)
        continue;
    x += i;
}
```

Ejercicios

1. Reescriba el programa del listado 10.1. Esta vez utilice la expresión lógica `i%6 == 0` en la instrucción `if`.
2. Reescriba el programa del listado 10.1 utilizando instrucciones `if` anidadas.
3. Escriba un programa que lea caracteres de la E/S estándar. Si los caracteres son `A`, `B` y `C`, despliegue sus valores numéricos en la pantalla (se necesita la instrucción `switch`).
4. Escriba un programa que siga leyendo caracteres de la entrada estándar hasta que se introduzca el carácter `q`.
5. Reescriba el programa del listado 10.7. Esta vez, en vez de saltar el `3` y el `5`, salte el entero que se pueda dividir entre `2` y entre `3`.

Materiał chroniony prawem autorskim

programa ramifica al bloque de instrucciones que están bajo la palabra reservada `else` y regresa a su pista original después de ejecutar las instrucciones que controla `else`. En otras palabras, la instrucción `if` permite ejecutar o saltar por completo un solo bloque de instrucciones, mientras que la instrucción `if-else` ejecuta uno de los dos bloques de instrucciones que están bajo su control.

P ¿Por qué necesito agregar la instrucción `break` dentro de la instrucción `switch`?

R Cuando se selecciona uno de los casos que están dentro de la instrucción `switch`, el control del programa lo ramificará y ejecutará todas las instrucciones del caso seleccionado y las de los casos que le siguen. Por lo tanto, podría obtener más resultados de los esperados. Para indicarle a la computadora que sólo ejecute las instrucciones de un caso seleccionado, puede poner una instrucción `break` al final del caso para que el control de flujo del programa salga de la estructura de `switch` después de ejecutar las instrucciones contenidas en el caso.

P ¿Qué hace la instrucción `continue` dentro de un ciclo?

R Cuando se ejecuta la instrucción `continue` dentro de un ciclo, el control del programa salta al final del ciclo para que se pueda ejecutar la instrucción `while` o `for` que tiene el control y comenzar otra iteración si aún se cumple la expresión condicional. Dentro del ciclo, se saltarán todas las instrucciones que sigan a la instrucción `continue` cada vez que se ejecute dicha instrucción.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. Dada `x = 0`, ¿se realizarán las operaciones aritméticas contenidas en la siguiente instrucción `if`?

```
if (x != 0)
   y = 123 / x + 456;
```

2. Dadas `x = 4`, `y = 2` y `operador = '-'`, ¿cuál es el valor final de `x` después de ejecutar la siguiente instrucción `switch`?

```
switch (operador){
   case '+':  x += y;
   case '-':  x -= y;
   case '*':  x *= y;
```

Resumen

En esta lección aprendió las siguientes instrucciones y palabras reservadas de ciclos y ramificación condicional importantes de C:

- Una tarea importante de un programa es instruir a la computadora para que salte a diferentes partes del código de acuerdo con las condiciones de ramificación especificadas.
- La instrucción `if` es muy importante en C para la bifurcación condicional.
- La instrucción `if` se puede anidar para formar en su programa una serie de decisiones relacionadas.
- La instrucción `if-else` es una ampliación de la instrucción `if`.
- La instrucción `switch` le ayuda a mantener más legible su programa cuando hay en su código más de dos decisiones relacionadas que tomar.
- La palabra reservada `case`, seguida de dos puntos y un valor constante integral, se usa como etiqueta en la instrucción `switch`. La etiqueta `default:` se usa al final de una instrucción `switch` cuando ningún caso corresponde a la condición.
- La instrucción `break` se puede emplear para salir de la estructura de un `switch` o de un ciclo (por lo regular de un ciclo infinito).
- La instrucción `continue` le permite permanecer dentro de un ciclo saltando al mismo tiempo algunas instrucciones.
- La instrucción `goto` le permitirá a la computadora saltar a otro punto en su código: No se recomienda el uso de esta instrucción, ya que podría hacer que su programa sea difícil de depurar y poco confiable.

En la siguiente lección aprenderá acerca de un concepto muy importante, los apuntadores.

Preguntas y respuestas

P ¿Cuántas expresiones hay en la instrucción `if`?

R La instrucción `if` toma sólo una expresión que contiene el criterio condicional. Las instrucciones que controla la instrucción `if` se ejecutan cuando la expresión se evalúa como diferente de cero (es decir, cuando se cumplen las condiciones). En caso contrario se saltan estas instrucciones y se ejecuta la instrucción que sigue al bloque controlado por `if`.

P ¿Por qué la instrucción `if-else` es una ampliación de la instrucción `if`?

R Cuando la expresión condicional de la instrucción `if` se evalúa como cero, el control de flujo del programa regresa a la pista original. Sin embargo, cuando la expresión condicional de la instrucción `if-else` se evalúa como cero, el control de flujo del

Materiał chroniony prawem autorskim

La siguiente es la forma general de la instrucción `goto`:

```
nombreetiqueta:
    instrucción1;
    instrucción2;
    .
    .
    .
    goto  nombreetiqueta;
```

Aquí, *nombreetiqueta* es un nombre de etiqueta que le indica a la instrucción `goto` a dónde saltar. Debe colocar el *nombreetiqueta* en dos lugares: uno es donde la instrucción `goto` saltará (observe que se deben poner dos punto después del nombre de la etiqueta), y el otro lugar es después de la palabra reservada `goto`.

La etiqueta a la que salta la instrucción `goto` puede aparecer antes o después de dicha instrucción.

Una de las mejores cosas que tiene C es que favorece la *programación estructurada*. Los programas deben actuar de manera predecible y su comportamiento debe ser razonablemente obvio tan sólo con leer el código fuente.

Por supuesto, otra de las mejores características de C es que no exige este modelo de perfección. Depende de usted, el programador, usar las herramientas que se le dan para escribir un código estructurado, claro, elegante y legible.

En este capítulo vimos las instrucciones `break`, `continue` y `goto`. El uso inadecuado de estas intrucciones de ramificación puede conducir a lo que se conoce como "código espagueti". (Si imprime el código fuente y dibuja líneas en la página para indicar el flujo de ejecución, terminará con un dibujo del espagueti.) Cuando la ejecución de un programa salta por ahí de manera impredecible, es muy difícil (a menudo imposible, en un proyecto extenso y complejo) determinar la intención o el comportamiento real de un programa.

El uso de instrucciones `goto` puede conducir con facilidad al código espagueti, en especial si se usa para saltar hacia atrás, o hacia fuera de instrucciones de control de flujo. Cuando vea una etiqueta al azar que simplemente está puesta en el código, sólo sabrá que algún otro código saltará ahí por medio de una instrucción `goto`, no sabrá cuándo o por qué, sin tener que detenerse a examinar el resto de la función.

Los efectos de la instrucción `continue` están, por lo menos, limitados a la instrucción de ciclo en que se usan. Sin embargo, los ciclos complejos pueden ser difíciles de descifrar, a menos que conozca exactamente cuándo y porqué terminará el ciclo.

Lo mismo se puede decir de `break`. Además de las instrucciones `switch`, se puede utilizar para saltar fuera de los ciclos `for`, `while` o `do-while`. En general, esto sólo debe hacerse para interrumpir un ciclo infinito. En caso contrario, utilice su propia expresión condicional para terminar el ciclo.

LISTADO 10.7 continuación

```
12:         suma += i;
13:     }
14:     printf("La suma de 1, 2, 4, 6 y 7 es: %d\n", suma);
15:     return 0;
16: }
```

Después de ejecutar en mi computadora el archivo `10L07.exe`, se muestra el siguiente resultado en la pantalla:

SALIDA

```
La suma de 1, 2, 4, 6 y 7 es: 20
```

ANÁLISIS Queremos calcular la suma de los valores enteros de `1`, `2`, `4`, `6` y `7` en el listado 10.7. Debido a que los enteros son casi consecutivos, se construye un ciclo `for` en las líneas 9 a 13. La instrucción de la línea 12 suma todos los enteros consecutivos del `1` al `7` (con excepción del 3 y el 5, los cuales no están en la lista y se saltan en el ciclo `for`).

Para saltar estos dos números se evalúa la expresión `(i==3) || (i==5)` de la instrucción `if` de la línea 10. Si la expresión se evalúa como `1` (es decir que el valor de `i` es igual a `3` o a `5`), entonces se ejecuta la instrucción `continue` de la línea 11, la cual hace que se salte la suma de la línea 12, y que se inicie otra iteración en la instrucción `for`. En esta forma se obtiene la suma de los valores enteros `1`, `2`, `4`, `6` y `7`, pero saltando automáticamente el 3 y el 5 mediante un ciclo `for`.

Después del ciclo `for` se despliega en la pantalla el valor de `suma`, `20`, por medio de la llamada a `printf()` de la instrucción de la línea 14.

La instrucción goto

Este libro no estaría completo sin mencionar la instrucción `goto`, aunque no le recomiendo utilizarla. La razón principal de esto es porque es probable que su uso haga que su programa de C sea poco confiable y difícil de depurar.

A menudo los programadores se ven tentados a usar la instrucción `goto`, en especial si han utilizado otros lenguajes que no tienen el rico conjunto de instrucciones condicionales de ramificación que proporciona C. El hecho es que cualquier uso de `goto` se puede evitar por completo utilizando las otras instrucciones de ramificación.

Usted debe conocer la instrucción `goto`, pero sólo para saber cómo manejarla de manera apropiada cuando la vea en el código de otra persona.

Materiał chroniony prawem autorskim

Obtuve el siguiente resultado al ejecutar en mi máquina el archivo `10L06.exe`:

SALIDA

```
Introduzca un carácter:
(introduzca x para salir)
H
I
x
El ciclo while infinito ha sido interrumpido. ¡Hasta pronto!
```

ANÁLISIS En el listado 10.6 hay un ciclo `while` infinito, el cual inicia en la línea 9 y termina en la línea 13. Dentro del ciclo infinito, los caracteres que introduce el usuario se asignan, uno a la vez, a la variable entera `c` (vea la línea 10).

La expresión relacional `c == 'x'` de la instrucción `if` se evalúa cada vez que itera el ciclo (vea la línea 11). Si la expresión se evalúa como `0` (es decir, si el usuario no introdujo la letra `x`), el ciclo continúa. En caso contrario se ejecuta la instrucción `break` de la línea 12, la cual hace que la computadora salte fuera del ciclo infinito y comience a ejecutar la siguiente instrucción, la cual se muestra en la línea 14.

Puede ver en los resultados de muestra que el ciclo `while` siguió hasta que introduje la letra `x`, lo que hizo que el ciclo infinito se interrumpiera y se desplegara en la pantalla el mensaje `El ciclo while infinito ha sido interrumpido. ¡Hasta pronto!`

La instrucción `continue`

En lugar de interrumpir un ciclo, hay ocasiones en las que usted necesita permanecer en un ciclo pero saltar sobre algunas instrucciones que están dentro de él. Para hacer esto, puede utilizar la instrucción `continue`. Esta instrucción hace que la ejecución salte de inmediato al final del ciclo.

Por ejemplo, el listado 10.7 muestra cómo usar la instrucción `continue` en un ciclo que realiza sumas.

LISTADO 10.7 Uso de la instrucción `continue`

```
1:  /* 10L07.c: Uso de la instrucción continue */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, suma;
7:
8:     suma = 0;
9:     for (i=1; i<8; i++){
10:       if ((i==3) || (i==5))
11:          continue;
```

continúa

Interrupción de un ciclo infinito

También puede utilizar la instrucción `break` para interrumpir un ciclo infinito. Un ciclo infinito es aquel en el que la expresión condicional se omite completamente; de esta manera, depende del código que está dentro del ciclo determinar las condiciones para terminarlo. Los siguientes son ejemplos de ciclos infinitos `for` y `while`:

```
for (;;){
    instrucción1;
    instrucción2;
    .
    .
    .
}

while(1) {
    instrucción1;
    instrucción2;
    .
    .
    .
}
```

El ciclo `for` de arriba omite las tres expresiones. Esto hace que la instrucción `for` ejecute siempre el ciclo. La instrucción `while` utiliza `1` como su expresión condicional, y ya que esto desde luego nunca se evalúa como `0`, dicha instrucción siempre continúa el ciclo.

El programa del listado 10.6 muestra un ejemplo del uso de la instrucción `break` en un ciclo `while` infinito.

Listado 10.6 Interrupción de un ciclo infinito

```
1:  /* 10L07.c: Interrupción de un ciclo infinito */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int c;
7:
8:     printf("Introduzca un carácter:\n(introduzca x para salir)\n");
9:     while(1)  {
10:       c = getc(stdin);
11:       if (c == 'x')
12:          break;
13:    }
14:    printf("El ciclo while infinito ha sido interrumpido. ¡Hasta pronto!\n");
15:    return 0;
16: }
```

Materiał chroniony prawem autorskim

```
20:             break;
21:          case '4':
22:             printf("El día 4 es miércoles.\n");
23:             break;
24:          case '5':
25:             printf("El día 5 es jueves.\n");
26:             break;
27:          case '6':
28:             printf("El día 6 es viernes.\n");
29:             break;
30:          case '7':
31:             printf("El día 7 es sábado.\n");
32:             break;
33:          default:
34:             printf("El entero no pertenece al rango de 1 a 7.\n");
35:             break;
36:       }
37:       return 0;
38: }
```

Con la ayuda de la instrucción `break` puedo ejecutar el archivo `10L05.exe` y obtener sólo los resultados del caso seleccionado:

SALIDA

```
Introduzca un entero para un día
(dentro del rango de 1 a 7):
1
El día 1 es domingo.
```

ANÁLISIS Este programa tiene siete etiquetas `case` seguidas de las expresiones constantes `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'` y `'7'`, respectivamente (vea las líneas 12, 15, 18, 21, 24, 27 y 30).

En cada caso hay una instrucción a la que le sigue una instrucción `break`. Como ya mencioné, las instrucciones `break` saldrán de la estructura del `switch` antes de que la computadora llegue siquiera a la siguiente etiqueta `case` y a las instrucciones que le siguen.

Por ejemplo, después de que a la variable `día` de tipo `int` se le asigna el valor de `1` y se evalúa en la instrucción `switch`, se selecciona el caso con `'1'`, y se ejecuta la instrucción de la línea 13. Después se ejecuta la instrucción `break` de la línea 14, la cual interrumpe el control de la instrucción `switch` y lo regresa a la siguiente instrucción que está fuera de la estructura del `switch`. En el listado 10.5, la instrucción que sigue es la instrucción `return` de la línea 37, la cual finaliza la función `main`.

La llamada a la función `printf()` de la línea 13 imprime en la pantalla la cadena `El día 1 es domingo.`

Observe que en una instrucción `switch` no se necesitan las llaves para agrupar las instrucciones dentro de un caso individual, ya que `case` es sólo una etiqueta y no controla las instrucciones que le siguen. Éstas simplemente se ejecutan en orden, comenzando con la etiqueta.

```
(dentro del rango de 1 a 3):
1
Día 1
Día 2
Día 3
```

Puede ver en los resultados que se ejecuta la instrucción controlada por el caso seleccionado, `case '1'`, así como las instrucciones controladas por el resto de los casos, ya que se despliegan `Día 1`, `Día 2` y `Día 3` en la pantalla. De la misma manera, si introduzco `2` desde mi teclado, se mostrarán `Día 2` y `Día 3` en la pantalla.

Ésta es una característica importante de la instrucción `switch`: la computadora continúa ejecutando las instrucciones que están después del caso seleccionado hasta el final de dicha instrucción.

En la siguiente sección aprenderá cómo salir antes en la ejecución de la instrucción `switch`.

La instrucción break

Si desea salir por completo de la instrucción `switch` después de cada etiqueta `case`, puede agregar una instrucción `break` al final de la lista de instrucciones que sigue a cada etiqueta `case`. La instrucción `break` simplemente sale de la instrucción `switch` y continúa la ejecución después del bloque de instrucciones `switch`.

El programa del listado 10.5 se asemeja al del listado 10.4, pero esta vez se usa la instrucción `break` y los resultados son diferentes.

LISTADO 10.5 Incorporación de la instrucción break

```
1:  /* 10L05.c Incorporación de la instrucción break */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int dia;
7:
8:     printf("Introduzca un entero para un día\n");
9:     printf("(dentro del rango de 1 a 7):\n");
10:    dia = getchar();
11:    switch (dia){
12:       case '1':
13:          printf("El día 1 es domingo.\n");
14:          break;
15:       case '2':
16:          printf("El día 2 es lunes.\n");
17:          break;
18:       case '3':
19:          printf("El día 3 es martes.\n");
```

Materiał chroniony prawem autorskim

```
3:
4:  main()
5:  {
6:     int dia;
7:
8:     printf("Introduzca un entero para un dia\n");
9:     printf("(dentro del rango de 1 a 3):\n");
10:    dia = getchar();
11:    switch (dia){
12:       case '1':
13:          printf("Día 1\n");
14:       case '2':
15:          printf("Día 2\n");
16:       case '3':
17:          printf("Día 3\n");
18:       default:
19:          ;
20:    }
21:    return 0;
22: }
```

Si ejecuto el archivo `10L04.exe` e introduzco un `3`, obtendré los siguientes resultados:

SALIDA

```
Introduzca un entero para un día
(dentro del rango de 1 a 3):
3
Día 3
```

ANÁLISIS Como puede ver, en la línea 6 se declara una variable `int`, `dia`; en la línea 10 se le asigna la entrada que introduce el usuario.

En la instrucción `switch` de la línea 11 se evalúa el valor de la variable entera `dia`. Si el valor es igual a uno de los valores de las expresiones constantes, la computadora comienza a ejecutar instrucciones a partir de ahí. Las expresiones constantes se etiquetan anteponiéndoles el prefijo `case`.

Por ejemplo, yo introduje `3` y después oprimí la tecla Entrar. En la línea 10 se asigna a `dia` el valor numérico de `3`. Luego, después de encontrar un caso en el que el valor de la expresión constante coincida con el valor contenido en `dia`, la computadora salta a la línea 17 para ejecutar la función `printf()` y desplegar `Día 3` en la pantalla.

Observe que en la línea 19, debajo de la etiqueta `default` del listado 10.4, hay una instrucción vacía (es decir, nula) que termina con un punto y coma. La computadora no hace nada con una instrucción vacía. Esto significa que si no se aplica ninguna de las expresiones constantes, entonces la instrucción `switch` termina sin realizar acción alguna.

Sin embargo, si introduzco `1` desde mi teclado y después oprimo la tecla Entrar durante la ejecución del archivo `10L04.exe`, obtendré los siguientes resultados:

```
Introduzca un entero para un día
```

```
        instrucción2;

    .
    .
    .
    default:
        instrucción_predeterminada;
}
```

Aquí, primero se evalúa la expresión condicional `expresión`. La instrucción `switch` salta después a la etiqueta `case` correspondiente y ejecuta, en orden, todas las instrucciones que siguen. Si `expresión` produce un valor igual a `expresión_constante1`, entonces se ejecuta `instrucción1`, seguida de `instrucción2` y de todas las demás hasta `instrucción_predeterminada`. Si el valor que produjo `expresión` es el mismo que el valor de `expresión_constante2`, entonces `instrucción2` se ejecuta primero. Sin embargo, si el valor de `expresión` no es igual a ninguno de los valores de las expresiones constantes etiquetadas por la palabra reservada `case`, entonces se ejecuta la instrucción (`instrucción_predeterminada`) que sigue a la palabra reservada `default`.

Debe utilizar la palabra reservada `case` para etiquetar cada caso. Observe que cada etiqueta `case` termina con *dos puntos*, no con punto y coma. Ésta es la sintaxis de las etiquetas en C. La palabra reservada `default` se debe usar para el caso "predeterminado", es decir, cuando ninguna etiqueta `case` coincide con la expresión condicional. Observe también que dentro de la instrucción `switch` ninguna de las expresiones constantes asociadas con etiquetas `case` puede ser idéntica.

En C, una *etiqueta* se usa en su código como una especie de marcador para que la utilice la instrucción condicional de ramificación. Una etiqueta no es, en sí misma, una instrucción; más bien le señala un lugar al cual saltar cuando necesita apartarse del flujo de ejecución normal de arriba hacia abajo.

La sintaxis adecuada para una etiqueta es un identificador único seguido de dos puntos, no de un punto y coma. En este capítulo verá varias formas de usar etiquetas, así como los nombres de etiqueta reservados `case expresion:` y `default:`.

El programa del listado 10.4 presenta un ejemplo del uso de la instrucción `switch`. El programa también muestra una característica importante de dicha instrucción.

LISTADO 10.4 Uso de la instrucción `switch`

```
1:  /* 10L04.c Uso de la instrucción switch */
2:  #include <stdio.h>
```

Materiał chroniony prawem autorskim

Primero se tiene que tomar una decisión con base en la evaluación de la expresión relacional `i > 0` de la instrucción `if` de la línea 9. La expresión `i > 0` se usa para comprobar si el valor de `i` es positivo o tiene otro valor (enteros negativos, incluyendo al cero). Si la expresión se evalúa como `1`, la computadora salta a la segunda instrucción `if` (es decir, a la anidada) de la línea 10.

Observe que la línea 10 contiene otra expresión relacional, `i%2 == 0`, la cual comprueba si la variable entera `i` es par o non. Por lo tanto, la segunda decisión de desplegar enteros pares o nones se toma de acuerdo con la evaluación de la segunda expresión relacional, `i%2 == 0`. Si esta evaluación produce `1`, la llamada a `printf()` de la línea 11 imprime un entero par. En caso contrario se ejecuta la instrucción de la línea 13 y se muestra un entero non en la pantalla.

La computadora ramifica a la línea 14 si la expresión `i > 0` de la línea 9 se evalúa como `0`; es decir, si el valor de `i` es menor que 0. En la línea 14 se anida otra instrucción `if` dentro de una frase `else`, y se evalúa la expresión relacional `i == 0`. Si `i == 0` se evalúa como verdadero lógicamente, lo que significa que `i` sí contiene el valor de cero, se despliega en la pantalla la cadena `El entero es cero`. En caso contrario, el valor de `i` debe ser negativo, de acuerdo con la evaluación previa de la expresión `i > 0` de la línea 9. La instrucción de la línea 17 imprime entonces el entero negativo en la salida estándar.

Como puede ver en el ejemplo, el valor de `i` está dentro del rango de `5` a `-5`. Debido a esto, `-5`, `-4`, `-3`, `-2` y `-1` se imprimen como enteros negativos. Después se imprime un mensaje cuando `i` es cero, y luego se imprimen los enteros nones `1`, `3` y `5`, así como los enteros pares `2` y `4`.

La instrucción switch

En la sección anterior vio que se utilizan las instrucciones `if` anidadas cuando hay que tomar decisiones sucesivas relacionadas. Sin embargo, la instrucción `if` anidada se puede volver muy compleja si es necesario tomar muchas decisiones. En ocasiones, un programador tendrá problemas para simplemente seguirles la pista a algunas instrucciones `if` anidadas.

Por fortuna existe otra instrucción en C, la instrucción `switch`, misma que usted puede usar para tomar decisiones u opciones ilimitadas con base en el valor de una expresión condicional y casos específicos.

La forma general de la instrucción `switch` es

```
switch (expresión) {
   case expresión_constante1:
         instrucción1;

   case expresión_constante2:
```

Instrucciones `if` anidadas

Como vio en las secciones anteriores, una instrucción `if` le permite a un programa tomar una decisión. En muchos casos, un programa tiene que tomar una serie de decisiones relacionadas. Para este fin, usted puede utilizar instrucciones `if` anidadas.

El listado 10.3 muestra cómo usar instrucciones `if` anidadas.

LISTADO 10.3 Uso de instrucciones `if` anidadas

```
1:  /* 10L03.c Uso de instrucciones if anidadas */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     for (i=-5; i<=5; i++){
9:        if (i > 0)
10:           if (i%2 == 0)
11:              printf("%d es un entero par.\n", i);
12:           else
13:              printf("%d es un entero non.\n", i);
14:       else if (i == 0)
15:              printf("El entero es cero.\n");
16:            else
17:              printf("Entero negativo: %d\n", i);
18:    }
19:    return 0;
20: }
```

Después de ejecutar el archivo `10L03.exe`, obtuve los siguientes resultados:

SALIDA

```
Entero negativo: -5
Entero negativo: -4
Entero negativo: -3
Entero negativo: -2
Entero negativo: -1
El entero es cero.
1 es un entero non.
2 es un entero par.
3 es un entero non.
4 es un entero par.
5 es un entero non.
```

ANÁLISIS El listado 10.3 contiene un ciclo `for` que comienza en la línea 8 y termina en la 18. De acuerdo con las expresiones de la instrucción `for` de la línea 8, toda tarea controlada por la instrucción `for` se ejecuta hasta 11 veces.

Materiał chroniony prawem autorskim

LISTADO 10.2 Uso de la instrucción `if-else`

```
1:  /* 10L02.c Uso de la instrucción if-else */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     printf("Entero par   Entero non\n");
9:     for (i=0; i<10; i++)
10:       if (i%2 == 0)
11:          printf("%d", i);
12:       else
13:          printf("%14d\n", i);
14:
15:    return 0;
16: }
```

Al ejecutar el archivo `10L02.exe` se obtienen los siguientes resultados:

SALIDA

```
Entero par          Entero non
0                    1
2                    3
4                    5
6                    7
8                    9
```

ANÁLISIS La línea 6 del listado 10.2 declara una variable entera, `i`. La función `printf()` de la línea 8 despliega una línea de encabezado en la pantalla.

La variable entera `i` se inicializa en la primera expresión de la instrucción `for` de la línea 9. Controlada por la instrucción `for`, la instrucción `if-else` de las líneas 10 a 13 se ejecuta 10 veces. De acuerdo con la instrucción `if-else`, la llamada a `printf()` de la línea 11 imprime enteros pares si la expresión relacional `i%2 == 0` de la línea 10 se evalúa como `1` (verdadero lógicamente). Si dicha expresión se evalúa como `0` (falso lógicamente), la llamada a `printf()` de la línea 13, controlada por la palabra reservada `else`, imprime enteros nones en la salida estándar.

Debido a que se trata a la instrucción `if-else` como a una sola instrucción, no se necesitan las llaves `{` y `}` para formar un bloque de instrucciones bajo la instrucción `for`. De la misma manera, no se usan llaves en la instrucción `if-else`, ya que las palabras reservadas `if` y `else` controlan, cada una, una sola instrucción en las líneas 11 y 13 respectivamente.

Observe que en la función `printf()` de la línea 13 se especifica un ancho mínimo de `14`, de modo que los resultados de los enteros nones se listan a la derecha de los enteros pares, como puede ver en la sección de salida. El programa del listado 10.2 verifica los enteros en un rango del `0` al `9`, y muestra que `0`, `2`, `4`, `6` y `8` son enteros pares y que, `1`, `3`, `5`, `7` y `9` son nones.

Dentro del ciclo `for`, la instrucción `if` de las líneas 11 y 12 evalúa la expresión lógica `(i%2 == 0) && (i%3 == 0)`. Si la expresión se evalúa como `1` (es decir, el valor de `i` se puede dividir por completo entre 2 y entre 3), entonces se despliega en la pantalla el valor de `i` llamando a la función `printf()` de la línea 12. En caso contrario se salta la instrucción de la línea 12.

Observe que no se usan las llaves (`{` y `}`), ya que sólo hay una instrucción bajo el control de la instrucción `if`.

El resultado que se muestra en la pantalla presenta todos los enteros dentro del rango de `0` a `100` que se pueden dividir entre `2` y entre `3`.

La instrucción if-else

En la instrucción `if`, cuando la expresión condicional se evalúa como diferente de cero, la computadora pasará a las instrucciones que controla la instrucción `if` y las ejecutará de inmediato. Si la expresión se evalúa como cero, la computadora ignorará las instrucciones que controla la instrucción `if`.

A menudo necesitará que la computadora ejecute un conjunto alterno de instrucciones cuando la expresión condicional de la instrucción `if` se evalúe como falso lógicamente. Para ello, puede usar otra instrucción condicional de ramificación: la instrucción `if-else`.

Como una extensión de la instrucción `if`, la instrucción `if-else` tiene la siguiente forma:

```
if (expresión) {
   instrucción1;
   instrucción2;
   .
   .
   .
}
else {
   instrucción_A;
   instrucción_B;
   .
   .
   .
}
```

Si *`expresión`* se evalúa como diferente de cero, entonces se ejecutan las instrucciones que controla el `if`, incluyendo a *`instrucción1`* e *`instrucción2`*. Sin embargo, si *`expresión`* se evalúa como cero, en su lugar se ejecutan *`instrucción _A`* e *`instrucción_B`*, que siguen a la palabra reservada `else`.

El programa del listado 10.2 muestra cómo usar la instrucción `if-else`.

Materiał chroniony prawem autorskim

`x > 0.0`, la cual se evalúa a un valor de `1` (verdadero lógicamente) si `x` es mayor que cero, y se evalúa como `0` (falso lógicamente) si `x` es menor o igual a cero.

El listado 10.1 proporciona otro ejemplo del uso de la instrucción `if`.

Listado 10.1 Uso de la instrucción `if` en la toma de decisiones

```
1:  /* 10L01.c Uso de la instrucción if */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     printf("Enteros que se pueden dividir entre 2 y entre 3\n");
9:     printf("(dentro del rango de 0 a 100):\n");
10:    for (i=0; i<=100; i++){
11:       if ((i%2 == 0) && (i%3 == 0))
12:         printf("   %d\n", i);
13:    }
14:    return 0;
15: }
```

Después de crear y ejecutar el programa `10L01.exe` del listado 10.1, se desplegaron los siguientes resultados en la pantalla:

Salida

```
Enteros que se pueden dividir entre 2 y entre 3
(dentro del rango de 0 a 100):
   0
   6
   12
   18
   24
   30
   36
   42
   48
   54
   60
   66
   72
   78
   84
   90
   96
```

Análisis Como puede ver en el listado 10.1, la línea 6 declara una variable entera, `i`. Las líneas 8 y 9 imprimen dos líneas de encabezado. Comenzando en la línea 10, la instrucción `for` realiza un ciclo `101` veces.

Cómo decir siempre "si..."

Si la vida fuera una línea recta, sería muy aburrida. Esto también es cierto para la programación. Sería demasiado aburrido si las instrucciones sólo se pudieran ejecutar en el orden en que aparecen.

De hecho, una tarea importante de un programa consiste en instruir a la computadora para que *ramifique* (es decir, que salte) a diferentes partes del código y trabaje sobre diferentes tareas cuando se cumplan las condiciones especificadas.

Sin embargo, en la mayoría de los casos usted no sabe con anticipación qué vendrá después. Lo que sí sabe es que con toda seguridad algo sucederá si se cumplen ciertas condiciones. Por lo tanto, puede escribir en el programa solamente tareas y condiciones. Las instrucciones condicionales de ramificación deciden cuándo realizar las tareas.

La instrucción `if` es la instrucción condicional de ramificación más común en C; se puede utilizar para evaluar las condiciones así como para decidir si se va a ejecutar el bloque de código controlado por la instrucción.

La forma general de la instrucción `if` es

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

Aquí, *`expresión`* es el criterio condicional. Si la *`expresión`* se evalúa como un valor diferente de cero, entonces se ejecutan las instrucciones que están dentro de las llaves (`{` y `}`), como *`instrucción1`* e *`instrucción2`*. Si *`expresión`* se evalúa como cero, entonces se saltan las instrucciones.

Observe que las llaves (`{` y `}`) forman un bloque de instrucciones que está bajo el control de la instrucción `if`. Las llaves se pueden omitir si sólo hay una instrucción dentro del bloque. Sin embargo, siempre se deben usar los paréntesis (`(` y `)`) para encerrar la expresión condicional.

Por ejemplo, la expresión

```
if (x > 0.0)
  printf("La raiz cuadrada de x es: %f\n", sqrt(x));
```

le indica a la computadora que si el valor de la variable de punto flotante `x` es mayor que `0.0` (es decir, positivo), debe calcular la raíz cuadrada de `x` llamando a la función `sqrt()`, e imprimir después el resultado. Aquí, el criterio condicional es la expresión relacional

Material chroniony prawem autorskim

HORA 10

Control de flujo del programa

Es más difícil mandar que obedecer.

—F. Nietzsche

En la hora 7, "Ciclos", aprendió a usar las instrucciones `while`, `do-while` y `for` para hacer lo mismo una y otra vez. Estas instrucciones se pueden agrupar dentro de la categoría de instrucciones *de ciclos* que se emplean en C para el *control de flujo*.

En esta lección aprenderá acerca de las instrucciones que pertenecen a otro grupo de instrucciones de control de flujo, *la ramificación* (*o salto*) *condicional*, como son:

- La instrucción `if`
- La instrucción `if-else`
- La instrucción `switch`
- La instrucción `break`
- La instrucción `continue`
- La instrucción `goto`

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. Dada una variable `x` de tipo `int` y una variable `y` de tipo `unsigned int`, siendo `x = 0x8765` y `y = 0x8765`, y empleando como bit de signo el que está más a la izquierda, ¿es `x` igual a `y`?
2. ¿Qué debe hacer si intenta asignar un valor grande a una variable `int`, pero el valor que asigna es demasiado grande y obtiene un número negativo que no esperaba?
3. ¿Qué especificador de formato se debe emplear para especificar una variable `unsigned long int`, `%ld` o `%lu`?
4. ¿Qué nombre de archivo de encabezado debe incluir si llama a alguna función matemática en su programa de C?

Ejercicios

1. Dadas las instrucciones

   ```
   int x;
   unsigned int y;
   x = 0xAB78;
   y = 0xAB78;
   ```

   escriba un programa para desplegar en la pantalla los valores decimales de `x` y `y`.
2. Escriba un programa para medir en su máquina el tamaño de `short int`, `long int` y `long double`.
3. Escriba un programa para multiplicar dos variables `signed int` con valores positivos, y desplegar el resultado como `long int`.
4. Escriba un programa para mostrar enteros negativos en formato hexadecimal junto con sus equivalentes `signed int`.
5. Dado un ángulo de 30 grados, escriba un programa para calcular los valores de su seno y su tangente.
6. Escriba un programa para calcular la raíz cuadrada no negativa de `0x19A1`.

Materiał chroniony prawem autorskim

- Existe un conjunto de funciones de biblioteca de C, como `sin()`, `cos()` y `tan()`, que se puede emplear para realizar cálculos trigonométricos o hiperbólicos.
- Existe otro grupo de funciones matemáticas en C, como `pow()`, que puede efectuar cálculos exponenciales y logarítmicos.
- La función `sqrt()` devuelve una raíz cuadrada no negativa. La expresión `sqrt(x)` equivale a la expresión `pow(x, 0.5)`, si `x` tiene un valor no negativo. No puede pasar un valor negativo a la función `sqrt()`, ya que esto ocasionaría un error.
- Debe incluir en su programa de C el archivo de encabezado `math.h` si llama a cualquiera de las funciones matemáticas declaradas en ese archivo de encabezado.

En la siguiente lección aprenderá varias instrucciones de control de flujo muy importantes en C.

Preguntas y respuestas

P ¿Qué bit se puede usar como bit de signo en un entero?

R Se puede usar el bit más a la izquierda, el más significativo. Por ejemplo, suponga que el tipo de datos `int` tiene 16 bits de longitud. Si cuenta la posición de los bits de derecha a izquierda, comenzando por el bit 0, entonces el bit 15 es el bit más hacia la izquierda, y es el que se puede utilizar como bit de signo.

P ¿Qué hace el especificador de formato `%lu`?

R El especificador de formato `%lu` se puede usar en una cadena de `printf()` para convertir el argumento correspondiente al tipo de datos `unsigned long int`. Además, el especificador de formato `%lu` equivale a `%Lu`.

P ¿Cuándo se utilizan `short` y `long`?

R Si necesita ahorrar espacio de memoria, y sabe que el valor de una variable de datos entera permanece dentro de un rango reducido, puede utilizar el modificador `short` para indicarle al compilador de C que reduzca el espacio de memoria predeterminado asignado a la variable, por ejemplo, de 32 a 16 bits.

Por otra parte, si una variable tiene que contener un número que exceda el rango actual de un tipo de datos, puede usar el modificador `long` para aumentar el espacio de almacenamiento de la variable, a fin de que contenga ese número.

P ¿La función `sin()` toma un valor en grados o en radianes?

R Al igual que otras funciones trigonométricas de C, la función `sin()` toma un valor en radianes. Si tiene un ángulo en grados, debe convertirlo a radianes. La fórmula es:

```
radianes = grados * (3.141593 / 180.0).
```

La función `pow()` de la línea 12 toma `x` y `y`, para calcular después el valor de `x` elevado a la potencia `y`. Debido a que el resultado es de tipo `double`, como ya sé que la parte fraccionaria será toda de dígitos decimales `0`, utilizo el especificador de formato `%7.0f` en la función `printf()` para convertir sólo la parte no fraccionaria del valor. El resultado se muestra en la pantalla como `262144`.

En la línea 13 se calcula la raíz cuadrada no negativa de `x` por medio de la llamada a la función `sqrt()`. Al igual que en la línea 12, en la línea 13 se usa el especificador de formato `%2.0f` para convertir la parte no fraccionaria del valor que devuelve la función `sqrt()`, debido a que la parte fraccionaria consta solamente de ceros. Como puede ver en los resultados, la raíz cuadrada no negativa de `x` es `8`.

Como mencioné antes, la expresión `pow(x, 0.5)` es equivalente a la expresión `sqrt(x)`. Por lo tanto, no es una sorpresa que la expresión `pow(x, z)` de la instrucción de la línea 14 produzca el mismo resultado que `sqrt(x)` en la línea 13.

Todos los cálculos de punto flotante, incluyendo los tipos de datos `float` y `double`, se realizan en aritmética de `doble` precisión. Esto significa que una variable de datos `float` se debe convertir a `double` para realizar el cálculo. Después del cálculo, el tipo `double` se tiene que convertir de nuevo a `float` antes de asignar el resultado a la variable `float`. Por lo tanto, un cálculo `float` podría tardar más tiempo.

La principal razón por la que C maneja el tipo de datos `float` es el ahorro de espacio de memoria, ya que el tipo de datos `double` ocupa el doble de memoria que el tipo `float`. En muchos casos, la precisión que ofrece el tipo `float` es suficiente.

Resumen

En esta lección aprendió los siguientes modificadores y funciones matemáticas importantes:

- El modificador `signed` se puede utilizar para declarar tipos de datos `char` e `int` que contengan valores negativos y no negativos.
- Todas las variables `int` de C tienen signo de manera predeterminada.
- El modificador `unsigned` se puede utilizar para declarar tipos de datos `char` e `int` que no contengan valores negativos. Hacer esto duplica efectivamente los valores positivos que puede contener la variable.
- El espacio en memoria que ocupa una variable de datos se puede reducir e incrementar por medio de los modificadores `short` y `long`, respectivamente.

Materiał chroniony prawem autorskim

SINTAXIS

La sintaxis de la función `pow()` es

```
#include <math.h>
double pow(double x, double y);
```

Aquí, el valor de la variable `double x` se eleva a la potencia de `y`. La función `pow()` devuelve el resultado en el tipo de datos `double`.

SINTAXIS

La sintaxis de la función `sqrt()` es

```
#include <math.h>
double sqrt(double x);
```

Aquí, la función `sqrt()` devuelve la raíz cuadrada no negativa de `x` en el tipo de datos `double`. Si `x` es negativa, ocurre un error.

Si usted pasa `0.5` como segundo argumento a la función `pow()`, y `x` contiene un valor no negativo, entonces las dos expresiones, `pow(x, 0.5)` y `sqrt(x)`, son equivalentes.

Observe ahora cómo llamar a las funciones `pow()` y `sqrt()` en el programa que se muestra en el listado 9.5.

LISTADO 9.5 Aplicación de las funciones `pow()` y `sqrt()`

```
1:  /* 09L05.c: Uso de las funciones pow() y sqrt() */
2:  #include <stdio.h>
3:  #include <math.h>
4:
5:  main()
6:  {
7:     double x, y, z;
8:
9:     x = 64.0;
10:    y = 3.0;
11:    z = 0.5;
12:    printf("pow(64.0, 3.0) devuelve: %7.0f\n", pow(x, y));
13:    printf("sqrt(64.0) devuelve:     %2.0f\n", sqrt(x));
14:    printf("pow(64.0, 0.5) devuelve: %2.0f\n", pow(x, z));
15:    return 0;
16: }
```

Después de ejecutar el archivo `09L05.exe`, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
pow(64.0, 3.0) devuelve: 262144
sqrt(64.0) devuelve:     8
pow(64.0, 0.5) devuelve: 8
```

ANÁLISIS

En las líneas 9 a 11 se inicializan las variables `x`, `y` y `z` del listado 9.5 con `64.0`, `3.0` y `0.5`, respectivamente.

LISTADO 9.4 continuación

```
 5:  main()
 6:  {
 7:     double x;
 8:
 9:     x = 45.0;                  /* 45 grados */
10:     x *= 3.141593 / 180.0;     /* convierte a radianes */
11:     printf("El seno de 45 es:    %f.\n", sin(x));
12:     printf("El coseno de 45 es:  %f.\n", cos(x));
13:     printf("La tangente de 45 es: %f.\n", tan(x));
14:     return 0;
15: }
```

Al ejecutar el archivo `09L04.exe` se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
El seno de 45 es:    0.707107.
El coseno de 45 es:  0.707107.
La tangente de 45 es: 1.000000.
```

ANÁLISIS Observe que en la línea 3 se incluye el archivo de encabezado `math.h`, el cual es necesario para las funciones matemáticas de C.

La variable `x` de tipo `double` del listado 9.4 se inicializa con `45.0` en la línea 9. Aquí, `45.0` es el valor del ángulo en grados, el cual se convierte al valor correspondiente en radianes en la línea 10.

Después, la instrucción de la línea 11 calcula el seno de `x` por medio de la llamada a la función `sin()` e imprime el resultado en la pantalla. Asimismo, la línea 12 obtiene el coseno de `x` y lo muestra en la pantalla. Debido a que `x` contiene el valor de un ángulo de 45 grados, no es una sorpresa que los valores del seno y el coseno sean iguales, alrededor de `0.707107`.

La línea 13 nos da el valor de la tangente de `x` por medio de la función `tan()`. Como tal vez sepa, la tangente de `x` es igual al seno de `x` dividido entre el coseno de `x`. Debido a que el seno de un ángulo de 45 grados es igual al coseno del mismo ángulo, la tangente es igual a 1. El resultado (en el formato de punto flotante) de `1.000000`, en la tercera línea de la salida del programa, lo demuestra.

Para simplificar las cosas, podría declarar una variable `PI` inicializada a `3.141593`, y otra variable inicializada a `180.0`, y usarlas en sus cálculos. O podría simplemente declarar una sola constante inicializada al resultado de `3.141593/180.0`.

Llamadas a `pow()` y `sqrt()`

Las funciones `pow()` y `sqrt()` son otras funciones matemáticas útiles de C. A diferencia de otros lenguajes, C no tiene un operador intrínseco para elevar un número a una potencia.

Materiał chroniony prawem autorskim

Las dos secciones siguientes presentan varias funciones matemáticas y le muestran cómo usarlas en sus programas.

Llamadas a `sin()`, `cos()` y `tan()`

Si no es un fanático de las matemáticas, puede saltarse las dos secciones que siguen, pues éstas no son vitales para entender el lenguaje C. Sin embargo, si necesita hacer cálculos matemáticos, podría apreciar que C le ofrece este conjunto de funciones matemáticas.

Por ejemplo, dado un ángulo x en radianes, la expresión `sin` devuelve el seno del ángulo.

Puede utilizar la siguiente fórmula para convertir el valor de un ángulo en grados al valor en radianes:

```
radianes = grados * (3.141593 / 180.0).
```

Aquí, `3.141593` es el valor aproximado de pi. Si es necesario, puede usar más dígitos decimales de pi.

Veamos ahora la sintaxis de las funciones `sin()`, `cos()` y `tan()`.

SINTAXIS

La sintaxis de la función `sin()` es

```
#include <math.h>
double sin(double x);
```

Aquí, la variable `x` de tipo `double` contiene el valor de un ángulo en radianes. La función `sin()` devuelve el seno de `x` en el tipo de datos `double`.

SINTAXIS

La sintaxis de la función `cos()` es

```
#include <math.h>
double cos(double x);
```

Aquí, la variable `x` de tipo `double` contiene el valor de un ángulo en radianes. La función `cos()` devuelve el coseno de `x` en el tipo de datos `double`.

SINTAXIS

La sintaxis de la función `tan()` es

```
#include <math.h>
double tan(double x);
```

Aquí, la variable `x` de tipo `double` contiene el valor de un ángulo en radianes. La función `tan()` devuelve la tangente de `x` en el tipo de datos `double`.

El listado 9.4 muestra cómo utilizar las funciones `sin()`, `cos()` y `tan()`.

LISTADO 9.4 Cálculo de valores trigonométricos con `sin()`, `cos()` y `tan()`

```
1:  /* 09L04.c: Uso de las funciones sin(), cos() y tan() */
2:  #include <stdio.h>
3:  #include <math.h>
4:
```

continúa

Listado 9.3 continuación

```
 7:     unsigned int      y;
 8:     long int          s;
 9:     unsigned long int t;
10:
11:     x = 0xFFFF;
12:     y = 0xFFFFU;
13:     s = 0xFFFFFFFFl;
14:     t = 0xFFFFFFFFL;
15:     printf("El valor short int de 0xFFFF es %hd.\n", x);
16:     printf("El valor unsigned int de 0xFFFF es %u.\n", y);
17:     printf("El valor long int de 0xFFFFFFFF es %ld.\n", s);
18:     printf("El valor unsigned long int de 0xFFFFFFFF es %lu.\n", t);
19:     return 0;
20: }
```

Después de crear y ejecutar en mi máquina el archivo `09L03.exe`, se desplegaron en la pantalla los siguientes resultados:

Salida

```
El valor short int de 0xFFFF es -1.
El valor unsigned int de 0xFFFF es 65535.
El valor long int de 0xFFFFFFFF es -1.
El valor unsigned long int de 0xFFFFFFFF es 4294967295
```

Análisis Hay cuatro tipos de datos declarados en el listado 9.3: la variable `x` de tipo `short int`, la variable `y` de tipo `unsigned int`, la variable `s` de tipo `long int`, y la variable `t` de tipo `unsigned long int`. Estas cuatro variables se declaran en las líneas 6 a 9 y se inicializan en las líneas 11 a 14.

Para imprimir los valores decimales de `x`, `y`, `s` y `t`, se utilizan los especificadores de formato `%hd`, `%u`, `%ld` y `%lu` en las líneas 15 a 18, respectivamente, para convertir los correspondientes números hexadecimales a números decimales. Los resultados del programa del listado 9.3 muestran que los valores contenidos por `x`, `y`, `s` y `t` se desplegaron correctamente en la pantalla.

Funciones matemáticas en C

Básicamente, las funciones matemáticas que proporciona el lenguaje C se pueden clasificar en tres grupos:

- Funciones trigonométricas e hiperbólicas, como `acos()`, `cos()` y `cosh()`.
- Funciones exponenciales y logarítmicas, como `exp()`, `pow()` y `log10()`.
- Funciones matemáticas diversas, como `ceil()`, `fabs()` y `floor()`.

Tiene que incluir el archivo de encabezado `math.h` en su programa de C, antes de poder utilizar cualquiera de las funciones matemáticas definidas en ese archivo de encabezado.

Materiał chroniony prawem autorskim

ANÁLISIS En el listado 9.2 se emplean el operador `sizeof` y la función `printf()` para medir el tamaño de los tipos de datos modificados y desplegar los resultados en la pantalla.

Por ejemplo, las líneas 6 y 7 obtienen el tamaño del tipo de datos `short int` e imprimen en la pantalla el número de bytes, `2`. Por los resultados, usted sabe que en mi máquina el tipo de datos `short int` es de 2 bytes de longitud.

De la misma manera, las líneas 8 y 9 muestran que el tamaño del tipo de datos `long int` es de 4 bytes de longitud, que es la misma que la del tipo de datos `float` obtenida en las líneas 10 y 11.

Las líneas 12 y 13 obtienen el tamaño del tipo de datos `double`, el cual es de 8 bytes en mi máquina. Luego, después de ser modificado por el modificador `long`, el tamaño del tipo de datos `double` se incrementa a 10 bytes (es decir, 80 bits), el cual es impreso por la llamada a la función `printf()` de las líneas 14 y 15.

Como en el ejemplo anterior, es probable que sus resultados sean diferentes si su sistema maneja anchos de datos distintos a los de mi máquina. Al ejecutar este programa en su propia máquina, puede determinar el ancho de estos tipos de datos para su sistema.

Cómo agregar `h`, `l` o `L` a los especificadores de formato `printf` y `fprintf`

Las funciones `printf` y `fprintf` necesitan saber el tipo de datos exacto de los argumentos que se les pasan, a fin de evaluar de manera adecuada dichos argumentos e imprimir sus valores en forma significativa. La cadena de formato de las funciones `printf` y `fprintf` emplea los especificadores de conversión `d`, `i`, `o`, `u`, `x` o `X` para indicar que el argumento correspondiente es un entero, y que es de tipo `int` o `unsigned int`.

Puede agregar `h` dentro del especificador de formato entero (de la siguiente manera: `%hd`, `%hi` o `%hu`) para indicar que el argumento correspondiente es un `short int` o un `unsigned short int`.

Por otra parte, utilizar `%ld` o `%Ld` especifica que el argumento correspondiente es `long int`. Y `%lu` o `%Lu` se utilizan para los datos `unsigned long int`.

El programa del listado 9.3 muestra el uso de `%hd`, `%lu` y `%ld`.

LISTADO 9.3 Uso de `%hd`, `%ld` y `%lu`

```
1:  /* 09L03.c: Uso de los especificadores %hd, %ld y %lu */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     short int         x;
```

continúa

El estándar ANSI le permite indicar que una constante tiene el tipo `long`, con sólo agregar el sufijo `l` o `L` a la constante:

```
long int x, y;
x = 123456789l;
y = 0xABCD1234L;
```

Aquí, las constantes del tipo de datos `long int`, `123456789l` y `0xABCD1234L`, se asignan a las variables `x` y `y`, respectivamente.

Además puede declarar una variable entera `long` de la siguiente manera:

```
long x;
```

lo que equivale a

```
long int x;
```

El listado 9.2 contiene un programa que puede imprimir el número de bytes para diferentes tipos de datos modificados que proporciona el compilador de C utilizado para compilar el programa.

LISTADO 9.2 Modificación de datos con `short` y `long`

```
1:  /* 09L02.c: Uso de los modificadores short y long */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     printf("El tamaño de short int es: %d.\n",
7:         sizeof(short int));
8:     printf("El tamaño de long int es: %d.\n",
9:         sizeof(long int));
10:    printf("El tamaño de float es: %d.\n",
11:        sizeof(float));
12:    printf("El tamaño de double es: %d.\n",
13:        sizeof(double));
14:    printf("El tamaño de long double es: %d.\n",
15:        sizeof(long double));
16:    return 0;
17: }
```

Obtuve los siguientes resultados después de ejecutar en mi computadora el archivo `09L02.exe`:

SALIDA

```
El tamaño de short int es: 2.
El tamaño de long int es: 4.
El tamaño de float es: 4.
El tamaño de double es: 8.
El tamaño de long double es: 10.
```

Materiał chroniony prawem autorskim

la función `printf()` de la línea 15 (de hecho, quizá recuerde que en la hora anterior utilizamos `%u` para especificar el tipo de datos `unsigned int` como formato de despliegue).

Con base en los resultados, puede ver que `0xFFFF` es igual a `-1` para el tipo de datos `signed int`, y `65535` para el tipo de datos `unsigned int`. Aquí, el tipo de datos entero tiene una longitud de 16 bits.

Las líneas 16 y 17 imprimen `0x3039` y `0xCFC7`, que son los formatos hexadecimales de los valores decimales `12345` y `-12345`, respectivamente. De acuerdo con el método mencionado en la sección anterior, `0xCFC7` se obtiene sumando `1` al valor de complemento de `0x3039`.

Puede obtener resultados diferentes a los de este ejemplo, dependiendo del ancho de los distintos tipos de datos de su sistema. Lo que es importante entender, es la diferencia entre los tipos de datos `signed` y `unsigned`.

Modificación del tamaño de los datos

En ocasiones desea reducir la memoria que ocupan las variables, o necesita incrementar el espacio de almacenamiento para ciertos tipos de datos. Por fortuna, el lenguaje C le proporciona la flexibilidad para modificar el tamaño de los tipos de datos. En las dos secciones siguientes se presentan los modificadores de datos `short` y `long`.

El modificador `short`

Por medio del modificador `short` se puede modificar un tipo de datos para que ocupe menos memoria. Por ejemplo, puede aplicar el modificador `short` a una variable entera de 32 bits, lo cual reduce a 16 bits la memoria que ocupa la variable.

Puede usar el modificador `short` de la siguiente manera:

```
short x;
```

o bien,

```
unsigned short y;
```

De manera predeterminada, un tipo de datos `short int` es un número con signo. Por lo tanto, en la instrucción `short x;`, `x` es una variable con signo de tipo entero corto.

El modificador `long`

Si necesita más memoria para almacenar valores dentro de un rango más amplio, puede usar el modificador `long` para definir un tipo de datos con espacio de almacenamiento aumentado.

Por ejemplo, dada una variable entera `x` de 16 bits de longitud, la declaración

```
long int x;
```

aumenta el tamaño de `x` a por lo menos 32 bits.

LISTADO 9.1 Modificación de datos con `signed` y `unsigned`

```
1:  /* 09L01.c: Uso de los modificadores signed y unsigned */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     signed char  ch;
7:     int          x;
8:     unsigned int y;
9:
10:    ch = 0xFF;
11:    x = 0xFFFF;
12:    y = 0xFFFFu;
13:    printf("El valor decimal de signed 0xFF es %d.\n", ch);
14:    printf("El valor decimal de signed 0xFFFF es %d.\n", x);
15:    printf("El valor decimal de unsigned 0xFFFFu es %u.\n", y);
16:    printf("El valor hexadecimal del decimal 12345 es 0x%X.\n", 12345);
17:    printf("El valor hexadecimal del decimal -12345 es 0x%X.\n", -12345);
18:    return 0;
19: }
```

En mi máquina, el archivo ejecutable del programa del listado 9.1 se llama `09L01.exe`. (Observe que al compilar este programa podría obtener un mensaje de advertencia con respecto a la instrucción de asignación `ch = 0xFF;` de la línea 10, debido a que `ch` fue declarada como una variable `signed char`. Puede ignorar el mensaje de advertencia.)

Los siguientes resultados se desplegaron en la pantalla de mi máquina después de ejecutar el archivo `09L01.exe`:

SALIDA

```
El valor decimal de signed 0xFF es -1
El valor decimal de signed 0xFFFF es -1.
El valor decimal de unsigned 0xFFFFu es 65535.
El valor hexadecimal del decimal 12345 es 0x3039.
El valor hexadecimal del decimal -12345 es 0xCFC7.
```

ANÁLISIS Como puede ver en el listado 9.1, la línea 6 declara una variable `char` con signo, `ch`. Las variables `int x` y `unsigned int y` se declaran en las líneas 7 y 8, respectivamente. En las líneas 10 a 12 se inicializan las tres variables, `ch`, `x` y `y`. Observe que en la línea 12 se pone el sufijo `u` a `0xFFFF` para indicar que la constante es un entero sin signo.

La instrucción de la línea 13 muestra el valor decimal de la variable `signed char`, `ch`. Los resultados muestran que el valor decimal correspondiente de `0xFF` es `-1` para la variable `char` con signo `ch`.

Las líneas 14 y 15 imprimen los valores decimales de la variable `int x` (la cual tiene signo de manera predeterminada) y de la variable `unsigned int y`, respectivamente. Observe que se utiliza el especificador de formato para enteros sin signo `%u` para la variable `y` de

Materiał chroniony prawem autorskim

utilizan el bit de signo. (Los modificadores `short` y `long` se presentan más adelante en este capítulo.) De manera predeterminada, todos los tipos de datos enteros, con excepción del tipo de datos `char`, son cantidades con signo. Sin embargo, el estándar ANSI no requiere que el tipo de datos `char` tenga signo; depende de los fabricantes de compiladores. Por lo tanto, si desea utilizar una variable de carácter con signo, y asegurarse de que el compilador la reconoce, puede declarar la variable de tipo carácter de la siguiente manera:

```
signed char ch;
```

a fin de que el compilador sepa que la variable de tipo carácter `ch` tiene signo, lo que significa que la variable puede contener valores negativos y positivos. Si el tipo `char` es de 8 bits de longitud, entonces una variable `signed char` contendría valores desde `-128` (es decir, 2^7) hasta `127` (2^7-1). En contraste, un carácter sin signo estaría en el rango de `0` a `255` (2^8-1).

El modificador `unsigned`

El lenguaje C también le proporciona el modificador `unsigned`, el cual se puede utilizar para indicar al compilador de C que el tipo de datos especificado sólo es capaz de contener valores no negativos.

Al igual que el modificador `signed`, el modificador `unsigned` sólo tiene sentido para los tipos de datos enteros (`char`, `int`, `short int` y `long int`).

Por ejemplo, la declaración

```
unsigned int x;
```

le indica al compilador de C que la variable entera `x` solamente puede asumir valores positivos del `0` al `65535` (es decir, 2^{16}-1), si el tipo de datos `int` tiene una longitud de 16 bits; un tipo `int` (con signo) contendría valores desde `-32768` (-2^{15}) hasta `32767` (2^{15}-1).

De hecho, de acuerdo con el estándar ANSI, `unsigned int` es equivalente (por sí mismo) a `unsigned`. En otras palabras, `unsigned int x;` es lo mismo que `unsigned x;`.

Además el estándar ANSI le permite indicar que una constante es de tipo `unsigned` poniendo el sufijo `u` o `U` a la constante. Por ejemplo,

```
unsigned int x, y;
x = 12345U;
y = 0xABCDu;
```

Aquí se asignan las constantes enteras sin signo `12345U` y `0xABCDu` a las variables `x` y `y`, respectivamente.

El programa del listado 9.1 es un ejemplo del uso de los modificadores `signed` y `unsigned`.

Además aprenderá acerca de varias funciones matemáticas que ofrece el lenguaje C, como son:

- La función `sin()`
- La función `cos()`
- La función `tan()`
- La función `pow()`
- La función `sqrt()`

Cómo habilitar o inhabilitar el bit de signo

Como usted sabe, es muy fácil expresar un número negativo en decimal. Todo lo que necesita hacer es poner un signo de menos frente al valor absoluto del número (el valor absoluto es la distancia del número a partir del cero). Pero, ¿cómo representa la computadora un número negativo en el formato binario?

Normalmente se puede utilizar un bit para indicar si el valor de un número representado en el formato binario es negativo. A este bit se le llama *bit de signo*. Las dos secciones siguientes le presentan dos modificadores de datos, `signed` y `unsigned`, que pueden emplearse para habilitar o inhabilitar el bit de signo.

Cabe señalar que algunas computadoras podrían no expresar los números negativos en la forma descrita; de hecho, el lenguaje C estándar no hace ningún requerimiento de que se utilice un bit de signo, aunque éste es un método común. Lo importante es entender las diferencias entre los tipos de datos `signed` y `unsigned`.

El modificador signed

En los enteros, el bit más a la izquierda se puede usar como bit de signo. Por ejemplo, si el tipo de datos `int` es de 16 bits de longitud y el bit más a la derecha se cuenta como el bit 0, entonces puede usar el bit 15 como bit de signo. Cuando se le asigna `1` al bit de signo, el compilador de C sabe que el valor representado por la variable de datos es negativo.

Existen varias formas de representar un valor negativo para los tipos de datos `float` o `double`. Las implementaciones de los tipos de datos `float` y `double` exceden el alcance de este libro. Para más detalles sobre la representación de valores negativos de los tipos de datos `float` y `double`, consulte el libro *El Lenguaje de Programación C*, de Kernighan y Ritchie.

El lenguaje C proporciona un modificador de datos, `signed`, que se puede utilizar para indicarle al compilador que los tipos de datos enteros (`char`, `int`, `short int` y `long int`)

Materiał chroniony prawem autorskim

HORA 9

Modificadores de datos y funciones matemáticas

Si no tiene éxito a la primera, transforme sus datos.

—Leyes de Murphy de las computadoras

En la hora 4, "Tipos de datos y palabras reservadas", aprendió acerca de varios tipos de datos del lenguaje C, como `char`, `int`, `float` y `double`. En esta hora aprenderá acerca de cuatro modificadores de datos que le permiten tener un gran control sobre los datos. Las palabras reservadas de C para los cuatro modificadores de datos son las siguientes:

- `signed`
- `unsigned`
- `short`
- `long`

2. Tomando los valores de `x` y `y` asignados en el ejercicio 1, escriba un programa que imprima los valores de `!x` y `!y` utilizando los formatos `%d` y `%u` en la función `printf()`.
3. Dadas `x = 123` y `y = 4`, escriba un programa que despliegue los resultados de las expresiones `x << y` y `x >> y`.
4. Escriba un programa que muestre los valores (en hexadecimal) de las expresiones `0xFFFF^0x8888`, `0xABCD & 0x4567`, y `0xDCBA | 0x1234`.
5. Use el operador `?:` y la instrucción `for` para escribir un programa que continúe tomando caracteres introducidos por el usuario hasta encontrar el carácter `q`. (Sugerencia: ponga la expresión `x!='q' ? 1 : 0` como la segunda expresión en una instrucción `for`.)

Por otra parte, ¦¦ es el operador lógico OR, y requiere de dos operandos (o expresiones). El operador produce `0` sólo si ambos operandos se evalúan como `0`. En caso contrario, produce `1`.

P ¿Por qué `1 << 3` equivale a 1 * 2^3?

R La expresión `1 << 3` le indica a la computadora que desplace 3 bits del operando `1` hacia la izquierda. El formato binario del operando es `0001` (observe que aquí sólo se muestran los cuatro bits de orden inferior). Después de desplazar 3 bits a la izquierda, el número binario se convierte en `1000`, que equivale a $1 * 2^3+0 * 2^2+0 * 2^1+0 * 2^0$; es decir, $1 * 2^3$.

P ¿Qué puede hacer el operador condicional (`?:`)?

R Si, bajo ciertas condiciones, hay dos respuestas posibles, puede usar el operador `?:` para seleccionar una de las dos respuestas con base en el resultado producido al comprobar las condiciones. Por ejemplo, la expresión `(edad > 65) ? "Retirado": "No retirado"` le indica a la computadora que si el valor de `edad` es mayor que `65`, entonces se debe elegir la cadena `Retirado`; en caso contrario se selecciona `No retirado`.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Qué producen, respectivamente, las expresiones `(x=1) && (y=10)` y `(x=1) & (y=10)`?
2. Dadas `x = 96`, `y = 1`, y `z = 69`, ¿cómo se evalúa la expresión `!y ? x == z : y`?
3. Si tiene dos variables enteras `x` y `y`, teniendo `x` el valor binario 0011000000111001 y teniendo `y` el valor binario `1100111111000110`, ¿qué valores producen las dos expresiones `~x` y `~y`?
4. Dada x=9, ¿qué produce `(x%2==0)¦¦(x%3==0)`? ¿Y que produce `(x%2==0)&&(x%3==0)`?
5. ¿Es `8 >> 3` equivalente a 8 / 2^3? ¿Qué hay de `1 << 3`?

Ejercicios

1. Dadas `x = 0xEFFF` y `y = 0x1000` (es decir, `EFFF` y `1000` como valores hexadecimales), ¿qué valores hexadecimales se obtienen al evaluar `~x` y `~y`?

Materiał chroniony prawem autorskim

Resumen

En esta lección aprendió los siguientes operadores lógicos y de manejo de bits, los cuales son muy importantes en C:

- El operador `sizeof` evalúa el número de bytes que tiene un tipo de datos específico. Puede utilizar este operador para medir el tamaño de un tipo de datos en su máquina.
- El operador lógico AND (`&&`) produce `1 (verdadero lógico)` sólo si sus dos expresiones de operandos se evalúan como valores diferentes de cero. En caso contrario, produce `0`.
- El operador lógico OR (`¦¦`) produce `0` sólo si sus dos operandos se evalúan como `0`. En caso contrario, produce `1`.
- El operador lógico NOT (`!`) produce `0` cuando su operando se evalúa como diferente de cero, y produce `1` sólo si su operando se evalúa como `0`.
- Hay seis operadores de manejo de bits: el operador AND a nivel de bits (`&`), el operador OR a nivel de bits (`¦`), el operador XOR a nivel de bits (`^`), el operador de complemento a nivel de bits (`~`), el operador de desplazamiento a la derecha (`>>`) y el operador de desplazamiento a la izquierda (`<<`).
- El operador condicional (`?:`) es el único operador en C que puede tener tres operandos.

En la siguiente lección aprenderá acerca de los modificadores de tipo de datos del lenguaje C.

Preguntas y respuestas

P ¿Por qué es necesario el operador `sizeof`?

R El operador `sizeof` se puede utilizar para medir los tamaños de todos los tipos de datos definidos en C. Al escribir un programa portable en C que necesite saber el tamaño de una variable entera, es una mala idea codificar directamente el tamaño con base en la máquina que esté usando en ese momento. La mejor forma de indicarle al programa el tamaño de la variable es utilizar el operador `sizeof`, el cual produce el tamaño de la variable entera en tiempo de ejecución.

P ¿Cuál es la diferencia entre `¦` y `¦¦`?

R `¦` es el operador OR a nivel de bits que tiene dos operandos. El operador `¦` compara cada bit de un operando con el bit correspondiente en otro operando. Si ambos bits son `0`, entonces se coloca `0` en el resultado, en la misma posición del bit. En caso contrario, se pone `1` en el resultado.

LISTADO 8.7 continuación

```
6:      int   x;
7:
8:      x = sizeof(int);
9:      printf("%s\n",
10:        (x == 2)
11:        ? "El tipo de datos int tiene 2 bytes."
12:        : "int no tiene 2 bytes.");
13:     printf("El valor máximo de int es: %d\n",
14:        (x != 2) ? ~(1 << x * 8 - 1) : ~(1 << 15) );
15:     return 0;
16: }
```

El siguiente resultado se desplegó en la pantalla al ejecutar en mi máquina el archivo `08L07.exe`:

SALIDA

```
El tipo de datos int tiene 2 bytes.
El valor máximo de int es: 32767
```

ANÁLISIS En la línea 8 del listado 8.7 se mide primero el tamaño del tipo de datos `int` por medio del operador `sizeof`, y se asigna el número de bytes a la variable entera `x`.

Las líneas 9 a 12 contienen una instrucción, en la cual se usa el operador condicional (`?:`) para comprobar si el número de bytes guardados en `x` es igual a `2`, y se imprime el resultado. Si la expresión `x == 2` se evalúa como diferente de cero, entonces la función `printf()` de la instrucción imprime la cadena `El tipo de datos int tiene 2 bytes`. En caso contrario, se imprime en la pantalla la cadena `int no tiene 2 bytes`.

Además, la instrucción de las líneas 13 y 14 trata de determinar el valor máximo del tipo de datos `int` en la máquina actual. En la instrucción se evalúa primero la expresión `x != 2`. Si la expresión devuelve un valor diferente de cero (es decir, el número de bytes del tipo de datos `int` no es igual a `2`), se evalúa la expresión `~(1 << x * 8 - 1)`, y se elige el resultado como el valor de retorno. Aquí, la expresión `~(1 << x * 8 - 1)` es una forma general para calcular el valor máximo del tipo de datos `int`, el cual equivale a $2^{(x * 8 - 1)} - 1$. (En la sección anterior se presentaron los operadores de complemento, `~`, y de desplazamiento, `<<`.)

Por otra parte, si la condición `x != 2` de la línea 14 devuelve `0`, lo cual significa que el valor de `x` sí es igual a `2`, se elige el resultado de la expresión `~(1 << 15)`. Aquí tal vez ya haya usted determinado que `~(1 << 15)` equivale a $2^{15}-1$, que es el valor máximo que puede tener un tipo de datos `int` de 16 bits.

El resultado que se despliega en la pantalla muestra que el tipo de datos `int` es de 2 bytes (o `16` bits) de longitud en mi máquina, y que el valor máximo del tipo de datos `int` es `32767`.

Material chroniony prawem autorskim

> La operación del operador de desplazamiento hacia la derecha (>>) equivale a dividir entre potencias de dos. En otras palabras, la instrucción
>
> `x >> y`
>
> equivale a
>
> $x / 2^y$
>
> Aquí, `x` es un entero no negativo.
>
> Por otra parte, el desplazamiento hacia la izquierda equivale a multiplicar por potencias de dos; es decir,
>
> `x << y`
>
> equivale a
>
> $x * 2^y$

¿Qué significa x?y:z?

Al operador `?:` se le llama *operador condicional*, y es el único operador que tiene tres operandos. La forma general del operador condicional es

```
x ? y : z
```

Aquí, `x`, `y` y `z` son tres expresiones de operandos. Entre ellos, `x` contiene la condición a probar, mientras que `y` y `z` representan los dos valores finales posibles de la expresión. Si `x` se evalúa como diferente de cero (verdadero lógicamente), entonces se elige `y`; en caso contrario, `z` es el resultado que produce la expresión condicional. El operador condicional se utiliza como una especie de forma abreviada de una instrucción `if`.

Por ejemplo, la expresión

```
x > 0 ? 'T' : 'F'
```

se evalúa como `'T'` si el valor de `x` es mayor que `0`. En caso contrario, la expresión condicional se evalúa como `'F'`.

El listado 8.7 muestra el uso del operador condicional.

TECLEE **LISTADO 8.7** Uso del operador condicional

```
1:  /* 08L07.c: Uso del operador ?: */
2:  #include <stdio.h>
3:
4:  main()
5:  {
```

continúa

Asimismo, la expresión `5 << 1` desplaza al operando `5` un bit hacia la izquierda, y produce `10` en decimal.

El programa del listado 8.6 imprime más resultados por medio de los operadores de desplazamiento.

TECLEE **LISTADO 8.6** Uso de los operadores de desplazamiento

```
1:  /* 08L06.c: Uso de los operadores de desplazamiento */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int   x, y, z;
7:
8:     x = 255;
9:     y = 5;
10:    printf("Dadas x = %4d, es decir, 0X%04X\n", x, x);
11:    printf("      y = %4d, es decir, 0X%04X\n", y, y);
12:    z = x >> y;
13:    printf("x >> y  produce: %6d, es decir, 0X%04X\n", z, z);
14:    z = x << y;
15:    printf("x << y  produce: %6d, es decir, 0X%04X\n", z, z);
16:    return 0;
17: }
```

Los siguientes resultados se desplegaron al ejecutar en mi computadora el archivo `08L06.exe`:

SALIDA

```
Dadas x =  255, es decir, 0X00FF
      y =    5, es decir, 0X0005
x >> y  produce:      7, es decir, 0X0007
x << y  produce:   8160, es decir, 0X1FE0
```

ANÁLISIS En la línea 6 del listado 8.6 se declaran tres variables enteras, `x`, `y` y `z`. En la línea 8 se inicializa `x` a `255`; en la línea 9 se inicializa `y` a `5`. Después, las líneas 10 y 11 despliegan en la pantalla los valores de `x` y `y`.

La instrucción de la línea 12 desplaza `y` bits hacia la derecha al operando `x`, y luego asigna el resultado a `z`. La línea 13 imprime el resultado del desplazamiento realizado en la línea 12. El resultado es `7` en decimal, o `0X0007` en hexadecimal.

Las líneas 14 y 15 desplazan `y` bits hacia la izquierda al operando `x`, y también despliegan el resultado en la pantalla. El resultado del desplazamiento hacia la izquierda es `8160` en decimal, o `0x1FE0` en hexadecimal.

Materiał chroniony prawem autorskim

Las líneas 14 y 15 realizan la operación que especifica el operador OR a nivel de bits (¦) e imprimen el resultado en los formatos decimal y hexadecimal. De la misma manera, las líneas 16 y 17 dan el resultado de la operación que realizó el operador XOR a nivel de bits (^).

Por último, la instrucción de la línea 18 imprime el valor complementario de `x` por medio del operador de complemento (~) a nivel de bits. El resultado se despliega en la pantalla en los formatos decimal y hexadecimal.

Observe que en la función `printf()` se utilizan el especificador de formato entero sin signo con un ancho mínimo de campo de `6`, `%6u`, y el especificador de formato hexadecimal en mayúsculas con un ancho mínimo de campo de `4`, `%04X`. Aquí se utiliza el tipo de datos entero sin signo para que se pueda mostrar y entender con facilidad el valor complementario de un entero. En la hora 9 se presentan más detalles sobre el modificador de datos sin signo.

> No confunda los operadores a nivel de bits `&` y `¦` con los *operadores lógicos* `&&` y `¦¦`. Por ejemplo,
>
> `(x=1) & (y=10)`
>
> es una expresión completamente distinta de
>
> `(x=1) && (y=10)`

Uso de operadores de desplazamiento

En C hay dos operadores de desplazamiento. El operador `>>` desplaza a la derecha los bits de un operando; el operador `<<` los desplaza hacia la izquierda.

Las formas generales de los dos operadores de desplazamiento son

```
x >> y
x << y
```

Aquí, `x` es un operando que va ser desplazado. `y` contiene el número especificado de posiciones a desplazar.

Por ejemplo, la expresión `8 >> 2` le indica a la computadora que desplace al operando `8` dos bits a la derecha, lo cual produce el número `2` en decimal. La expresión

$8 >> 2$ que equivale a $(1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0) >> 2$

produce lo siguiente:

$(0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0)$, lo cual equivale a 0010 (en el formato binario), o a 2 (en el formato decimal).

Observe que el valor complementario de 12 es 65523 debido a que el tipo de datos entero sin signo (de 16 bits) tiene el número máximo 65535. En otras palabras, 65,523 es el resultado de restar 12 de 65,535. (El modificador de datos sin signo se presenta en la hora 9, "Modificadores de datos y funciones matemáticas".)

El programa del listado 8.5 muestra el uso de los operadores a nivel de bits.

TECLEE **LISTADO 8.5** Uso de operadores a nivel de bits

```
1:  /* 08L05.c: Uso de operadores a nivel de bits */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int   x, y, z;
7:
8:     x = 4321;
9:     y = 5678;
10:    printf("Dadas x = %u, es decir, 0X%04X\n", x, x);
11:    printf("      y = %u, es decir, 0X%04X\n", y, y);
12:    z = x & y;
13:    printf("x & y  devuelve: %6u, es decir, 0X%04X\n", z, z);
14:    z = x ¦ y;
15:    printf("x ¦ y  devuelve: %6u, es decir, 0X%04X\n", z, z);
16:    z = x ^ y;
17:    printf("x ^ y  devuelve: %6u, es decir, 0X%04X\n", z, z);
18:    printf("  ~x   devuelve: %6u, es decir, 0X%04X\n", ~x, ~x);
19:    return 0;
20: }
```

Los siguientes resultados se desplegaron en la pantalla después de crear el archivo 08L05.exe y ejecutarlo en mi máquina:

SALIDA

```
Dadas x = 4321, es decir, 0X10E1
      y = 5678, es decir, 0X162E
x & y  devuelve:  4128, es decir, 0X1020
x ¦ y  devuelve:  5871, es decir, 0X16EF
x ^ y  devuelve:  1743, es decir, 0X06CF
  ~x   devuelve: 61214, es decir, 0XEF1E
```

ANÁLISIS En la línea 6 del listado 8.5 se declaran tres variables enteras, x, y y z. Las líneas 8 y 9 asignan a x y a y los valores 4321 y 5678, respectivamente. Las líneas 10 y 11 imprimen los valores de x y y en formatos decimal y hexadecimal. Los números hexadecimales tienen el prefijo 0X.

La instrucción de la línea 12 asigna el resultado de la operación que realizó el operador AND a nivel de bits (&) con las variables x y y. Luego, la línea 13 despliega el resultado en los formatos decimal y hexadecimal.

Materiał chroniony prawem autorskim

En la línea 10, la expresión relacional `num < 7` se evalúa como `0` debido a que el valor de `num` no es menor que `7`. Sin embargo, mediante el operador lógico de negación, `!(num < 7)` produce `1` (consulte la tabla 8.3).

Asimismo, la expresión lógica `!(num > 7)` se evalúa como `1` en la línea 11.

La expresión relacional `num == 7` produce 1, debido a que `num` tiene el valor de `7`; sin embargo, la expresión lógica `!(num == 7)` de la línea 12 se evalúa como `0`.

Manejo de bits

En las horas anteriores aprendió que los datos y archivos de la computadora están hechos de bits. En esta sección aprenderá acerca de un conjunto de operadores que le permite acceder y manipular bits específicos. Pero antes de continuar, aprendamos más acerca de

El operador lógico NOT (!)

El formato general del operador lógico NOT es:

!expresión

en donde *expresión* es el operando del operador NOT.

En la tabla 8.3 se muestra la tabla de verdad del operador NOT.

TABLA 8.3 Los valores que devuelve el operador NOT

expresión	*Valor que devuelve !*
diferente de cero	0
0	1

Veamos ahora el ejemplo del listado 8.4, el cual muestra cómo usar el operador lógico NOT, o de negación, (!).

TECLEE **LISTADO 8.4** Uso del operador lógico NOT (!)

```
1:  /* 08L04.c: Uso del operador lógico NOT */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int   num;
7:
8:     num = 7;
9:     printf("Dado num = 7\n");
10:    printf("!(num < 7)  produce: %d\n", !(num < 7));
11:    printf("!(num > 7)  produce: %d\n", !(num > 7));
12:    printf("!(num == 7) produce: %d\n", !(num == 7));
13:    return 0;
14: }
```

Los siguientes resultados se desplegaron al ejecutar el archivo `08L04.exe`:

SALIDA

```
Dado num = 7
!(num < 7)  produce: 1
!(num > 7)  produce: 1
!(num == 7) produce: 0
```

ANÁLISIS Observe que en la línea 8 se inicializa la variable entera `num` a `7`, lo cual es desplegado después por la función `printf()` de la línea 9.

Materiał chroniony prawem autorskim

```
3:
4:  main()
5:  {
6:     int   num;
7:
8:     printf("Introduzca un entero entre 0 y 9 que se pueda \ndividir
       entre 2 y entre 3:\n");
9:     for (num = 1; (num%2 != 0) || (num%3 != 0); )
10:        num = getchar() - '0';
11:    printf("¡Bien! El entero es: %d\n", num);
12:    return 0;
13: }
```

Los siguientes resultados se desplegaron después de ejecutar en mi máquina el archivo `08L03.exe`. Los números en negritas son los que yo introduje. (Es necesario oprimir la tecla Entrar cada vez que se introduzca un número.) En el rango del 0 al 9, 0 y 6 son los únicos números que se pueden dividir tanto entre 2 como entre 3 sin producir residuo:

SALIDA

```
Introduzca un entero entre 0 y 9 que se pueda
dividir entre 2 y entre 3:
2
3
4
5
6¡Bien! El entero es: 6
```

ANÁLISIS En la línea 6 del listado 8.3 se declara una variable entera, `num`. La línea 8 imprime dos líneas que solicitan al usuario que introduzca un entero entre 0 y 9.

La variable entera `num` se inicializa en la primera expresión de la instrucción `for` de la línea 9. `num` se inicializa con `1` debido a que `1` no es divisible entre `2` ni entre `3`. De este modo, se garantiza que el ciclo se ejecute por lo menos una vez.

La parte clave del programa del listado 8.3 es la expresión lógica de la instrucción `for`:

```
(num%2 != 0) || (num%3 != 0)
```

Aquí se evalúan las dos expresiones relacionales `num%2 != 0` y `num%3 != 0`. De acuerdo con la tabla de verdad del operador `||` (vea la tabla 8.2), usted sabe que si una de las expresiones relacionales se evalúa como diferente de cero, lo cual indica que el valor de `num` no se puede dividir entre `2` o entre `3`, entonces la expresión lógica OR se evalúa como `1`, lo cual permite que el ciclo `for` continúe.

El ciclo `for` se detiene sólo si el usuario introduce un entero entre 0 y 9 que sea divisible entre 2 y entre 3. En otras palabras, cuando ambas expresiones relacionales se evalúan como `0`, el operador lógico OR produce `0`, lo cual ocasiona la terminación del ciclo `for`.

num%3 == 0 también produce 1. Entonces, de acuerdo con la tabla de verdad del operador && (vea la tabla 8.1), usted sabe que la combinación del operador lógico AND (&&) con las dos expresiones relacionales produce 1 si ambas expresiones se evalúan como diferentes de cero. En caso contrario, produce 0.

En nuestro caso, cuando num se inicializa a 0 en la línea 8, tanto 0%2 como 0%3 producen residuos de cero, así que las dos expresiones relacionales se evalúan como 1. Por lo tanto, el operador lógico AND produce 1.

Sin embargo, cuando a num se le asigna el valor de 2 o 3 como se muestra en las líneas 11 y 14, los operadores lógicos AND de las líneas 13 y 16 producen 0. La razón es que 2 y 3 no se pueden dividir entre 2 y 3, respectivamente, y obtener un residuo.

La línea 17 asigna luego el valor de 6 a num. Ya que 6 es un múltiplo tanto de 2 como de 3, el operador lógico AND de la línea 19 produce 1, el cual es impreso por la función printf() de las líneas 18 y 19.

El operador lógico OR (¦¦)

Como mencioné antes, el operador lógico OR produce 1 (verdadero lógicamente) si una o ambas expresiones se evalúan a un valor diferente de cero. El operador ¦¦ produce 0 si, y sólo si, ambas expresiones producen 0.

Un formato general del operador lógico OR es:

exp1 ¦¦ *exp2*

en donde *exp1* y *exp2* son dos expresiones de operandos que evalúa el operador OR.

La tabla 8.2 es la tabla de verdad del operador OR.

Tabla 8.2 Los valores que devuelve el operador OR

exp1	exp2	¦¦ Produce
diferente de cero	diferente de cero	1
diferente de cero	0	1
0	diferente de cero	1
0	0	0

El programa del listado 8.3 muestra cómo usar el operador lógico OR (¦¦).

TECLEE **Listado 8.3** Uso del operador lógico OR (¦¦)

```
1:  /* 08L03.c: Uso del operador lógico OR */
2:  #include <stdio.h>
```

Materiał chroniony prawem autorskim

8

El listado 8.2 es un ejemplo del uso del operador lógico AND (&&).

TECLEE **LISTADO 8.2** Uso del operador lógico AND (&&)

```
1:  /* 08L02.c: Uso del operador lógico AND */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int   num;
7:
8:     num = 0;
9:     printf("El operador AND produce: %d\n",
10:           (num%2 == 0) && (num%3 == 0));
11:    num = 2;
12:    printf("El operador AND produce: %d\n",
13:           (num%2 == 0) && (num%3 == 0));
14:    num = 3;
15:    printf("El operador AND produce: %d\n",
16:           (num%2 == 0) && (num%3 == 0));
17:    num = 6;
18:    printf("El operador AND produce: %d\n",
19:           (num%2 == 0) && (num%3 == 0));
20:
21:    return 0;
22: }
```

Después de compilar y enlazar este programa, se crea un archivo ejecutable, `08L02.exe`. Los siguientes resultados se desplegaron en la pantalla después de ejecutar dicho archivo en mi máquina:

SALIDA

```
El operador AND produce: 1
El operador AND produce: 0
El operador AND produce: 0
El operador AND produce: 1
```

ANÁLISIS En la línea 6 del listado 8.2 se declara una variable entera, `num`, y se inicializa por primera vez en la línea 8. Las líneas 9 y 10 imprimen el valor que produce el operador lógico AND en la siguiente expresión:

```
(num%2 == 0) && (num%3 == 0)
```

Aquí ve dos expresiones relacionales, `num%2 == 0` y `num%3 == 0`. En la hora 3, "La estructura de un programa de C", aprendió que se puede usar el operador aritmético `%` para obtener el residuo de la división del primer operando entre el segundo. Por lo tanto, `num%2` produce el residuo de `num` dividido entre `2`. La expresión relacional `num%2 == 0` produce `1` si el residuo es igual a `0`, es decir, el valor de `num` se puede dividir entre `2` sin dejar residuo. Asimismo, si el valor de `num` se puede dividir entre `3`, la expresión relacional

Todo es lógico

Es el momento de aprender acerca de un nuevo conjunto de operadores: *los operadores lógicos.*

En el lenguaje C hay tres operadores lógicos:

`&&`	El operador lógico AND
`¦¦`	El operador lógico OR
`!`	El operador lógico NOT

Los dos primeros operadores, AND y OR, son operadores binarios; es decir, ambos tienen dos operandos (uno a la izquierda y otro a la derecha del operador). El operador lógico AND (`&&`) se usa para evaluar la veracidad o falsedad de un par de expresiones. Si ambas expresiones se evalúan como `0` (es decir, falsas lógicamente), el operador produce un valor de `0`. En caso contrario, si, y sólo si, las expresiones de ambos operandos se evalúan como valores diferentes de cero, el operador lógico AND produce un valor de `1` (verdadero lógicamente).

El operador lógico OR (`¦¦`) produce un valor de `1` cada vez que una o ambas expresiones de los operandos se evalúan como diferentes de cero (verdadero lógicamente). El operador `¦¦` produce `0` sólo si ambas expresiones de los operandos se evalúan como cero (falso). El operador lógico NOT (`!`) es un operador unario; es decir, sólo tiene un operando (la expresión a su derecha). Si el operando se evalúa como un valor diferente de cero, el operador `!` produce `0` (falso lógicamente); sólo cuando la expresión del operando se evalúa como `0`, el operador produce `1` (verdadero lógicamente).

Las siguientes tres secciones contienen ejemplos que le muestran cómo usar los tres operadores lógicos.

El operador lógico AND (&&)

Un formato general del operador lógico AND es:

```
exp1 && exp2
```

en donde *`exp1`* y *`exp2`* son dos expresiones de operandos que evalúa el operador AND.

Una buena forma de entender el operador AND es observar una tabla que muestre los valores que produce el operador AND dependiendo de los valores posibles de *`exp1`* y *`exp2`*. Vea la tabla 8.1, a la cual podemos llamar *tabla de verdad* del operador AND.

TABLA 8.1 Los valores que devuelve el operador AND

exp1	exp2	&& Produce
diferente de cero	diferente de cero	1
diferente de cero	0	0
0	diferente de cero	0
0	0	0

Materiał chroniony prawem autorskim

```
17:     printf("El tamaño de double es: %d bytes\n", sizeof(double));
18:     printf("El tamaño de dbl_num es: %d bytes\n", sizeof dbl_num);
19:     return 0;
20: }
```

Después de compilar y enlazar este programa, se creó el archivo ejecutable `08L01.exe`. Los siguientes resultados se imprimieron en la pantalla después de ejecutar dicho archivo en mi máquina:

SALIDA

```
El tamaño de char es: 1 byte
El tamaño de ch es: 1 byte
El tamaño de int es: 2 bytes
El tamaño de int_num es: 2 bytes
El tamaño de float es: 4 bytes
El tamaño de flt_num es: 4 bytes
El tamaño de double es: 8 bytes
El tamaño de dbl_num es: 8 bytes
```

La línea 2 del listado 8.1 incluye el archivo de encabezado `stdio.h` para la función `printf()` que se utiliza en la instrucción que está dentro del cuerpo de la función `main()`. Las líneas 6 a 9 declaran una variable de tipo `char` (`ch`), una variable de tipo `int` (`int_num`), una variable de tipo `float` (`flt_num`) y una variable de tipo `double` (`dbl_num`), respectivamente. Además, se inicializan estas cuatro variables. Observe que en la línea 8, el valor inicial para `flt_num` tiene una `f` como sufijo para especificar `float`. (Como aprendió en la hora 4, puede utilizar `f` o `F` para especificar el tipo `float` de un número de punto flotante.)

Las líneas 11 y 12 despliegan el tamaño del tipo de datos `char`, así como el de la variable `ch` de tipo `char`. Observe que el operador `sizeof` se usa tanto en la línea 11 como en la 12 para obtener el número de bytes que pueden tener el tipo de datos `char` o la variable `ch`. Debido a que la variable `ch` no es una palabra reservada en C, en la línea 12 se omiten los paréntesis del operador `sizeof`.

Las instrucciones de las líneas 11 y 12 imprimen las dos primeras líneas de los resultados, respectivamente. Puede ver en los resultados que el tamaño del tipo de datos `char` es de 1 byte, que es el mismo tamaño de la variable `ch`. Esto no es ninguna sorpresa, ya que la variable `ch` está declarada como de tipo `char`.

De la misma manera, las líneas 13 y 14 imprimen los tamaños del tipo de datos `int` y de la variable `int_num` de tipo `int` utilizando el operador `sizeof`. Puede ver que el tamaño de ambos es de 2 bytes.

Asimismo, utilizando el operador `sizeof`, las líneas 15 a 18 proporcionan el tamaño del tipo de datos `float`, el de la variable `flt_num` de tipo `float`, el del tipo de datos `double` y el de la variable `dbl_num` de tipo `double`, respectivamente. La sección de salida muestra que el tipo de datos `float` y la variable `flt_num` tienen el mismo tamaño (4 bytes). El tamaño del tipo de datos `double` y de la variable `dbl_num` es, en ambos casos, de 8 bytes.

Cómo medir el tamaño de los datos

Tal vez recuerde que en la hora 4, "Tipos de datos y palabras reservadas", mencioné que cada tipo de datos tiene su propio tamaño. El tamaño de los tipos de datos varía dependiendo del sistema operativo y del compilador de C que utilice. Por ejemplo, en la mayoría de las estaciones de trabajo de UNIX, un entero tiene una longitud de 32 bits, mientras que la mayoría de los compiladores de C sólo manejan 16 bits en una máquina basada en DOS.

Así que, ¿cómo puede saber el tamaño de un tipo de datos en su máquina? La respuesta es que puede medir el tipo de datos por medio del operador `sizeof` que proporciona C.

La forma general del operador `sizeof` es

```
sizeof (expresión)
```

Aquí, *`expresión`* es el tipo de datos o variable cuyo tamaño es medido por el operador `sizeof`. El operador `sizeof` evalúa el tamaño, en bytes, de su operando. El operando del operador `sizeof` puede ser una palabra reservada de C que represente a un tipo de datos (como `int`, `char` o `float`), o puede ser una expresión que se refiera a los tipos de datos cuyo tamaño se puede determinar (como una constante o el nombre de una variable).

En la forma general del operador los paréntesis son opcionales. Si la expresión no es una palabra reservada de C para un tipo de datos, se pueden omitir los paréntesis. Por ejemplo, la siguiente instrucción

```
tamanio = sizeof(int);
```

coloca el tamaño, en bytes, del tipo de datos `int` en la variable `tamanio`. El programa del listado 8.1 encuentra los tamaños de los tipos de datos `char`, `int`, `float` y `double` en mi máquina.

TECLEE **LISTADO 8.1** Uso del operador sizeof

```
1:  /* 08L01.c: Uso del operador sizeof */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char   ch = ' ';
7:     int    int_num = 0;
8:     float  flt_num = 0.0f;
9:     double dbl_num = 0.0;
10:
11:    printf("El tamaño de char es: %d byte\n", sizeof(char));
12:    printf("El tamaño de ch es: %d byte\n", sizeof ch );
13:    printf("El tamaño de int es: %d bytes\n", sizeof(int));
14:    printf("El tamaño de int_num es: %d bytes\n", sizeof int_num);
15:    printf("El tamaño de float es: %d bytes\n", sizeof(float));
16:    printf("El tamaño de flt_num es: %d bytes\n", sizeof flt_num);
```

Materiał chroniony prawem autorskim

HORA 8

Uso de operadores condicionales

La civilización avanza ampliando el número de operaciones importantes que podemos realizar sin pensar en ellas.

—A. N. Whitehead

En la hora 6, "Manejo de datos", aprendió acerca de algunos operadores importantes de C, como los operadores de asignación aritmética, el operador unario menos, u operador de negación, los operadores de incremento y decremento y los operadores relacionales. En esta lección conocerá más operadores que son muy importantes en la programación en C, incluyendo

- El operador `sizeof`
- Los operadores lógicos
- Los operadores de manipulación de bits
- El operador condicional

```
37:     for (i=0; i<max; i++)
38:         suma += lista[i];
39:     return suma;
40: }
```

Después de ejecutar el programa `16L05.exe`, obtuve los siguientes resultados desplegados en la pantalla de mi computadora:

SALIDA

```
¡Es una cadena!
¡Es una cadena!
La suma devuelta por SumaDatos() es: 15
La suma devuelta por SumaDatos() es: 15
```

ANÁLISIS El objetivo del programa del listado 16.5 es mostrar cómo pasar dos apuntadores, un apuntador entero que apunta a un arreglo de enteros y un apuntador de carácter que hace referencia a una cadena de caracteres, a dos funciones que se declaran en las líneas 4 y 5.

Observe que se usan expresiones como `char *ch` e `int *lista` como argumentos en las declaraciones de función, lo cual le indica al compilador que se están pasando a las funciones `ImpCh()` y `SumaDatos()` un apuntador de tipo `char` y un apuntador de tipo `int`, respectivamente.

Dentro del cuerpo de la función `main()`, las líneas 8 y 9 declaran un arreglo (`cadena`) de tipo `char` que se inicializa con una cadena de caracteres, y una variable de apuntador (`aptr_cadena`) de tipo `char`. La línea 10 declara e inicializa un arreglo entero (`lista`) con un conjunto de enteros. En la línea 11 se declara una variable de apuntador de tipo `int`, `aptr_int`.

La dirección de inicio del arreglo `cadena` se asigna al apuntador `aptr_cadena` mediante la instrucción de asignación de la línea 14. Después, el apuntador `aptr_cadena` se pasa como argumento a la función `ImpCh()` de la línea 15. De acuerdo con la definición de `ImpCh()` de las líneas 27 a 30, la llamada a `printf()` de la línea 29 imprime el contenido del arreglo `cadena`, cuya dirección de inicio se pasa a la función como argumento, dentro de la función `ImpCh()`.

De hecho, puede seguir usando el nombre del arreglo `cadena` como argumento y pasarlo a la función `ImpCh()`. La línea 16 muestra que la dirección de inicio del arreglo de caracteres se pasa a `ImpCh()` por medio del nombre del arreglo.

La instrucción de la línea 19 asigna la dirección de inicio del arreglo entero `lista` al apuntador entero `aptr_int`. Luego, en la línea 21, el apuntador `aptr_int` se pasa a la función `SumaDatos()`, junto con `5`, que es el número máximo de elementos contenidos por el arreglo `lista`. El argumento `max` se utiliza debido a que la función no puede determinar el tamaño del arreglo si solamente se da la dirección de inicio. Puede ver en la definición de la función `SumaDatos()` de las líneas 32 a 40, que `SumaDatos()` suma todos los elementos enteros en `lista` y devuelve la suma al invocador. A continuación, las líneas 20 y 21 imprimen el resultado que devuelve `SumaDatos()`.

Materiał chroniony prawem autorskim

La expresión de la línea 23 también invoca a la función `SumaDatos()`, pero esta vez se usa el nombre del arreglo `lista` como argumento para la función. No es ninguna sorpresa que la dirección de inicio del arreglo `lista` se pase con éxito a la función `SumaDatos()`, y que la instrucción `printf()` de las líneas 22 y 23 despliegue el resultado correcto en la pantalla.

Paso de arreglos multidimensionales como argumentos

En la hora 12, "Arreglos", aprendió acerca de los arreglos multidimensionales. En esta sección verá cómo pasar estos arreglos a funciones.

Quizá ya adivinó que pasar un arreglo multidimensional a una función es similar a pasarle un arreglo de una dimensión. Puede pasarle a la función el formato sin especificación de tamaño de un arreglo multidimensional, o un apuntador que contenga la dirección de inicio del arreglo multidimensional. El listado 16.6 es un ejemplo de estos dos métodos.

TECLEE **LISTADO 16.6** Paso de arreglos multidimensionales a funciones

```
1:  /* 16L06.c: Paso de arreglos multidimensionales a funciones */
2:  #include <stdio.h>
3:  /* declaración de funciones */
4:  int SumaDatos1(int lista[][5], int max1, int max2);
5:  int SumaDatos2(int *lista, int max1, int max2);
6:  /* función main() */
7:  main()
8:  {
9:     int lista[2][5] = {1, 2, 3, 4, 5,
10:                       5, 4, 3, 2, 1};
11:    int *aptr_int;
12:
13:    printf("La suma devuelta por SumaDatos1() es: %d\n",
14:            SumaDatos1(lista, 2, 5));
15:    aptr_int = &lista[0][0];
16:    printf("La suma devuelta por SumaDatos2() es: %d\n",
17:            SumaDatos2(aptr_int, 2, 5));
18:
19:    return 0;
20: }
21: /* definición de función */
22: int SumaDatos1(int lista[][5], int max1, int max2)
23: {
24:    int i, j;
25:    int suma = 0;
26:
27:    for (i=0; i<max1; i++)
28:       for (j=0; j<max2; j++)
29:       suma += lista[i][j];
30:    return suma;
31: }
```

Materiał chroniony prawem autorskim

Usted puede acceder a una cadena de caracteres utilizando el apuntador correspondiente del arreglo. De hecho, en el listado 16.7 hay dos funciones, `CadImp1()` y `CadImp2()`, mismas que se pueden llamar para acceder a las cadenas de caracteres. Puede ver en la declaración de función de la línea 4 que la función `CadImp1()` toma un apuntador de apuntadores (`**cadena1`), el cual es indireccionado dentro de la función `CadImp1()` para representar los cuatro apuntadores que apuntan a las cuatro cadenas de caracteres. La definición de `CadImp1()` está en las líneas 23 a 29.

Por otra parte, la función `CadImp2()` sólo toma como argumento una variable de apuntador, e imprime una cadena de caracteres a la que hace referencia el apuntador. Las líneas 31 a 35 dan la definición de la función `CadImp2()`.

Regresemos ahora a la función `main()`. En la línea 16 se llama a la función `CadImp1()` con el nombre del arreglo de apuntadores, `cadena`, como argumento. `CadImp1()` despliega entonces en la pantalla los cuatro versos del poema de Byron. El ciclo `for` de las líneas 17 y 18 hace lo mismo llamando cuatro veces a la función `CadImp2()`. En cada llamada se pasa a `CadImp2()` la dirección de inicio de un verso. Por lo tanto, usted ve los versos del poema impresos dos veces en la pantalla.

Apuntadores a funciones

Antes de concluir esta hora, hay otra cosa interesante que necesita aprender: los apuntadores a funciones.

Al igual que con los apuntadores a arreglos, usted puede declarar un apuntador que se inicialice con el valor izquierdo de una función (el valor izquierdo es la dirección de memoria en la que está ubicada la función). Entonces puede llamar a la función a través del apuntador.

El programa del listado 16.8 es un ejemplo que declara un apuntador a una función.

TECLEE **LISTADO 16.8** Cómo apuntar a una función

```
1:  /* 16L08.c: Cómo apuntar a una función */
2:  #include <stdio.h>
3:  /* declaración de función */
4:  int CadImp(char *cadena);
5:  /* función main() */
6:  main()
7:  {
8:     char cadena[25] = "Apuntando a una función.";
9:     int (*aptr)(char *cadena);
10:
11:    aptr = CadImp;
12:    if (!(*aptr)(cadena))
```

Materiał chroniony prawem autorskim

Resumen

En esta lección aprendió los siguientes conceptos y aplicaciones muy importantes de apuntadores y arreglos de C:

- Antes de usar un apuntador, siempre debe asegurarse de que apunte a una ubicación de memoria válida.
- Es posible mover la posición de un apuntador sumando o restando un entero.
- El tamaño escalar de un apuntador se determina por el tamaño de sus datos, el cual se especifica en la declaración del apuntador.
- En dos apuntadores del mismo tipo, usted puede restar el valor de un apuntador del otro para obtener el desplazamiento entre ellos.
- Puede acceder a los elementos de un arreglo a través de un apuntador que contenga la dirección de inicio del arreglo.
- Puede pasar un arreglo sin especificación de tamaño como un solo argumento a una función.
- También puede pasar un arreglo a una función a través de un apuntador. El apuntador debe contener la dirección de inicio del arreglo.
- Puede pasarle a una función el formato sin especificación de tamaño de un arreglo multidimensional, o un apuntador que contenga la dirección de inicio del arreglo multidimensional.
- Una función que toma un arreglo como argumento no sabe cuántos elementos hay en éste. También debe pasarle a la función el número de elementos como otro argumento.
- Los arreglos de apuntadores son útiles al tratar con cadenas de caracteres.
- Puede asignar un apuntador a la dirección de una función, y llamar después a la función empleando ese apuntador.

En la siguiente lección aprenderá cómo asignar memoria en C.

Preguntas y respuestas

P ¿Por qué es necesaria la aritmética de apuntadores?

R Lo bello de utilizar apuntadores es que puede moverlos a cualquier lugar para tener acceso a datos válidos almacenados en las ubicaciones de memoria a las que hacen referencia los apuntadores. Para ello, puede emplear la aritmética de apuntadores para sumar (o restar) un entero a un apuntador. Por ejemplo, si un apuntador de tipo carácter, `aptr_cadena`, contiene la dirección de inicio de una cadena de caracteres, la expresión `aptr_cadena+1` se mueve a la siguiente ubicación de memoria, la cual contiene el segundo carácter de la cadena.

P ¿Cómo determina el compilador el tamaño escalar de un apuntador?

R El compilador determina el tamaño escalar de un apuntador por medio de su tipo de datos, el cual se especifica en la declaración. Cuando se suma o se resta un entero a un apuntador, el valor real que emplea el compilador es la multiplicación del entero por el tamaño del tipo del apuntador. Por ejemplo, dado el apuntador `aptr_int` de tipo `int`, el compilador interpreta la expresión `aptr_int + 1` como `aptr_int + 1 * sizeof(int)`. Si el tamaño del tipo de datos `int` es de 2 bytes, la expresión `aptr_int + 1` en realidad significa desplazarse 2 bytes más arriba de la ubicación de memoria a la que hace referencia el apuntador.

P ¿Cómo puede acceder a un elemento del arreglo utilizando un apuntador?

R En los arreglos de una dimensión puede asignar la dirección de inicio de un arreglo a un apuntador del mismo tipo, y luego mover el apuntador a la ubicación de memoria que contenga el valor del elemento en el que esté interesado. Después indirecciona el apuntador para obtener el valor del elemento. En los arreglos multidimensionales el método es similar, aunque tiene que pensar en las otras dimensiones al mismo tiempo (vea el ejemplo del listado 16.6).

P ¿Por qué necesita usar arreglos de apuntadores?

R En muchos casos es muy útil usar arreglos de apuntadores. Por ejemplo, es conveniente utilizar un arreglo de apuntadores para apuntar a un conjunto de cadenas de caracteres para que pueda acceder a cualquiera de las cadenas a las que hace referencia el apuntador correspondiente del arreglo.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios del taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. Dados un apuntador `aptr_ch de tipo char`, un apuntador `aptr_int de tipo int`, y un apuntador `aptr_flt` de tipo `float`, ¿cuántos bytes se sumarán en su máquina en cada una de las expresiones siguientes?

    - `aptr_ch + 4`
    - `aptr_int + 2`
    - `aptr_flt + 1`
    - `aptr_ch + 12`

Material chroniony prawem autorskim

- `aptr_int + 6`
- `aptr_flt + 3`

2. ¿Qué obtendrá al realizar la resta de los apuntadores de tipo `int` `aptr1` y `aptr2`, si la dirección que contiene `aptr1` es `0x100A`, y la dirección que contiene `aptr2` es `0x1006`?
3. Dado que el tamaño del tipo de datos `double` es de 8 bytes de longitud, y que la dirección actual que contiene la variable de apuntador `aptr_db` de tipo `double` es `0x0238`, ¿cuáles son las direcciones que contienen `aptr_db-1` y `aptr_db+5`?
4. Dadas las siguientes declaraciones y asignaciones:

```
char ch[] = {'a', 'b', 'c', 'd', 'A', 'B', 'C', 'D'};
char *aptr;
aptr = &ch[1];
```

¿qué hacen estas expresiones?

- `*(aptr + 3)`
- `aptr - ch`
- `*(aptr - 1)`
- `*aptr = 'F'`

Ejercicios

1. Dada la cadena de caracteres `¡Me atrajo C!`, escriba un programa que pase la cadena a una función y despliegue la cadena en la pantalla.
2. Reescriba el programa del ejercicio 1. Esta vez cambie la cadena por `¡Me agrada C!`; para esto mueva un apuntador que se inicialice con la dirección de inicio de la cadena y actualice dicha dirección con los nuevos caracteres. Luego pase la cadena actualizada a la función para desplegar en la pantalla el contenido de la cadena.
3. Dado el arreglo bidimensional `cadena` que se inicializa como

```
char cadena[2][15] = { "¿Sabes qué?", "C es poderoso." };
```

escriba un programa que pase la dirección de inicio de `cadena` a una función que imprima el contenido del arreglo de caracteres.

4. Reescriba el programa del listado 16.7. Esta vez, el arreglo de apuntadores se inicializa con las siguientes cadenas:

```
"Domingo", "Lunes", "Martes", "Miércoles", "Jueves", "Viernes", y "Sábado".
```

Materiał chroniony prawem autorskim

Asignación de memoria en tiempo de ejecución

Existen muchos casos en los que usted no sabe el tamaño exacto de los arreglos que utiliza en sus programas, y lo sabe hasta mucho después, al estar ejecutándolos. Puede especificar el tamaño de los arreglos por adelantado, pero éstos pueden ser muy pequeños o demasiado grandes si durante la ejecución cambia de manera dramática el número de elementos que desea depositar en ellos.

Por fortuna, C le proporciona cuatro funciones de asignación dinámica de memoria que puede emplear para asignar o reasignar ciertos espacios de memoria mientras su programa está en ejecución. Además puede liberar el espacio de memoria asignada tan pronto como deje de ser útil. Estas cuatro funciones, `malloc()`, `calloc()`, `realloc()` y `free()`, se presentan en las secciones siguientes

La función `malloc()`

Puede utilizar la función `malloc()` para asignar un espacio de memoria del tamaño especificado.

SINTAXIS

La sintaxis de la función `malloc()` es

```
#include <stdlib.h>
void *malloc(size_t tamaño);
```

Aquí, *tamaño* indica el número de bytes de almacenamiento a asignar. La función `malloc()` devuelve un apuntador `void`.

Observe que se debe incluir el archivo de encabezado `stdlib.h` para poder llamar a la función `malloc()`. Debido a que la función `malloc()` devuelve por sí misma un apuntador `void`, entonces su tipo se convierte en forma automática al tipo del apuntador que está a la izquierda de un operador de asignación, de acuerdo con el estándar ANSI de C.

En el mercado hay algunos compiladores de C, por lo regular versiones antiguas, que no son 100% compatibles con el ANSI de C. Si utiliza uno de esos compiladores, podría obtener mensajes de advertencia o mensajes de error acerca de sus declaraciones de `malloc()`, `calloc()` o `realloc()`. Por ejemplo, si escribe algo como lo siguiente:

```
int *aptr;
aptr = malloc ( 10 * sizeof(int) );
```

al tratar de compilar el código, podría obtener un mensaje de error relacionado con el tipo de retorno de la función `malloc()`. Esto se debe a que el compilador de C que está utilizando no es 100% compatible con el ANSI C, y no sabe cómo convertir el tipo de retorno de `malloc()`. Si éste es el caso, lo que puede hacer es usar el operador de conversión `(int *)` que se muestra a continuación:

```
int *aptr;
aptr = (int *) malloc ( 10 * sizeof(int) );
```

> Esto le indica al compilador el tipo de apuntador que devuelve `malloc()`, y termina con los mensajes del compilador de C. Aquí tiene que asegurarse de que el tipo de la memoria asignada, el tipo del apuntador y el tipo del operador de conversión sean iguales. Asimismo, puede anteponer un operador de conversión a las funciones `calloc()` o `realloc()` cuando utilice un compilador que no sea compatible con el ANSI C.
>
> Debido a que este libro se centra en el ANSI C, que es el estándar de la industria, no utilizo operadores de conversión en los ejemplos de programas que usan `malloc()`, `calloc()` o `realloc()`.

La función `malloc()` devuelve un apuntador nulo si no puede asignar una porción del espacio de memoria. Por lo regular, esto sucede cuando no hay memoria suficiente. Por lo tanto, siempre debe verificar el apuntador que devuelve `malloc()` antes de utilizarlo.

El listado 17.1 muestra el uso de la función `malloc()`.

TECLEE **LISTADO 17.1** Uso de la función `malloc()`

```
1:  /* 17L01.c: Uso de la función malloc */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>
5:  /* declaración de función */
6:  void CopiaCad(char *cadena1, char *cadena2);
7:  /* función main() */
8:  main()
9:  {
10:    char cadena[] = "Usa malloc() para asignar memoria.";
11:    char *aptr_cadena;
12:    int resultado;
13:    /* llama a malloc() */
14:    aptr_cadena = malloc( strlen(cadena) + 1);
15:    if (aptr_cadena != NULL){
16:       CopiaCad(cadena, aptr_cadena);
17:       printf("La cadena a la que apunta aptr_cadena es:\n%s\n",
18:              aptr_cadena);
19:       resultado = 0;
20:    }
21:    else{
22:       printf("Error en malloc().\n");
23:       resultado = 1;
24:    }
25:    return resultado;
```

continúa

Materiał chroniony prawem autorskim

LISTADO 17.1 continuación

```
26: }
27: /* definición de función */
28: void CopiaCad(char *cadena1, char *cadena2)
29: {
30:     int i;
31:
32:     for (i=0; cadena1[i]; i++)
33:        cadena2[i] = cadena1[i];
34:     cadena2[i] = '\0';
35: }
```

Después de crear y ejecutar el programa `17L01.exe` del listado 17.1, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
La cadena a la que apunta aptr_cadena es:
Usa malloc() para asignar memoria.
```

ANÁLISIS La finalidad del programa del listado 17.1 es usar la función `malloc()` para asignar una porción de espacio de memoria que tenga el mismo tamaño que una cadena de caracteres. Luego, el contenido de la cadena se copia a la memoria asignada y referenciada por el apuntador que devuelve la función `malloc()`. El contenido de la memoria se despliega en la pantalla para demostrar que el espacio de memoria sí contiene la cadena después de la asignación y duplicación.

Observe que en las líneas 3 y 4 se incluyen otros dos archivos de encabezado, `stdlib.h` y `string.h`, respectivamente, para las funciones `malloc()` y `strlen()` que se llaman en la línea 14.

La línea 10 declara un arreglo de tipo `char`, `cadena`, que se inicializa con la cadena `"Usa malloc() para asignar memoria."`. En la línea 11 se declara una variable de apuntador de tipo `char`, `aptr_cadena`.

La instrucción de la línea 14 asigna un espacio de memoria de `strlen(cadena)+1` bytes llamando a la función `malloc()`. Debido a que la función `strlen()` no cuenta el carácter nulo al final de una cadena, sumar `1` al valor que devuelve `strlen(cadena)` nos da el número total de bytes que es necesario asignar. El valor del apuntador devuelto se asigna a la variable de apuntador de tipo `char`, `aptr_cadena`, después de que se llama a la función `malloc()` en la línea 14.

La instrucción `if-else` de las líneas 15 a 24 verifica el apuntador devuelto por la función `malloc()`. Si es un apuntador nulo, en las líneas 22 y 23 se imprime un mensaje de error, y se asigna 1 al valor de devolución de la función `main()`. (Recuerde que un valor diferente de cero devuelto por la instrucción `return` indica una terminación anormal.)

Pero si el apuntador devuelto no es un apuntador nulo, entonces la dirección de inicio del arreglo `cadena` y el apuntador `aptr_cadena` se pasan a la función `CopiaCad()` de la línea 16. La función `CopiaCad()`, cuya definición se da en las líneas 28 a 35, copia el contenido del arreglo `cadena` en la memoria asignada a la que apunta `aptr_cadena`. Entonces, la llamada a `printf()` de las líneas 17 y 18 imprime el contenido copiado en la memoria asignada. La línea 19 asigna `0` al valor de retorno después del éxito en la asignación de memoria y la duplicación de la cadena.

Los resultados desplegados en mi pantalla muestran que se asignó una porción de memoria y que la cadena se copió en la memoria.

Debido a que siempre hay un límite, habrá un problema potencial si continúa asignando memoria. Es muy fácil que se quede sin memoria, si la asigna sin nunca liberarla. En la siguiente sección aprenderá cómo usar la función `free()` para liberar, cuando ya no los necesite, los espacios de memoria que haya asignado.

Uso de `free()` para liberar memoria asignada

Debido a que la memoria es un recurso limitado, debe asignar una porción exacta de memoria justo antes de que la necesite, y liberarla tan pronto termine de utilizarla.

El programa del listado 17.2 muestra el uso de la función `free()` para liberar memoria asignada.

TECLEE **LISTADO 17.2** Uso de las funciones `free()` y `malloc()` juntas

```
1:  /* 17L02.c: Uso de la función free() */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  /* declaración de funciones */
5:  void MultiplicaDatos(int max, int *aptr);
6:  void ImpTabla(int max, int *aptr);
7:  /* función main() */
8:  main()
9:  {
10:    int *aptr_int, max;
11:    int final;
12:    char tecla = 'c';
13:
14:    max = 0;
15:    final = 0;
16:    while (tecla != 'x'){
17:       printf("Escriba un entero de un dígito:\n");
```

continúa

Materiał chroniony prawem autorskim

LISTADO 17.2 continuación

```
18:       scanf("%d", &max);
19:
20:       aptr_int = malloc(max * max * sizeof(int));  /* llama a malloc() */
21:       if (aptr_int != NULL){
22:          MultiplicaDatos(max, aptr_int);
23:          ImpTabla(max, aptr_int);
24:          free(aptr_int);
25:       }
26:       else{
27:          printf("Error en malloc().\n");
28:          final = 1;
29:          tecla = 'x';  /* detiene el ciclo while */
30:       }
31:       printf("\n\nOprima la tecla x para terminar; oprima otra tecla para
          continuar.\n");
32:       scanf("%s", &tecla);
33:    }
34:    printf("\n¡Hasta pronto!\n");
35:    return final;
36: }
37: /* definición de función */
38: void MultiplicaDatos(int max, int *aptr)
39: {
40:    int i, j;
41:
42:    for (i=0; i<max; i++)
43:       for (j=0; j<max; j++)
44:          *(aptr + i * max + j) = (i+1) * (j+1);
45: }
46: /* definición de función */
47: void ImpTabla(int max, int *aptr)
48: {
49:    int i, j;
50:
51:    printf("La tabla de multiplicar de %d es:\n",
52:            max);
53:    printf(" ");
54:    for (i=0; i<max; i++)
55:       printf("%4d", i+1);
56:    printf("\n ");
57:    for (i=0; i<max; i++)
58:       printf("----", i+1);
59:    for (i=0; i<max; i++){
60:       printf("\n%d¦", i+1);
61:       for (j=0; j<max; j++)
62:         printf("%3d ", *(aptr + i * max + j));
63:    }
64: }
```

Materiał chroniony prawem autorskim

expresión `sizeof(int)` da el tamaño en bytes del tipo de datos `int` para la computadora en la que se ejecuta el programa.

Si la función `malloc()` devuelve un apuntador nulo, entonces se asigna 1 al valor de retorno de la función `main()` para indicar una terminación anormal (vea la línea 28), y se detiene el ciclo `while` asignando una `'x'` a la variable `tecla` de la línea 29.

En caso contrario, si la función `malloc()` asigna con éxito el espacio de memoria, se llama a la función `MultiplicaDatos()` de la línea 22 para realizar cada multiplicación. Los resultados se guardan en el espacio de memoria al que apunta el apuntador `aptr_int`. Luego se imprime la tabla de multiplicar llamando a la función `ImpTabla()` de la línea 23.

Cuando la memoria para mantener la tabla de multiplicar ya no es necesaria, se llama a la función `free()` de la línea 24 para liberar el espacio de memoria asignado al que apunta `aptr_int`.

Si no se liberara la memoria, el programa ocuparía más y más memoria, ya que el usuario seguiría introduciendo enteros para generar más tablas de multiplicar. El programa provocaría a la larga una colisión del sistema operativo, o se vería forzado a terminar. Por medio de las funciones `free()` y `malloc()`, puedo seguir ejecutando el programa y ocupar la cantidad exacta de espacio de memoria que necesito, ni más ni menos.

La función calloc()

Además de la función `malloc()`, también pude usar la función `calloc()` para asignar espacio de memoria en forma dinámica. Las diferencias entre ambas funciones son que la segunda toma dos argumentos y que el espacio de memoria que asigna se inicializa siempre a `0`. Sin embargo, no hay una garantía de que el espacio de memoria que asigna `malloc()` se inicialice a `0`.

▼ **SINTAXIS**

La sintaxis de la función `calloc()` es

```
#include <stdlib.h>
void *calloc(size_t nelementos, size_t tamaño);
```

Aquí, *`nelementos`* es el número de elementos que usted desea guardar en el espacio de memoria asignado. *`tamaño`* indica el número de bytes que ocupa cada elemento. La función `calloc()` también devuelve un apuntador `void`.

▲

La función `calloc()` devuelve un apuntador nulo si no puede asignar una porción del espacio de memoria.

El listado 17.3 presenta un ejemplo del uso de la función `calloc()`. En este ejemplo se imprime el valor inicial del espacio de memoria asignado por `calloc()`.

TECLEE **LISTADO 17.3** Uso de la función `calloc()`

```
1:  /* 17L03.c: Uso de la función calloc() */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  /* función main() */
5:  main()
6:  {
7:     float *aptr1, *aptr2;
8:     int i, n;
9:     int final = 1;
10:
11:    n = 5;
12:    aptr1 = calloc(n, sizeof(float));
13:    aptr2 = malloc(n * sizeof(float));
14:    if (aptr1 == NULL)
15:       printf("Error en malloc().\n");
16:    else if (aptr2 == NULL)
17:       printf("Error en calloc().\n");
18:    else {
19:       for (i=0; i<n; i++)
20:          printf("aptr1[%d]=%5.2f,  aptr2[%d]=%5.2f\n",
21:            i, *(aptr1 + i), i, *(aptr2 + i));
22:       free(aptr1);
23:       free(aptr2);
24:       final = 0;
25:    }
26:    return final;
27: }
```

Después de ejecutar el programa `17L03.exe`. aparecieron en la pantalla los siguientes resultados:

SALIDA

```
aptr1[0] = 0.00,  aptr2[0] = 7042.23
aptr1[1] = 0.00,  aptr2[1] = 1427.00
aptr1[2] = 0.00,  aptr2[2] = 2787.14
aptr1[3] = 0.00,  aptr2[3] =    0.00
aptr1[4] = 0.00,  aptr2[4] = 5834.73
```

ANÁLISIS La finalidad del programa del listado 17.3 es utilizar la función `calloc()` para asignar una porción de espacio de memoria. Para demostrar que la función `calloc()` inicializa a `0` el espacio de memoria asignado, se imprimen los valores iniciales de la memoria. Además se asigna otra porción de espacio de memoria mediante la función `malloc()`, y se imprimen también los valores iniciales del segundo espacio de memoria.

En la línea 12 se llama a la función `calloc()` y se le pasan dos argumentos: la variable `n` de tipo `int` y la expresión `sizeof(float)`. El valor que devuelve la función `calloc()` se asigna a la variable de apuntador `aptr1` de tipo `float`.

Materiał chroniony prawem autorskim

Asimismo, en la línea 13 se llama a la función `malloc()`. Esta función sólo toma un argumento que especifica el número total de bytes que debe tener la memoria asignada. El valor que devuelve la función `malloc()` se asigna entonces a otra variable de apuntador de tipo `float`, `aptr2`.

Como puede ver en las líneas 12 y 13, las funciones `calloc()` y `malloc()` en realidad planean asignar dos porciones de espacio de memoria del mismo tamaño.

La instrucción `if-else-if-else` de las líneas 14 a 25 verifica los dos valores devueltos por las funciones `calloc()` y `malloc()` e imprime los valores iniciales de los dos espacios de memoria asignados si ninguno de los dos valores de devolución es nulo.

Ejecuté varias veces el programa del listado 17.3. El espacio de memoria asignado por la función `calloc()` fue siempre `0`. Pero no hay tal garantía para el espacio de memoria asignado por la función `malloc()`. Los resultados que aquí se muestran corresponden a una ejecución del programa en mi máquina. Puede ver que hay cierta "basura" en el espacio de memoria asignado por la función `malloc()`. Es decir, el valor inicial de la memoria es impredecible. (A veces, el valor inicial de un bloque de memoria asignado por la función `malloc()` es `0`. Pero no se garantiza que el valor inicial sea siempre cero cada vez que se llame a la función `malloc()`.)

La función realloc()

La función `realloc()` le proporciona un medio para modificar el tamaño de una porción de memoria asignada por las funciones `malloc()`, `calloc()` o incluso por la misma `realloc()`.

SINTAXIS

La sintaxis de la función `realloc()` es

```
#include <stdlib.h>
void *realloc(void *bloque, size_t tamaño);
```

Aquí, *`bloque`* es el apuntador que está al inicio de la porción de espacio de memoria asignada previamente. *`tamaño`* especifica el número total de bytes que desea cambiar. La función `realloc()` devuelve un apuntador `void`.

La función `realloc()` devuelve un apuntador nulo si no puede reasignar una porción de espacio de memoria.

La función `realloc()` equivale a la función `malloc()` si el primer argumento que se pasa a `realloc()` es `NULL`. En otras palabras, las siguientes instrucciones son equivalentes:

```
aptr_flt = realloc(NULL, 10 * sizeof(float));
aptr_flt = malloc(10 * sizeof(float));
```

Además puede utilizar la función `realloc()` en lugar de la función `free()`. Para hacer esto, puede pasar el valor `0` a `realloc()` como segundo argumento. Por ejemplo, para liberar un bloque de memoria al que apunta un apuntador `aptr`, puede llamar a la función `free()` como se muestra a continuación:

```
free(aptr);
```

o usar la función `realloc()` de la siguiente manera:

```
realloc(aptr, 0);
```

El programa del listado 17.4 muestra el uso de la función `realloc()` en la reasignación de memoria.

TECLEE **LISTADO 17.4** Uso de la función `realloc()`

```
1:  /* 17L04.c: Uso de la función realloc() */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>
5:  /* declaración de función */
6:  void CopiaCad(char *cadena1, char *cadena2);
7:  /* función main() */
8:  main()
9:  {
10:    char *cadena[4] = {"Hay música en el suspirar de una caña;",
11:                       "Hay música en el ímpetu de un arroyuelo;",
12:                       "En todo hay música si el hombre prestara oído;",
13:                       "La tierra no es más que un eco de las esferas.\n"
14:                        };
15:    char *aptr;
16:    int i;
17:
18:    int final = 0;
19:
20:    aptr = malloc((strlen(cadena [0]) + 1) * sizeof(char));
21:    if (aptr == NULL){
22:      printf("Error en malloc().\n");
23:      final = 1;
24:    }
25:    else{
26:      CopiaCad(cadena [0], aptr);
27:      printf("%s\n", aptr);
28:      for (i=1; i<4; i++){
29:        aptr = realloc(aptr, (strlen(cadena[i]) + 1) * sizeof(char));
30:        if (aptr == NULL){
31:          printf("Error en realloc().\n");
32:          final = 1;
```

continúa

Materiał chroniony prawem autorskim

LISTADO 17.4 continuación

```
33:           i = 4;     /* rompe el ciclo for */
34:         }
35:         else{
36:           CopiaCad(cadena[i], aptr);
37:           printf("%s\n", aptr);
38:         }
39:       }
40:    }
41:    free(aptr);
42:    return final;
43: }
44: /* definición de función */
45: void CopiaCad(char *cadena1, char *cadena2)
46: {
47:    int i;
48:
49:    for (i=0; cadena1[i]; i++)
50:       cadena2[i] = cadena1[i];
51:    cadena2[i] = '\0';
52: }
```

Obtuve los siguientes resultados al ejecutar el archivo `17L04.exe`.

SALIDA

```
Hay música en el suspirar de una caña;
Hay música en el ímpetu de un arroyuelo;
En todo hay música si el hombre prestara oído;
La tierra no es más que un eco de las esferas.
```

ANÁLISIS El objetivo del programa del listado 17.4 es asignar un bloque de espacio de memoria para contener una cadena de caracteres. En el ejemplo hay cuatro cadenas cuya longitud podría variar. Yo usé la función `realloc()` para ajustar el tamaño de la memoria previamente asignada, para que ésta pudiera contener una nueva cadena.

En las líneas 10 a 13 hay cuatro cadenas de caracteres que contienen un hermoso poema escrito por Lord Byron (como puede ver, me encanta la poesía de Byron). Aquí utilice un arreglo de apuntadores, `cadena`, para hacer referencia a las cadenas.

Primero se asigna una porción del espacio de memoria por medio de la llamada a la función `malloc()` de la línea 20. La expresión `(strlen(cadena[0])+1)*sizeof(char)` determina el tamaño del espacio de memoria. Como se mencionó antes, debido a que la función `strlen()` de C no cuenta el carácter nulo que está al final de la cadena, debe recordar asignar espacio para un carácter más para contener el tamaño completo de una cadena. La expresión `sizeof(char)` se usa aquí para portabilidad, aunque el tamaño del tipo de datos `char` es de 1 byte.

Materiał chroniony prawem autorskim

- Antes de poder llamar a las funciones `malloc()`, `calloc()`, `realloc()` o `free()`, debe incluir primero el archivo de encabezado `stdlib.h`.
- Siempre debe verificar los valores que devuelven las funciones `malloc()`, `calloc()` o `realloc()` antes de utilizar la memoria asignada por ellas.

En la siguiente lección aprenderá más acerca de los tipos de datos de C.

Preguntas y respuestas

P ¿Por qué necesita asignar memoria en tiempo de ejecución?

R Muchas veces usted conoce el tamaño exacto de los arreglos hasta que su programa está en ejecución. Usted debe ser capaz de estimar el tamaño de dichos arreglos, pero si los hace demasiado grandes, desperdiciaría memoria. Por otra parte, si los hace demasiado pequeños, perderá datos. La mejor forma es asignar bloques de memoria de manera dinámica y precisamente para aquellos arreglos cuyo tamaño se determina en tiempo de ejecución. Hay cuatro funciones de la biblioteca de C, `malloc()`, `calloc()`, `realloc()` y `free()`, mismas que usted puede utilizar para asignar memoria en tiempo de ejecución.

P ¿Qué significa que la función `malloc()` devuelva un apuntador nulo?

R Si la función `malloc()` devuelve un apuntador nulo, significa que no pudo asignar un bloque de memoria cuyo tamaño se especificó mediante el argumento transferido a la función. Por lo regular, la falla de la función `malloc()` ocurre por el hecho de que no queda suficiente memoria por asignar. Siempre debe verificar el valor que devuelve la función `malloc()` para asegurarse de que tuvo éxito, antes de usar el bloque de memoria asignado por la función.

P ¿Cuáles son las diferencias entre las funciones `calloc()` y `malloc()`?

R Básicamente hay dos diferencias, aunque ambas funciones pueden hacer el mismo trabajo. La primera diferencia es que la función `calloc()` toma dos argumentos, mientras que la función `malloc()` sólo toma uno. La segunda diferencia es que `calloc()` inicializa a `0` el espacio de memoria asignado, mientras que `malloc()` no ofrece dicha garantía.

P ¿Es necesaria la función `free()`?

R Sí. La función `free()` es muy necesaria, y debe utilizarla para liberar bloques de memoria asignada tan pronto como no los necesite. Como usted sabe, la memoria es un recurso limitado en una computadora. Su programa no debe ocupar demasiado espacio de memoria al asignar bloques. Una forma de reducir el tamaño de memoria que ocupa un programa es utilizar la función `free()` para liberar a tiempo la memoria asignada que no se utiliza.

HORA 18

Tipos de datos y funciones especiales

Eso es todo lo que hay, no hay nada más.

—E. Barrymore

En la hora 4, "Tipos de datos y palabras reservadas", aprendió la mayoría de los tipos de datos, como son `char`, `int`, `float` y `double`. En la hora 15, "Funciones", aprendió las bases del uso de funciones de C. En esta hora aprenderá más acerca de los tipos de datos y funciones, a partir de los siguientes temas:

- El tipo de datos `enum`
- La instrucción `typedef`
- La recursión de funciones
- Los argumentos en la línea de comandos

Materiał chroniony prawem autorskim

El tipo de datos `enum`

El lenguaje C le proporciona un tipo de datos adicional: el tipo de datos `enum`. `enum` es una abreviatura de *enumerado*. El tipo de datos enumerado se puede utilizar para declarar constantes enteras con nombre. El tipo de datos `enum` hace que su programa de C sea más legible y fácil de darle mantenimiento. (Otra forma de declarar una constante con nombre es usar la directiva `#define`, la cual se presenta más adelante en este libro.)

Declaración del tipo de datos `enum`

SINTAXIS

La forma general de la declaración del tipo de datos `enum` es

```
enum nom_etiqueta {lista_enumeración} lista_variables;
```

Aquí, *`nom_etiqueta`* es el nombre de la enumeración. *`lista_variables`* presenta una lista de nombres de variables que son del tipo de datos `enum`. *`lista_enumeración`* contiene nombres enumerados definidos que se usan para representar constantes enteras. (Tanto *`nom_etiqueta`* como *`lista_variables`* son opcionales.)

Por ejemplo, la siguiente es una declaración de un tipo de datos `enum` con el nombre de etiqueta `auto`:

```
enum auto {sedan, pick_up, deportivo};
```

Dado lo anterior, puede definir las variables `enum` de la siguiente manera:

```
enum auto local, extranjero;
```

Aquí se definen dos variables `enum`, `local` y `extranjero`.

Por supuesto, siempre puede declarar y definir una lista de variables `enum` en una sola instrucción, como se muestra en la forma general de la declaración de `enum`. Por lo tanto, puede reescribir la declaración `enum` de `local` y `extranjero` de la siguiente manera:

```
enum auto {sedan, pick_up, deportivo} local, extranjero;
```

Asignación de valores a nombres `enum`

De manera predeterminada, el valor entero asociado con el nombre más hacia la izquierda en el campo de lista de enumeración, encerrado entre llaves ({ y }), comienza con `0`, y el valor de cada nombre del resto de la lista se incrementa en uno de izquierda a derecha. Por lo tanto, en el ejemplo anterior, `sedan`, `pick_up` y `deportivo` tienen los valores de `0`, `1` y `2`, respectivamente.

De hecho, puede asignar valores enteros a nombres `enum`. Considerando el ejemplo anterior, puede inicializar los nombres enumerados de la siguiente manera:

```
enum auto {sedan = 60, pick_up = 30, deportivo = 10};
```

Ahora, sedan representa el valor de 60, pick_up tiene el valor de 30, y deportivo asume el valor de 10.

El programa que se muestra en el listado 18.1 imprime los valores de nombres enum.

TECLEE **LISTADO 18.1** Definición de tipos de datos enum

```
1:  /* 18L01.c: Definición de tipos de datos enum */
2:  #include <stdio.h>
3:  /* función main() */
4:  main()
5:  {
6:     enum lenguaje {humano=100,
7:                   animal=50,
8:                   computadora};
9:     enum dias{DOM,
10:              LUN,
11:              MAR,
12:              MIE,
13:              JUE,
14:              VIE,
15:              SAB};
16:
17:    printf("humano: %d,  animal: %d,  computadora: %d\n",
18:       humano, animal, computadora);
19:    printf("DOM: %d\n", DOM);
20:    printf("LUN: %d\n", LUN);
21:    printf("MAR: %d\n", MAR);
22:    printf("MIE: %d\n", MIE);
23:    printf("JUE: %d\n", JUE);
24:    printf("VIE: %d\n", VIE);
25:    printf("SAB: %d\n", SAB);
26:
27:    return 0;
28: }
```

Después de crear y ejecutar en mi computadora el archivo ejecutable 18L01.exe del programa del listado 18.1, se mostraron en la pantalla los siguientes resultados:

SALIDA

```
humano: 100,  animal: 50,  computadora: 51
DOM: 0
LUN: 1
MAR: 2
MIE: 3
JUE: 4
VIE: 5
SAB: 6
```

Materiał chroniony prawem autorskim

ANÁLISIS La finalidad del programa del listado 18.1 es mostrarle los valores predeterminados de los nombres `enum`, así como los valores asignados por el programador a ciertos nombres `enum`.

Como puede ver, hay dos declaraciones `enum` en las líneas 6 a 8 y 9 a 15, respectivamente. Observe que en ambas declaraciones `enum` se omiten las listas de variables, ya que el programa no las necesita.

La primera declaración tiene el nombre de etiqueta `lenguaje` y tres nombres enumerados, `humano`, `animal` y `computadora`. Además, a `humano` se le asigna el valor de `100`; `animal` se inicializa con `50`. De acuerdo con la definición de `enum`, el valor predeterminado de `computadora` es el valor de `animal` incrementado en 1. Por lo tanto, en este caso, el valor predeterminado de `computadora` es `51`.

Los resultados que produce la instrucción de la línea 17 muestran que los valores de `humano`, `animal` y `computadora` son en efecto `100`, `50` y `51`.

La segunda declaración `enum` del programa contiene siete elementos con sus valores predeterminados. Después, las líneas 19 a 25 imprimen estos valores predeterminados. No es una sorpresa ver que los valores representados por los nombres enumerados, `DOM`, `LUN`, `MAR`, `MIE`, `JUE`, `VIE` y `SAB`, son `0`, `1`, `2`, `3`, `4`, `5` y `6`, respectivamente.

Veamos ahora otro ejemplo, el cual se presenta en el listado 18.2, que muestra cómo usar el tipo de datos `enum`.

TECLEE **LISTADO 18.2** Uso del tipo de datos `enum`

```
1:  /* 18L02.c: Uso del tipo de datos enum */
2:  #include <stdio.h>
3:  /* función main() */
4:  main()
5:  {
6:     enum unidades{penny = 1,
7:                  nickel = 5,
8:                  dime = 10,
9:                  quarter = 25,
10:                 dollar = 100};
11:    int unidades_moneda[5] = {
12:                 dollar,
13:                 quarter,
14:                 dime,
15:                 nickel,
16:                 penny};
17:    char *nom_unidad[5] = {
18:                "dollar(s)",
19:                "quarter(s)",
```

```
20:                 "dime(s)",
21:                 "nickel(s)",
22:                 "penny(s)"};
23:     int cent, tmp, i;
24:
25:     printf("Escriba un valor monetario en centavos:\n");
26:     scanf("%d", &cent);  /* obtiene la entrada del usuario */
27:     printf("Lo que equivale a:\n");
28:     tmp = 0;
29:     for (i=0; i<5; i++){
30:        tmp = cent / unidades_moneda[i];
31:        cent -= tmp * unidades_moneda[i];
32:        if (tmp)
33:          printf("%d %s ", tmp, nom_unidad[i]);
34:     }
35:     printf("\n");
36:     return 0;
37: }
```

Durante la ejecución del archivo 18L02.exe, escribí 141 (por 141 centavos) y obtuve los siguientes resultados en la pantalla:

SALIDA

```
Escriba un valor monetario en centavos:
141
Lo que equivale a:
1 dollar(s) 1 quarter(s) 1 dime(s) 1 nickel(s) 1 penny(s)
```

ANÁLISIS

La finalidad del programa del listado 18.2 es usar el tipo de datos enum para representar el valor de la cantidad de dinero que introdujo el usuario.

Dentro de la función main(), en las líneas 6 a 10, se hace una declaración enum con el nombre de etiqueta unidades. Los números asignados a los nombres numerados se basan en su proporción con la unidad de centavo. Por ejemplo, un dollar es igual a 100 centavos. Por lo tanto, al nombre enum dollar se le asigna el valor de 100.

Después de la declaración enum, se declara un arreglo de tipo int, llamado unidades_moneda, y se inicializa con los nombres enumerados de la declaración enum. De acuerdo con la definición del tipo de datos enum, la declaración del arreglo unidades_moneda del programa en realidad equivale a la siguiente:

```
int unidades_moneda[5] = {
               100,
               25,
               10,
               5,
               1};
```

Así que ahora puede usar los nombres enumerados, en lugar de los número enteros, para formar otras expresiones de declaraciones en su programa.

Materiał chroniony prawem autorskim

En las líneas 17 a 22 se declara e inicializa el arreglo de apuntadores nom_unidad. (El uso de arreglos de apuntadores se presentó en la hora 16, "Uso de apuntadores".)

Entonces, la instrucción de la línea 25 solicita al usuario que escriba un número entero en la unidad de centavos. La llamada a scanf() de la línea 26 almacena el número introducido por el usuario en una variable de tipo int llamada cent.

El ciclo for de las líneas 29 a 34 divide el número introducido y lo representa en un formato desglosado por monedas.

Observe que, a través del arreglo unidades_moneda, en las líneas 30 y 31 se usan las constantes enteras representadas por los nombres enumerados. Si el valor de una unidad no es 0, se imprime en la línea 33 la cadena correspondiente a la que apunta la cadena de apuntadores nom_unidad. Por lo tanto, cuando introduje 141 (en unidades de centavos), observé en los resultados su equivalente: 1 dollar(s) 1 quarter(s) 1 dime(s) 1 nickel(s) 1 penny(s).

Cómo hacer definiciones typedef

Con la ayuda de la palabra reservada typedef de C, usted puede crear sus propios nombres de tipos de datos, y hacerlos sinónimos de los tipos de datos. Después, puede usar en sus programas los sinónimos de los nombres en lugar de los propios tipos de datos. Con frecuencia, los sinónimos de nombres definidos por typedef pueden hacer más legible su programa.

Por ejemplo, puede declarar DOS_BYTES como un sinónimo del tipo de datos int:

```
typedef  int  DOS_BYTES;
```

Luego puede comenzar a usar DOS_BYTES para declarar variables enteras de la siguiente manera:

```
DOS_BYTES i, j;
```

lo cual equivale a

```
int i, j;
```

Recuerde que debe hacer una definición typedef antes de utilizar en cualquier declaración de su programa el sinónimo creado en la definición.

Por qué usar typedef

Existen muchas ventajas al utilizar definiciones typedef. Primero, puede consolidar tipos de datos complejos en una sola palabra y después usar esa palabra en declaraciones de variables en su programa. De esta manera, no necesita escribir una y otra vez una declaración compleja, lo cual ayuda a evitar errores al teclear.

La segunda ventaja es que, si en el futuro el tipo de datos cambia, sólo necesita actualizar una definición `typedef`, lo cual corrige todos los usos de esa definición.

De hecho, `typedef` es tan útil, que hay un archivo de encabezado llamado `stddef.h` incluido en el estándar ANSI de C que contiene docenas de definiciones `typedef`. Por ejemplo, `size_t` es un `typedef` del valor que devuelve el operador `sizeof`.

El programa que se muestra en el listado 18.3 es un ejemplo del uso de definiciones `typedef`.

TECLEE **LISTADO 18.3** Uso de definiciones typedef

```
1:  /* 18L03.c: Uso de definiciones typedef */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>
5:
6:  enum constantes{ELEM_NUM = 3,
7:                  DIF='a'-'A'};
8:  typedef char *CADENA[ELEM_NUM];
9:  typedef char *APTR_CAD;
10: typedef char CAR;
11: typedef int ENTERO;
12:
13: void ConvierteaMAY(APTR_CAD cadena1, APTR_CAD cadena2);
14:
15: main()
16: {
17:    CADENA cad;
18:    CADENA luna= {"No importa lo que vistamos",
19:                  "nos convertimos en una hermosa",
20:                  "visión de la luna"};
21:    ENTERO i;
22:    ENTERO term = 0;
23:
24:    for (i=0; i<ELEM_NUM; i++){
25:      cad[i] = malloc((strlen(luna[i])+1) * sizeof(CAR));
26:      if (cad[i] == NULL){
27:        printf("Error en malloc().\n");
28:        term = 1;
29:        i =ELEM_NUM;  /* rompe el ciclo for */
30:      }
31:      ConvierteaMAY(luna[i], cad[i]);
32:      printf("%s\n", luna[i]);
33:    }
34:    for (i=0; i<ELEM_NUM; i++){
35:      printf("\n%s", cad[i]);
```

continúa

Por ejemplo, el prototipo de la función `ConvierteaMAY()` de la línea 13 contiene dos argumentos que son apuntadores de tipo `char` declarados con `APTR_CAD`.

En las líneas 17 a 20 se declaran con `CADENA` dos arreglos de apuntadores, `cad` y `luna`. `luna` se inicializa para apuntar a las cadenas del haikú japonés. En las líneas 21 y 22 se usa `ENTERO` para declarar dos variables de tipo `int`, `i` y `term`.

El ciclo `for` de las líneas 24 a 33 asigna en forma dinámica suficiente espacio de memoria con base en el tamaño del haikú. Luego, en la línea 31, se llama a la función `ConvierteaMAY()` para copiar las cadenas a las que hace referencia `luna` en las ubicaciones de memoria a las que apunta `cad`, así como para convertir todos los caracteres en minúsculas a su contraparte en mayúsculas. La línea 32 imprime las cadenas a las que hace referencia `luna`. La definición de la función `ConvierteaMAY()` se muestra en las líneas 42 a 54.

En las líneas 34 a 37, otro ciclo `for` imprime el contenido de las ubicaciones de memoria a las que hace referencia `cad`. En el contenido hay un total de tres cadenas con caracteres en mayúsculas. Después de desplegar una cadena en la pantalla, se libera el espacio de memoria asignado a ella por medio de la llamada a la función `free()`. Debido a que la línea 35 imprime cada cadena con un carácter de línea nueva al principio en lugar de al final, en la línea 38 se imprime un carácter de línea nueva que seguirá a la última cadena.

En la pantalla se ven dos copias del haikú: la original y otra con caracteres en mayúsculas.

Funciones recursivas

Usted ya sabe que en C una función puede llamar a otra función. Pero, ¿puede una función llamarse a sí misma? La respuesta es sí. Una función puede llamarse a sí misma desde una instrucción dentro del cuerpo de la propia función. Entonces, se dice que dicha función es *recursiva*.

El listado 18.4 contiene un ejemplo de cómo llamar a una función recursiva para sumar enteros del 1 al 100.

TECLEE **LISTADO 18.4** Cómo llamar a una función recursiva

```
1:  /* 18L04.c: Cómo llamar a una función recursiva */
2:  #include <stdio.h>
3:
4:  enum con{MIN_NUM = 0,
5:           MAX_NUM = 100};
6:
7:  int fRecur(int n);
8:
9:  main()
```

continúa

Materiał chroniony prawem autorskim

LISTADO 18.4 continuación

```
10: {
11:     int i, suma1, suma2;
12:
13:     suma1 = suma2 = 0;
14:     for (i=1; i<=MAX_NUM; i++)
15:       suma1 += i;
16:     printf("El valor de suma1 es %d.\n", suma1);
17:     suma2 = fRecur(MAX_NUM);
18:     printf("El valor que devuelve fRecur() es %d.\n", suma2);
19:
20:     return 0;
21: }
22: /* definición de función */
23: int fRecur(int n)
24: {
25:     if (n == MIN_NUM)
26:       return 0;
27:     return  fRecur(n - 1) + n;
28: }
```

Después de crear y ejecutar el archivo `18L04.exe`, se mostraron en la pantalla de mi computadora los siguientes resultados:

SALIDA

```
El valor de suma1 es 5050.
El valor que devuelve fRecur() es 5050.
```

ANÁLISIS En la línea 7 del programa del listado 18.4 se declara una función recursiva, `fRecur()`, y se define en las líneas 23 a 28.

Puede ver en la definición de la función `fRecur()`que la recursión se detiene en la línea 26 si la variable entrante `n` de tipo `int` es igual al valor contenido en el nombre `enum` `MIN_NUM`. En caso contrario, la función `fRecur()` se llama a sí misma una y otra vez en la línea 27. Observe que cada vez que se llama a la función `fRecur()`, el argumento entero que se le pasa se disminuye en uno.

Veamos ahora la función `main()` del programa. El ciclo `for`, que se muestra en las líneas 14 y 15, suma enteros desde `1` hasta el valor representado por otro nombre `enum`, `MAX_NUM`. En las líneas 4 y 5 se les asignan a `MIN_NUM` y `MAX_NUM` los valores `0` y `100`, respectivamente, en una declaración `enum`. La llamada a `printf()` de la línea 16 imprime entonces la suma realizada por el ciclo.

En la línea 17 se llama a la función recursiva `fRecur()`, y se le pasa un argumento entero que inicia en el valor de `MAX_NUM`. El valor que devuelve la función `fRecur()` se asigna entonces a la variable de tipo `int`, `suma2`.

Materiał chroniony prawem autorskim

En el programa prueba.c, argumento1, argumento2 y argumento3 se llaman argumentos de la línea de comandos.

La siguiente subsección le enseña cómo recibir argumentos de la línea de comandos.

Recepción de argumentos de la línea de comandos

En la función main() hay dos argumentos integrados que se pueden usar para recibir argumentos de la línea de comandos. Por lo regular, el nombre del primer argumento es argc, y se usa para almacenar el número de argumentos de la línea de comandos. El segundo argumento se llama argv y es un apuntador a un arreglo de apuntadores de tipo char. Cada elemento del arreglo de apuntadores apunta a un argumento de la línea de comandos que es tratado como una cadena.

Declare la función main() de su programa de la siguiente manera, a fin de utilizar argc y argv:

```
main(int argc, char *argv[])
{
   . . .
}
```

Sigamos usando el ejemplo de la sección anterior. Suponga que la función main() definida en el programa prueba.c luce como la siguiente:

```
main(int argc, char *argv[])
{
   . . .
}
```

Si ejecuta el archivo del programa desde la línea de comandos de la siguiente manera:

```
prueba argumento1 argumento2 argumento3
```

el valor que recibe argc es 4 debido a que el nombre del programa mismo se cuenta como el primer argumento de la línea de comandos. De acuerdo con esto, argv[0] tiene una representación del nombre del programa, y argv[1], argv[2] y argv[3] contienen las cadenas de argumento1, argumento2 y argumento3, respectivamente.

El programa del listado 18.5 es otro ejemplo de la transferencia de argumentos de la línea de comandos a la función main().

TECLEE **LISTADO 18.5** Transferencia de argumentos de la línea de comandos a la función main()

```
1:  /* 18L05.c: Argumentos de la línea de comandos */
2:  #include <stdio.h>
3:
4:  main (int argc, char *argv[])
5:  {
```

```
6:     int i;
7:
8:     printf("El valor recibido por argc es %d.\n", argc);
9:     printf("Hay %d argumentos de la línea de comandos que se pasan a
       main().\n",
10:           argc);
11:    if(argc) {
12:        printf("El primer argumento de la línea de comandos es: %s\n",
           argv[0]);
13:        printf("Los argumentos restantes de la línea de comandos son:\n");
14:        for (i=1; i<argc; i++)
15:            printf("%s\n", argv[i]);
16:    }
17:    return 0;
18: }
```

Después de ejecutar el archivo `18L05.exe` con varios argumentos de la línea de comandos, se desplegaron los siguientes resultados en la pantalla de mi computadora (en este ejemplo escribí "¡Hola, mundo!"):

SALIDA

```
18L05.exe  ¡Hola, mundo!
El valor recibido por argc es 3.
Hay 3 argumentos de la línea de comandos que se pasan a main().
El primer argumento de la línea de comandos es: C:\app\18L05.EXE
Los argumentos restantes de la línea de comandos son:
¡Hola,
mundo!
```

ANÁLISIS La primera línea de los resultados que se muestran arriba contiene el archivo ejecutable `18L05.exe`, y los argumentos de la línea de comandos, `¡Hola, mundo!`, desde el indicador de comandos de DOS de mi computadora. La finalidad del programa del listado 18.5 es mostrarle cómo verificar el número de argumentos de la línea de comandos e imprimir las cadenas que contienen los argumentos introducidos por el usuario.

Observe que en la línea 4 se declaran dos argumentos, `argc` y `argv`, para la función `main()`. Luego, las instrucciones de las líneas 8 y 9 imprimen el valor total del número de argumentos que contiene `argc`. Si el programa se ejecutara sin argumentos de la línea de comandos, `argc` podría ser `0`, en cuyo caso `argv` sería un apuntador nulo. La línea 11 se asegura de que `argc` y `argv` contengan datos; de no ser así, devuelve `0` y termina el programa.

La línea 12 imprime la primera cadena guardada en la ubicación de memoria a la que apunta `argv[0]`. Como puede ver en los resultados, el contenido de la cadena es el nombre del archivo ejecutable del listado 18.5, más la ruta de acceso al mismo.

El ciclo `for` de las líneas 14 y 15 despliega las cadenas restantes que contienen los argumentos de la línea de comandos introducidos por el usuario. En este ejemplo escribí dos cadenas de argumentos de la línea de comandos, "`¡H9ola,`" y "`mundo!`", los cuales se muestran de nuevo en la pantalla después de la ejecución del ciclo `for`.

Materiał chroniony prawem autorskim

> `argc` y `argv` se usan normalmente como los dos argumentos integrados en la función `main()`, pero en sus declaraciones usted puede usar otros nombres para reemplazarlos. Mientras los tipos de datos sean correctos, sus funciones `main()` podrán seguir recibiendo argumentos de la línea de comandos.

Resumen

En esta lección aprendió los siguientes tipos de datos, palabras reservadas y funciones importantes de C:

- El tipo de datos `enum` (es decir, enumerado) se puede usar para declarar constantes enteras con nombre.
- De manera predeterminada, el primer nombre `enum` comienza con el valor de `0`. Cada nombre en el resto de la lista incrementa en uno el valor contenido por el nombre a su izquierda.
- En caso necesario, puede asignar valores enteros a los nombres enumerados.
- Puede crear sus propios nombres de tipos de datos con la ayuda de la palabra reservada `typedef`. Estos nombres se pueden emplear como sinónimos de los tipos de datos.
- En el ANSI C hay un archivo de encabezado llamado `stddef.h` que contiene una docena de definiciones `typedef`.
- En C se puede hacer que una función se llame a sí misma. Se dice que dicha función es recursiva.
- Puede usar argumentos de la línea de comandos para pasar información a la función `main()` de su programa.
- Hay dos argumentos integrados para la función `main()`.
- El primer argumento integrado recibe el número de argumentos de la línea de comandos introducidos por el usuario. El segundo argumento integrado es un apuntador a un arreglo de apuntadores que hace referencia a las cadenas de los argumentos de la línea de comandos.

En la siguiente lección aprenderá acerca de la recolección de variables de diferentes tipos con estructuras.

Preguntas y respuestas

P ¿Qué puede realizar con el tipo de datos `enum`?

R El tipo de datos `enum` se puede usar para declarar nombres que representen constantes enteras. Puede emplear los valores predeterminados contenidos por los nombres `enum`, o puede asignarles valores a los nombres y usarlos más adelante en el programa. El tipo de datos `enum` hace más legible el programa de C y más fácil de darle

mantenimiento debido a que puede usar como nombres de `enum` palabras que usted entienda, y sólo tendrá que ir a un lugar cuando sea necesario actualizar los valores.

P ¿Por qué necesita usar la palabra reservada `typedef`?

R Al utilizar la palabra reservada `typedef`, puede definir sus propios nombres para representar los tipos de datos en C. Puede representar tipos de datos complejos en una sola palabra y luego usar esa palabra en declaraciones de variables subsiguientes. De esta manera puede evitar errores al teclear cuando escriba una declaración compleja una y otra vez. Además, si en el futuro cambia el tipo de datos, sólo necesitará actualizar la definición `typedef` del tipo de datos, lo cual corrige todo uso de la definición `typedef`.

P ¿Ayuda una función recursiva a mejorar el rendimiento de un programa?

R En realidad no. Por lo regular, una función recursiva sólo hace más claras y sencillas las implementaciones de algunos algoritmos. Una función recursiva puede hacer más lenta la velocidad de un programa debido al exceso de las repeticiones a las llamadas a la función.

Taller

Para consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Cuáles son los valores representados por los siguientes nombres `enum`?

```
enum meses{ Ene, Feb, Mar, Abr,
            May, Jun, Jul, Ago,
            Sep, Oct, Nov, Dic };
```

2. ¿Cuáles son los valores representados por los siguientes nombres `enum`?

```
enum etiqueta { nombre1,
                nombre2 = 10,
                nombre3,
                nombre4 };
```

3. ¿Cuáles de las siguientes instrucciones tienen declaraciones de variables equivalentes?

   - `typedef long int BYTE32; BYTE32 x, y, z;`
   - `typedef char *CADENA[16]; CADENA cadena1, cadena2, cadena3;`
   - `long int x, y, z;`
   - `char *cadena1[16], *cadena2[16], *cadena3[16];`

Material chroniony prawem autorskim

4. ¿Es posible pasar argumentos de la línea de comandos a una función `main()` que tenga la siguiente definición?

```
int main(void)
{
   . . .
}
```

Ejercicios

1. Escriba un programa para imprimir los valores representados por los nombres enumerados que se declararon en la pregunta 2 del cuestionario de esta hora.
2. Dadas las siguientes declaraciones:

```
typedef char WORD;
typedef int SHORT;
typedef long LONG;
typedef float FLOAT;
typedef double DFLOAT;
```

   escriba un programa que mida el tamaño de los sinónimos de los tipos de datos.
3. Reescriba el programa del listado 18.4. Esta vez sume enteros comenzando en el valor de `MIN_NUM` en lugar de en el valor de `MAX_NUM`.
4. Escriba un programa que acepte argumentos de la línea de comandos. Si el número de argumentos de la línea de comandos, sin contar el nombre del ejecutable, es menor que dos, imprima el formato de uso del programa y pida al usuario que introduzca los argumentos de la línea de comandos. En caso contrario, despliegue todos los argumentos de la línea de comandos introducidos por el usuario.

22 Funciones de archivo especiales

23 Compilación: el preprocesador de C

24 ¿Qué sigue después?

Materiał chroniony prawem autorskim

Qué es una estructura

Como aprendió, los arreglos se pueden utilizar para reunir grupos de variables del mismo tipo. Ahora la cuestión es cómo agregar piezas de datos que no sean del mismo tipo.

La respuesta es que puede agrupar variables de diferentes tipos con un tipo de datos llamado *estructura*. En C, una estructura reúne variables de diferentes elementos de datos de manera que se les puede hacer referencia como a una sola unidad.

Hay varias diferencias importantes entre un arreglo y una estructura. Además de que los elementos de datos en una estructura pueden tener distintos tipos, cada elemento de datos tiene su propio nombre en vez del valor de un índice. De hecho, los elementos de datos de una estructura se llaman *miembros* de la estructura.

Las dos subsecciones que siguen le enseñan cómo declarar estructuras y cómo definir variables de estructuras.

Declaración de estructuras

La forma general para declarar una estructura es

```
struct etiq_estructura {
      tipo_datos1 variable1;
      tipo_datos2 variable2;
      tipo_datos3 variable3;
      .
      .
      .
      };
```

Aquí, `struct` es la palabra reservada que se usa en C para iniciar la declaración de una estructura. `etiq_estructura` es el nombre de etiqueta de la estructura. `variable1`, `variable2` y `variable3` son los miembros de la estructura. Sus tipos de datos se especifican, respectivamente, mediante `tipo_datos1`, `tipo_datos2` y `tipo_datos3`. Como puede ver, las declaraciones de los miembros que están en la declaración de la estructura deben estar encerradas entre llaves (`{` y `}`), y se debe incluir un punto y coma (`;`) al final de la declaración.

El siguiente es un ejemplo de declaración de una estructura:

```
struct auto {
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
      };
```

Aquí, `struct` se utiliza para iniciar una declaración de estructura. `auto` es el nombre de etiqueta de la estructura. En este ejemplo hay tres tipos de variables, `char`, `int` y `float`. Las variables tienen sus propios nombres, como `anio`, `modelo`, `potencia_motor` y `peso`.

Observe que un nombre de etiqueta de estructura, como `auto`, es el nombre de una estructura. El compilador emplea el nombre de etiqueta para identificar a la estructura.

Definición de variables de estructuras

Puede definir las variables de una estructura después de declararla. Por ejemplo, las siguientes variables de estructura se definen con el tipo de datos de estructura `auto` de la sección anterior:

```
struct auto sedan, pick_up, deportivo;
```

Aquí se definen tres variables de estructura, `sedan`, `pick_up` y `deportivo`, mediante la estructura `auto`. Las tres variables de estructura contienen los cuatro miembros de la estructura `auto`.

Además, en una instrucción puede combinar la declaración y la definición de la estructura de la siguiente manera:

```
struct auto {
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
      } sedan, pick_up, deportivo;
```

Aquí se definen en una sola instrucción las tres variables de estructura, `sedan`, `pick_up` y `deportivo`, junto con la estructura `auto`.

Cómo hacer referencia a miembros de estructuras con el operador de punto

Veamos ahora cómo hacer referencia a un miembro de una estructura. Por ejemplo, dadas la estructura `auto` y la variable de estructura `sedan`, puedo acceder a su miembro `anio`, y asignarle un entero de la siguiente manera:

```
sedan.anio = 1997;
```

Aquí, el nombre de la estructura y su miembro están separados por el operador de punto (`.`) para que el compilador sepa que el valor entero `1997` se asigna al miembro de nombre `anio`, el cual pertenece a la variable de estructura denominada `sedan`.

Materiał chroniony prawem autorskim

De la misma manera, la siguiente instrucción asigna la dirección de inicio del arreglo de caracteres `modelo`, que es otro miembro de la variable de estructura `sedan`, a un apuntador `aptr` de tipo `char`.

```
aptr = sedan.modelo;
```

El programa del listado 19.1 proporciona otro ejemplo de cómo hacer referencia a los miembros de una estructura.

TECLEE

LISTADO 19.1 Cómo hacer referencia a los miembros de una estructura

```
1:  /* 19L01.c Acceso a miembros de una estructura */
2:  #include <stdio.h>
3:
4:  main(void)
5:  {
6:     struct computadora {
7:        float costo;
8:        int anio;
9:        int velocidad_cpu;
10:       char tipo_cpu[16];
11:       } modelo;
12:
13:    printf("¿Tipo de CPU dentro de su computadora?\n");
14:       gets(modelo.tipo_cpu);
15:    printf("¿Velocidad (en MHz) de la CPU?\n");
16:       scanf("%d", &modelo.velocidad_cpu);
17:    printf("¿Año de fabricación de su computadora?\n");
18:       scanf("%d", &modelo.anio);
19:    printf("¿Cuál fue el precio de la computadora?\n");
20:       scanf("%f", &modelo.costo);
21:
22:    printf("Éstos son los datos que usted introdujo:\n");
23:    printf("Año: %d\n", modelo.anio);
24:    printf("Costo: $%6.2f\n", modelo.costo);
25:    printf("Tipo de CPU: %s\n", modelo.tipo_cpu);
26:    printf("Velocidad de CPU: %d MHz\n", modelo.velocidad_cpu);
27:
28:    return 0;
29: }
```

Después de ejecutar el archivo `19L01.exe` del programa del listado 19.1, y de introducir las respuestas a las preguntas, obtuve en la pantalla los siguientes resultados (en ellos, los caracteres o números en negritas corresponden a las respuestas que introduje desde mi teclado):

SALIDA

```
¿Tipo de CPU dentro de su computadora?
Pentium
¿Velocidad (en MHz) de la CPU?
100
¿Año de fabricación de su computadora?
1996
¿Cuál fue el precio de la computadora?
1234.56
Éstos son los datos que usted introdujo:
Año: 1996
Costo: $1234.56
Tipo de CPU: Pentium
Velocidad de CPU: 100 MHz
```

ANÁLISIS

La finalidad del programa del listado 19.1 es mostrarle cómo hacer referencia a los miembros de una estructura. Como puede ver, en el programa hay una estructura llamada `modelo` que se define con un tipo `struct computadora` en las líneas 6 a 11. La estructura tiene una variable `float`, dos variables `int` y un arreglo `char`. La instrucción de la línea 13 solicita al usuario que introduzca el tipo de CPU (unidad central de procesamiento) que se utiliza dentro de su computadora. Luego, la línea 14 recibe la cadena del tipo de CPU que introdujo el usuario y la guarda en un arreglo de caracteres denominado `tipo_cpu`. Debido a que `tipo_cpu` es un miembro de la estructura `modelo`, en la línea 14 se usa la expresión `modelo.tipo_cpu` para hacer referencia al miembro de la estructura. Observe que en la expresión se emplea el operador de punto (`.`) para separar los dos nombres.

Las líneas 15 y 16 solicitan la velocidad de la CPU y almacenan el valor entero que introdujo el usuario en la variable `velocidad_cpu` de tipo `int`, que es otro miembro de la estructura `modelo`. Observe que a la expresión `modelo.velocidad_cpu`, que está dentro de la función `scanf()` de la línea 16, se le pone como prefijo el operador de dirección (`&`), debido a que el argumento debe ser un apuntador de tipo `int`.

Asimismo, las líneas 17 y 18 reciben el valor del año de fabricación de la computadora del usuario, y las líneas 19 y 20 obtienen el número del costo de la computadora. Después de la ejecución, la variable `anio` de tipo `int` y la variable `costo` de tipo `float` de la estructura `modelo` contienen los valores correspondientes que introdujo el usuario.

Luego, las líneas 23 a 26 imprimen los valores de todos los miembros de la estructura `modelo`. A partir de los resultados puede saber que se accedió a cada miembro de la estructura y que se le asignó un número o una cadena en forma correcta.

Inicialización de estructuras

Puede inicializar una estructura por medio de una lista de datos llamados *inicializadores*. En una lista de datos se emplean comas para separar sus elementos.

Materiał chroniony prawem autorskim

TECLEE **LISTADO 19.3** Cómo pasar una estructura a una función

```
1:  /* 19L03.c Cómo pasar una estructura a una función */
2:  #include <stdio.h>
3:
4:  struct computadora {
5:     float costo;
6:     int anio;
7:     int velocidad_cpu;
8:     char tipo_cpu[16];
9:  };
10: /* crea sinónimo */
11: typedef struct computadora EC;
12: /* declaración de función */
13: EC RecibeDatos(EC s);
14:
15: main(void)
16: {
17:    EC modelo;
18:
19:    modelo = RecibeDatos(modelo);
20:    printf("Éstos son los datos que usted introdujo:\n");
21:    printf("Año: %d\n", modelo.anio);
22:    printf("Costo: $%6.2f\n", modelo.costo);
23:    printf("Tipo de CPU: %s\n", modelo.tipo_cpu);
24:    printf("Velocidad de CPU: %d MHz\n", modelo.velocidad_cpu);
25:
26:    return 0;
27: }
28: /* definición de función */
29: EC RecibeDatos(EC s)
30: {
31:    printf("¿Tipo de CPU dentro de su computadora?\n");
32:       gets(s.tipo_cpu);
33:    printf("¿Velocidad (en MHz) de la CPU?\n");
34:       scanf("%d", &s.velocidad_cpu);
35:    printf("¿Año de fabricación de su computadora?\n");
36:       scanf("%d", &s.anio);
37:    printf("¿Cuál fue el precio de la computadora?\n");
38:       scanf("%f", &s.costo);
39:    return s;
40: }
```

Después de ejecutar el archivo `19L03.exe`, y escribir mis respuestas, obtuve los siguientes resultados, que son los mismos del programa del listado 19.1:

Materiał chroniony prawem autorskim

Cómo hacer referencia a estructuras mediante apuntadores

Así como en una función puede pasar un apuntador que haga referencia a un arreglo, también puede pasar un apuntador que señale a una estructura.

Sin embargo, a diferencia de transferir una estructura a una función, lo cual envía una copia completa de la estructura a la función, pasar un apuntador a una estructura sólo envía la dirección de la estructura a la función. La función puede entonces usar la dirección para acceder directamente a los miembros de la estructura, con lo cual evita la carga de duplicar la estructura. Por lo tanto, es más eficiente pasar un apuntador a una estructura que pasar la estructura misma a una función. De acuerdo con esto, se puede reescribir el programa del listado 19.3 para pasarle a la función `RecibeDatos()` un apuntador que apunte a la estructura. El programa modificado se muestra en el listado 19.4.

TECLEE **LISTADO 19.4** Cómo pasarle a una función un apuntador que apunte a una estructura

```
1:  /* 19L04.c Cómo apuntar a una estructura */
2:  #include <stdio.h>
3:
4: struct computadora {
5:     float costo;
6:     int anio;
7:     int velocidad_cpu;
8:     char tipo_cpu[16];
9: };
10:
11: typedef struct computadora EC;
12:
13: void RecibeDatos(EC *aptr_s);
14:
15: main(void)
16: {
17:     EC modelo;
18:
19:     RecibeDatos(&modelo);
20:     printf("Éstos son los datos que usted introdujo:\n");
21:     printf("Año: %d\n", modelo.anio);
22:     printf("Costo: $%6.2f\n", modelo.costo);
23:     printf("Tipo de CPU: %s\n", modelo.tipo_cpu);
24:     printf("Velocidad de CPU: %d MHz\n", modelo.velocidad_cpu);
25:
26:     return 0;
27: }
28: /* definición de función */
29: void RecibeDatos(EC *aptr_s)
30: {
```

Material chroniony prawem autorskim

Otro ejemplo es la expresión `&(*aptr_s).velocidad_cpu` de la línea 34, la cual evalúa la dirección del miembro de estructura `velocidad_cpu` al que apunta `aptr_s`. Otra vez aparece entre paréntesis el apuntador con el operador de indirección `*aptr_s`.

La siguiente subsección le muestra cómo emplear el operador de flecha (`->`) para hacer referencia a un miembro de una estructura mediante un apuntador.

Cómo hacer referencia a un miembro de una estructura mediante ->

Puede emplear el operador de flecha `->` para hacer referencia a un miembro de una estructura asociado con un apuntador que apunta a la estructura.

Por ejemplo, puede reescribir la expresión `(*aptr_s).tipo_cpu` del listado 19.4 con lo siguiente:

```
aptr_s -> tipo_cpu
```

o puede reemplazar la expresión `&(*aptr_s).velocidad_cpu` con lo siguiente:

```
&(aptr_s->velocidad_cpu)
```

Ya que el operador `->` tiene una precedencia mayor que el operador `&`, puede omitir los paréntesis en la expresión anterior, y escribir en su lugar lo siguiente:

```
&aptr_s->velocidad_cpu
```

Debido a su mayor claridad, el operador `->` se usa con más frecuencia que el operador de punto en programas que acceden a miembros de estructuras mediante apuntadores a estructuras. Más adelante, el ejercicio 3 de esta hora le da la oportunidad de reescribir el programa completo del listado 19.4 utilizando el operador `->`.

Arreglos de estructuras

En C, usted puede declarar un arreglo de estructuras anteponiendo el nombre de la estructura al nombre del arreglo. Por ejemplo, dada una estructura con el nombre de etiqueta `x`, la siguiente instrucción:

```
struct x arreglo_de_estructuras[8];
```

declara un arreglo, denominado `arreglo_de_estructuras`, de la estructura `x`. El arreglo tiene ocho elementos, cada uno de los cuales es una instancia de la estructura `x`.

El programa que se muestra en el listado 19.5 muestra cómo usar un arreglo de estructuras para imprimir dos haikus japoneses y los nombres de sus autores.

Materiał chroniony prawem autorskim

Listado 19.5 continuación

```
47:     printf("%s\n", aptr_s->cadena3);
48:     printf("--- %s\n", aptr_s->autor);
49:     printf("    (%d-%d)\n\n", aptr_s->nacimiento, aptr_s->muerte);
50: }
```

Después de ejecutar el archivo `19L05.exe` del programa del listado 19.5, vi los dos haikus japoneses desplegados en la pantalla de mi computadora:

Salida

```
Llevándome con ella
mi sombra regresa a casa
tras mirar la luna.
--- Sodo
    (1641-1716)

Sopla un viento de tormenta
de entre las hierbas
crece la luna llena.
--- Chora
    (1729-1781)
```

Análisis En las líneas 4 a 11 del Listado 19.5 se declara un tipo de datos de estructura, con el nombre de etiqueta `haiku`. El tipo de datos de estructura contiene como miembros dos variables `int` y cuatro arreglos `char`. La instrucción de la línea 13 crea un sinónimo, `HK`, para el tipo de datos `struct haiku`.

Después, en las líneas 19 a 34 se declara un arreglo de dos elementos, `poema`, y se inicializa con dos piezas de haikus escritos por Sodo y Chora. La siguiente es una copia de los dos haikus de `poema`:

```
"Llevándome con ella",
"mi sombra regresa a casa",
"tras mirar la luna."
```

y

```
"Sopla un viento de tormenta",
"de entre las hierbas",
"crece la luna llena."
```

El inicializador también incluye los nombres de los autores y sus fechas de nacimiento y muerte (vea las líneas 20 a 22 y 27 a 29). Observe que el arreglo `poema`, declarado con el tipo `HK`, es en realidad un arreglo de la estructura `haiku`.

En el ciclo `for` de las líneas 37 y 38 se llama dos veces a la función `ExhibeDatos()`. En cada llamada se pasa la dirección de un elemento de `poema` a la función `ExhibeDatos()`. De acuerdo con la definición de la función en las líneas 43 a 50, `ExhibeDatos()` imprime tres cadenas de un haiku, el nombre del autor y el periodo en que vivió.

Materiał chroniony prawem autorskim

LISTADO 19.6 continuación

```
26:         { 1,
27:           "Mercadotecnia",
28:           "Gerente"
29:         },
30:         1,
31:         "B. Smith"
32:     };
33:
34:     printf("Ésta es una muestra:\n");
35:     ExhibeInf(&info);
36:
37:     CapturaInf(&info);
38:
39:     printf("\nÉstos son los datos que usted introdujo:\n");
40:     ExhibeInf(&info);
41:
42:     return 0;
43: }
44: /* definición de función */
45: void ExhibeInf(EMP *aptr)
46: {
47:     printf("Nombre: %s\n", aptr->nombre);
48:     printf("No. de ID: %04d\n", aptr->id);
49:     printf("Nombre Depto.: %s\n", aptr->d.nombre);
50:     printf("Clave Depto.: %02d\n", aptr->d.clave);
51:     printf("Puesto: %s\n", aptr->d.puesto);
52: }
53: /* definición de función */
54: void CapturaInf(EMP *aptr)
55: {
56:     printf("\n Introduzca sus datos:\n");
57:     printf("Nombre:\n");
58:         gets(aptr->nombre);
59:     printf("Puesto:\n");
60:         gets(aptr->d.puesto);
61:     printf("Nombre Depto.:\n");
62:         gets(aptr->d.nombre);
63:     printf("Clave Depto.:\n");
64:         scanf("%d", &(aptr->d.clave));
65:     printf("No. de ID:\n");
66:         scanf("%d", &(aptr->id));
67: }
```

Durante la ejecución del archivo `19L06.exe`, primero se imprimió el contenido de la estructura anidada. Luego introduje mi información de empleado, la cual aparece en negritas en los resultados y se despliega también en la pantalla:

SALIDA

```
Ésta es una muestra:
Nombre: B. Smith
No. de ID: 0001
Nombre Depto.: Mercadotecnia
Clave Depto.: 01
Puesto: Gerente

Introduzca sus datos:
Nombre:
T. Zhang
Puesto:
Ingeniero
Nombre Depto.:
Investigación y Desarrollo
Clave Depto.:
3
No. de ID:
1234

Éstos son los datos que usted introdujo:
Nombre: T. Zhang
No. de ID: 1234
Nombre Depto.: Investigación y Desarrollo
Clave Depto.: 03
Puesto: Ingeniero
```

ANÁLISIS Hay dos tipos de datos de estructuras en el listado 19.6. El primero, llamado `departamento`, se declara en las líneas 4 a 8. El segundo, `empleado`, declarado en las líneas 12 a 16, contiene un miembro del tipo de datos de estructura `departamento`. Por lo tanto, el tipo de datos de estructura `empleado` es un tipo de datos de estructura anidada.

En las dos instrucciones `typedef` de las líneas 10 y 18 se crean dos sinónimos, `DEPTO` para el tipo de datos `struct departamento`, y `EMP` para el tipo de datos `struct empleado`, respectivamente. En el programa hay dos funciones, `ExhibeInf()` y `CapturaInf()`, cuyos prototipos se declaran con un apuntador a un `EMP` como argumento (vea las líneas 20 y 21).

Las instrucciones de las líneas 25 a 32 inicializan una estructura anidada, la cual se llama `info` y tiene un tipo de datos `EMP`. Observe que las llaves anidadas ({ y }) de las líneas 26 y 29 encierran los inicializadores de la estructura de `DEPTO` que está anidada dentro de la estructura `info`.

Luego, la instrucción de la línea 35 despliega el contenido inicial de la estructura anidada `info`, por medio de la llamada a la función `ExhibeInf()`. La línea 37 llama a la función `CapturaInf()` para solicitar al usuario que introduzca sus datos de empleado y los guarda después en la estructura `info`. En la línea 40 se llama de nuevo a la función `ExhibeInf()` para desplegar la información que introdujo el usuario y que ahora se guarda en la estructura anidada.

Las definiciones de las dos funciones, `ExhibeInf()` y `CapturaInf()`, se presentan en las líneas 45 a 52 y 54 a 67, respectivamente.

P ¿Por qué es más eficiente pasarle a una función un apuntador que hace referencia a una estructura?

R Cuando se pasa la estructura completa a una función, se hace una copia de la estructura y se guarda en un bloque de almacenamiento temporal. Después de que la función modifica la copia, ésta se debe devolver y escribir de nuevo en el espacio de almacenamiento que contiene la información original de la estructura. Por otra parte, pasarle a la función un apuntador que señala a una estructura, simplemente le transfiere la dirección de la estructura en lugar de la estructura completa. La función puede entonces acceder a la ubicación de memoria original de la estructura y modificar el contenido de la estructura sin duplicarla en un almacenamiento temporal. Por lo tanto, es más eficiente pasarle a una función un apuntador que señale a una estructura, que pasarle la estructura misma.

Taller

Para ayudarle a consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Qué está mal en la siguiente declaración de estructura?

```
struct auto {
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
      }
```

2. ¿Cuántas variables de estructura se definen en la siguiente instrucción?

```
struct x {int y; char z} u, v, w;
```

3. Dada la declaración de estructura

```
struct auto {
      int anio;
      char modelo[8]};
```

agregue dos modelos de autos, `Taurus` y `Accord`, fabricados en `1997`; inicialice un arreglo, `carro`, de dos elementos de estructuras `auto`.

Materiał chroniony prawem autorskim

Qué es una unión

Una *unión* es un bloque de memoria que se usa para almacenar elementos de datos de diferentes tipos. En C, una unión es similar a una estructura, excepto que los elementos de datos guardados en la unión están sobrepuestos para que compartan la misma ubicación de memoria. Es decir, todos comparten la misma dirección de inicio, a diferencia de una estructura en donde cada miembro tiene su propia área de memoria. En las secciones siguientes verá más detalles sobre las diferencias entre uniones y estructuras. Pero antes, veamos la sintaxis de las uniones.

Declaración de uniones

La sintaxis para declarar una unión es similar a la de una estructura. El siguiente es un ejemplo de una declaración de unión:

```
union auto{
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
};
```

Aquí, `union` es la palabra reservada que especifica el tipo de datos de la unión. `auto` es el nombre de etiqueta de la unión. Las variables `anio`, `modelo`, `potencia_motor` y `peso` son miembros de la unión y se declaran dentro de las llaves (`{` y `}`). La declaración de la unión termina con un punto y coma (`;`).

Al igual que el nombre de etiqueta de una estructura, el nombre de etiqueta de una unión es el que nombra la unión y el que usa el compilador para identificarla.

Definición de variables de unión

Después de declarar una unión, puede definir variables de unión. Por ejemplo, las siguientes variables de unión se definen con la unión etiquetada como `auto` de la sección anterior:

```
union auto sedan, pickup, deportivo;
```

Aquí, las tres variables, `sedan`, `pickup` y `deportivo`, están definidas como variables de unión.

Desde luego, usted puede declarar una unión y definir las variables de la unión en una sola instrucción. Por ejemplo, puede reescribir la declaración y definición de unión anteriores de la siguiente manera:

```
union auto {
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
} sedan, pickup, deportivo;
```

Materiał chroniony prawem autorskim

LISTADO 20.1 continuación

```
9:        double precio;
10:    } platillo;
11:
12:    printf("Contenido asignado a la unión por separado:\n");
13:    /* referencia al nombre */
14:    strcpy(platillo.nombre, "Pollo agridulce");
15:    printf("Nombre del platillo: %s\n", platillo.nombre);
16:    /* referencia al precio */
17:    platillo.precio = 9.95;
18:    printf("Precio del platillo: %5.2f\n", platillo.precio);
19:
20:    return 0;
21: }
```

Después de ejecutar el archivo `20L01.exe` del programa del listado 20.1, se mostraron los siguientes resultados en la pantalla de mi computadora:

SALIDA

```
Contenido asignado a la unión por separado:
Nombre del platillo: Pollo agridulce
Precio del platillo: 9.95
```

ANÁLISIS La finalidad del programa del listado 20.1 es mostrarle cómo hacer referencia a los miembros de una unión con el operador de punto.

En las líneas 7 a 10, dentro de la función `main()`, se define primero una unión, denominada `platillo`, con el tipo de datos de unión `menu`. Hay dos miembros en la unión, `nombre` y `precio`.

Luego, la instrucción de la línea 14 copia la cadena `"Pollo agridulce"` dentro del arreglo de caracteres `nombre`, que es uno de los miembros de la unión. Observe que la expresión `platillo.nombre` se usa como el primer argumento para la función `strcpy()` de la línea 14. Cuando el compilador ve la expresión, sabe que usted desea hacer referencia a la ubicación de memoria `nombre` que es un miembro de la unión `platillo`

`strcpy()` es una función de C que copia el contenido de una cadena, a la que apunta el segundo argumento de la función, en el espacio de memoria al que apunta el primer argumento de la función. Antes de llamar a la función `strcpy()`, incluí en el programa el archivo de encabezado `string.h` (vea la línea 3).

La línea 15 imprime el contenido copiado al arreglo `nombre`, usando una vez más la expresión `platillo.nombre`.

La instrucción de la línea 17 asigna el valor `9.95` a la variable `precio` de tipo `double`, que es otro miembro de la unión `platillo`. Observe que se usa la expresión `platillo.precio` para hacer referencia al miembro de la unión. Después, la línea 18 despliega el valor asignado a `precio` por medio de la llamada a la función `printf()`, y le pasa a dicha función la expresión `platillo.precio` como un argumento.

De acuerdo con los resultados que se muestran en la salida, se hizo referencia a los dos miembros de la unión `platillo`, `nombre` y `precio`, y se les asignaron los valores correspondientes en forma correcta.

Uniones en comparación con estructuras

Tal vez haya notado que le asigné un valor a un miembro de la unión `platillo` del listado 20.1, y luego imprimí de inmediato el valor asignado antes de pasar al siguiente miembro de la unión. En otras palabras, no asigné valores a todos los miembros de la unión a la vez, antes de imprimir cada valor asignado a cada uno de los miembros de la unión.

Hice esto a propósito por la razón que se explica en la siguiente sección. Así que continúe leyendo. (Cuando reescriba el programa del listado 20.1, en el ejercicio 1 de esta lección, verá resultados diferentes, ya que intercambiará el orden de las instrucciones de las líneas 15 y 17.)

Inicialización de una unión

Como se mencionó anteriormente en esta lección, los elementos de una unión están sobrepuestos en la misma ubicación de memoria. En otras palabras, todos los miembros de una unión comparten en diferentes momentos la ubicación inicial de memoria de la unión. El tamaño de una unión es, como mínimo, tan grande como el elemento de datos más extenso de la lista de miembros de la unión. De esta forma, es lo suficientemente grande para contener, en un momento determinado, cualquier miembro de la unión. Por lo tanto, no tiene caso inicializar juntos todos los miembros de una unión, ya que el valor del último miembro inicializado sobreescribe el valor del miembro precedente. Usted inicializa un miembro de una unión sólo cuando lo va a utilizar. El valor que contiene la unión es siempre el último valor asignado a un miembro de la misma.

Por ejemplo, si declara y define una unión en una máquina de 16 bits (es decir, el tipo de datos `int` es de 2 bytes de longitud) de la siguiente manera:

```
union u {
   char ch;
   int x;
} una_union;
```

entonces, la siguiente instrucción inicializa la variable `ch` de tipo `char` con la siguiente constante de carácter `'H'`:

```
una_union.ch = 'H';
```

y el valor contenido por la unión `una_union` es la constante de carácter `'H'`. Sin embargo, si se inicializa la variable `x` de tipo `int` mediante la siguiente instrucción:

```
una_union.x = 365;
```

```
11:
12:     /* inicializa anio_inicial */
13:     info.anio_inicial = 1997;
14:     /* inicializa cve_depto */
15:     info.cve_depto = 8;
16:     /* inicializa id */
17:     info.num_id = 1234;
18:
19:     /* despliega el contenido de la unión */
20:     printf("Año inicial:  %d\n", info.anio_inicial);
21:     printf("Clave del Depto.:  %d\n", info.cve_depto);
22:     printf("No. de ID:  %d\n", info.num_id);
23:
24:     return 0;
25: }
```

SALIDA Después de crear y ejecutar el archivo `20L02.exe`, se desplegaron en la pantalla los siguientes resultados:

```
Año inicial:  1234
Clave del Depto.:   1234
No. de ID:   1234
```

ANÁLISIS Como puede ver en el listado 20.2, una unión llamada `info` tiene tres variables miembro de tipo `int`, `anio_inicial`, `cve_depto` y `num_id` (vea las líneas 6 a 10). Luego, a estos tres miembros de la unión se les asignan diferentes valores de manera consecutiva en las líneas 13, 15 y 17. En las líneas 20 a 22 se intenta imprimir los valores asignados a los tres miembros. Sin embargo, los resultados muestran que todos los miembros de la unión `info` tienen el mismo valor, `1234`, que es el entero que se asignó al tercer miembro de la unión, `num_id`. La unión `info` contiene, en efecto, el último valor asignado a sus miembros.

El tamaño de una unión

Le he dicho que todos los miembros de una unión comparten la misma ubicación de memoria. El tamaño de una unión es, como mínimo, tan grande como su miembro más grande.

En contraste con una unión, todos los miembros de una estructura se pueden inicializar juntos sin sobreescribirlos. Esto se debe a que cada miembro de una estructura tiene su propio espacio de almacenamiento en memoria. El tamaño de una estructura es, como mínimo, igual a la suma del tamaño de sus miembros, en vez del tamaño del miembro más grande, como es el caso de la unión.

El listado 20.3 contiene un programa que mide el tamaño de una unión así como el tamaño de una estructura. La estructura tiene exactamente los mismos miembros que la unión.

Material chroniony prawem autorskim

TECLEE **LISTADO 20.3** Cómo medir el tamaño de una unión

```
1:  /* 20L03.c Tamaño de una unión */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main(void)
6: {
7:     union u {
8:         double x;
9:         int y;
10:     } una_union;
11:
12:     struct s {
13:         double x;
14:         int y;
15:     } una_estructura;
16:
17:     printf("Tamaño del tipo double: %d bytes\n",
18:             sizeof(double));
19:     printf("Tamaño del tipo int:    %d bytes\n",
20:             sizeof(int));
21:
22:     printf("Tamaño de una_union:  %d bytes\n",
23:             sizeof(una_union));
24:     printf("Tamaño de una_estructura: %d bytes\n",
25:             sizeof(una_estructura));
26:
27:     return 0;
28: }
```

En su máquina, el compilador podría generar varios mensajes de advertencia, como "unreferenced local variables". Esto se debe a que en el programa no se inicializan la unión `una_union` y la estructura `una_estructura`. Puede ignorar estos mensajes ya que no se aplican a lo que estamos haciendo aquí. Después de crear y ejecutar el archivo `20L03.exe`, se desplegaron los siguientes resultados en la pantalla de mi computadora:

SALIDA

```
Tamaño del tipo double: 8 bytes
Tamaño del tipo int:    2 bytes
Tamaño de una_union:  8 bytes
Tamaño de una_estructura: 10 bytes
```

ANÁLISIS La finalidad del programa del listado 20.3 es mostrarle la diferencia entre las asignaciones de memoria de una unión y una estructura, aun cuando ambas tienen los mismos miembros.

En las líneas 7 a 10 se define una unión denominada `una_union`; ésta tiene dos miembros, una variable `x` de tipo `double`, y una variable `y` de tipo `int`. Además, en las líneas 12 a 15 se define una estructura llamada `una_estructura`, que consta también de dos miembros, una variable `x` de tipo `double`, y una variable `y` de tipo `int`.

Las instrucciones de las líneas 17 a 20 miden primero el tamaño de los tipos de datos `double` e `int` en la máquina en la que se está operando. Por ejemplo, en mi máquina, el tamaño del tipo de datos `double` es de 8 bytes y el del tipo de datos `int` es de 2 bytes.

Las líneas 22 a 25 miden el tamaño de la unión `una_union` y de la estructura `una_estructura`, respectivamente. Puede ver en los resultados que el tamaño de `una_union` es de 8 bytes.

Por otra parte, en mi máquina, el tamaño de la estructura es de 10 bytes, debido a que debe haber suficiente espacio de memoria para los tipos de datos `double` e `int` de la estructura.

Uso de uniones

Ahora nos concentraremos en las aplicaciones de uniones. Básicamente, hay dos tipos de aplicaciones de uniones, mismas que se presentan en las dos secciones siguientes.

Cómo hacer referencia a la misma ubicación de memoria con diferentes miembros

La primera aplicación de las uniones es hacer referencia a la misma ubicación de memoria con diferentes miembros de la unión.

Para tener una mejor idea acerca de cómo hacer referencia a la misma memoria con diferentes miembros de la unión, veamos el programa del listado 20.4, el cual utiliza los dos miembros de una unión para hacer referencia a la misma ubicación de memoria. (Se asume que el tipo de datos `char` es de 1 byte de longitud y que el tipo de datos `int` tiene 2 bytes de longitud.)

TECLEE **LISTADO 20.4** Cómo hacer referencia a la misma ubicación de memoria con diferentes miembros de la unión

```
1:  /* 20L04.c: Cómo hacer referencia a la misma ubicación de memoria en
                diferentes miembros de la unión */
2:  #include <stdio.h>
3:
4:  union u{
5:     char ch[2];
6:     int num;
7:  };
8:
9:  int InicializaUnion(union u val);
10:
11: main(void)
12: {
13:    union u val;
14:    int x;
15:
```

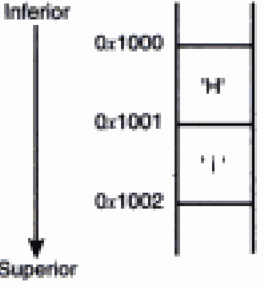
continúa

En la línea 20 se obtiene el byte de orden superior de `num` desplazando la variable `x` 8 bits hacia la derecha, es decir, utilizando el operador de desplazamiento a la derecha en la expresión `x >> 8`. (El operador a nivel de bits (`&`) y el operador de desplazamiento (`>>`) se presentaron en la hora 8, "Uso de operadores condicionales".)

Puede ver en los resultados que el contenido de la unión `val` se muestra correctamente en la pantalla.

La figura 20.2 muestra la ubicación en memoria de las dos constantes de carácter.

Figura 20.2
Las ubicaciones de memoria de las dos constantes de carácter.



Hay dos formatos para almacenar una cantidad de varios bytes, como la variable `num` de tipo `int` del listado 20.4. Dichos formatos son el *little-endian* y el *big-endian*.

En el formato little-endian, los bytes de orden superior de una cantidad de varios bytes se almacenan en las direcciones de memoria más altas y los bytes de orden inferior se guardan en las direcciones más bajas. El formato little-endian lo utilizan los microprocesadores 80x86 de Intel. La CPU de mi computadora es un microprocesador Pentium, que es uno de los miembros de la familia 80x86. Por lo tanto, en el listado 20.4, la constante de carácter `'H'`, que es un byte de orden inferior, se almacena en la dirección más baja. La `'i'` se almacena en la dirección más alta, ya que es un byte de orden superior.

En el formato big-endian es exactamente lo contrario. Los bytes de orden superior se almacenan en las direcciones más bajas y los de orden inferior en las direcciones más altas. La familia de microprocesadores 68000 de Motorola utiliza el formato big-endian.

Cómo hacer flexibles las estructuras

La segunda aplicación de las uniones consiste en anidar una unión dentro de una estructura para que la estructura pueda contener diferentes tipos de valores.

Por ejemplo, suponga que desea escribir un programa que le solicite al usuario el nombre de una compañía de cable o de antena parabólica vía satélite que proporcione servicio al usuario. Suponga que el usuario utiliza en casa uno de los dos servicios, pero no ambos. Entonces, si define dos arreglos de caracteres para almacenar los nombres de la compañía

Materiał chroniony prawem autorskim

de cable y de la compañía de antena parabólica, uno de los arreglos estará vacío. En este caso, puede declarar una unión con los dos arreglos de caracteres como miembros, para que la unión pueda contener el nombre de una compañía de cable o el de una de antena parabólica, dependiendo de la entrada del usuario. El listado 20.5 muestra cómo escribir un programa con dicha unión.

TECLEE **LISTADO 20.5** Cómo hacer flexible una estructura

```
1:  /* 20L05.c: Uso de uniones */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  struct encuesta {
6:      char nombre[20];
7:      char c_a_x;
8:      int edad;
9:      int horas_por_semana;
10:     union {
11:         char servicio_cable[16];
12:         char servicio_antena[16];
13:     } proveedor;
14: };
15:
16: void CapturaDatos(struct encuesta *s);
17: void ExhibeDatos(struct encuesta *s);
18:
19: main(void)
20: {
21:     struct encuesta tv;
22:
23:     CapturaDatos(&tv);
24:     ExhibeDatos(&tv);
25:
26:     return 0;
27: }
28: /* definición de función */
29: void CapturaDatos(struct encuesta *aptr)
30: {
31:     char respuesta[3];
32:
33:     printf("¿Utiliza TV por cable en casa? (Sí o No)\n");
34:         gets(respuesta);
35:     if ((respuesta[0] == 'S') ||
36:          (respuesta[0] == 's')){
37:         printf("Escriba el nombre de su compañía de cable:\n");
38:         gets(aptr->proveedor.servicio_cable);
39:         aptr->c_a_x = 'c';
40:     } else {
41:         printf("¿Usa una antena parabólica? (Sí o No)\n");
```

```
42:         gets(respuesta);
43:       if ((respuesta[0] == 'S') ||
44:            (respuesta[0] == 's')){
45:         printf("Escriba el nombre de su compañía de antena parabólica:\n");
46:         gets(aptr->proveedor.servicio_antena);
47:         aptr->c_a_x = 'a';
48:       } else {
49:         aptr->c_a_x = 'x';
50:       }
51:     }
52:     printf("Escriba su nombre:\n");
53:       gets(aptr->nombre);
54:     printf("Su edad:\n");
55:       scanf("%d", &aptr->edad);
56:     printf("¿Cuántas horas dedica a la semana a ver TV?:\n");
57:       scanf("%d", &aptr->horas_por_semana);
58: }
59: /* definición de función */
60: void ExhibeDatos(struct encuesta *aptr)
61: {
62:     printf("\nÉstos son los datos que usted introdujo:\n");
63:     printf("Nombre: %s\n", aptr->nombre);
64:     printf("Edad:   %d\n", aptr->edad);
65:     printf("Horas por semana: %d\n", aptr->horas_por_semana);
66:     if (aptr->c_a_x == 'c')
67:       printf("Su compañía de cable es: %s\n",
68:         aptr->proveedor.servicio_cable);
69:     else if (aptr->c_a_x == 'a')
70:       printf("Su compañía de antena parabólica es: %s\n",
71:         aptr->proveedor.servicio_antena);
72:     else
73:       printf("Usted no tiene servicio de cable ni de antena
parabólica.\n");
74:     printf("\n¡Gracias y adiós!\n");
75: }
```

Durante la ejecución del archivo 20L05.exe, escribí mis respuestas a la encuesta y se desplegaron en la pantalla los siguientes resultados (mis respuestas aparecen en negritas):

Salida

```
¿Utiliza TV por cable en casa? (Sí o No)
No
¿Usa una antena parabólica? (Sí o No)
Sí
Escriba el nombre de su compañía de antena parabólica:
Compañía ABCD
Escriba su nombre:
Tony Zhang
Su edad:
30
¿Cuántas horas dedica a la semana a ver TV?:
8
```

Materiał chroniony prawem autorskim


```
Éstos son los datos que usted introdujo:
Nombre: Tony Zhang
Edad: 30
Horas por semana: 8
Su compañía de antena parabólica es: Compañía ABCD

¡Gracias y adiós!
```

ANÁLISIS Como puede ver, en las líneas 5 a 14 se declara un tipo de datos `struct` con el nombre de etiqueta `encuesta`, y en su interior se anida una unión denominada `proveedor` que tiene dos miembros, los arreglos `servicio_cable` y `servicio_antena`. Ambos miembros de la unión se emplean para contener los nombres de compañías de cable o de antena parabólica, dependiendo de los datos que introduce el usuario.

Las instrucciones de las líneas 16 y 17 declaran dos funciones, `CapturaDatos()` y `ExhibeDatos()`, a las cuales se pasa un apuntador que tiene como argumento `struct encuesta`.

Dentro de la función `main()`, en la línea 21 se define una estructura llamada `tv`. Luego, en las líneas 23 y 24 se llama a las funciones `CapturaDatos()` y `ExhibeDatos()` con la dirección de la estructura `tv` como argumento.

Las líneas 29 a 58 contienen la definición de la función `CapturaDatos()`, la cual pide al usuario que introduzca la información adecuada con base en las preguntas de la encuesta. Bajo la suposición que hicimos antes, el usuario puede utilizar televisión por cable o antena parabólica, pero no ambas. Si el usuario usa cable, la línea 38 recibe el nombre de la compañía de cable que escribe el usuario y lo guarda en el espacio de memoria al que hace referencia uno de los miembros de la unión `proveedor, servicio_cable`.

Si el usuario utiliza una antena parabólica, la línea 46 almacena el nombre de la compañía que introduce el usuario en la misma ubicación de la unión `proveedor`. Pero esta vez se usa el nombre de otro miembro de la unión, `servicio_antena`, para hacer referencia a la ubicación de memoria. Ahora ya sabe cómo ahorrar memoria poniendo dos elementos de datos excluyentes dentro de una unión.

De hecho, el programa maneja otra situación, en la que el usuario no tiene cable ni antena parabólica. En este caso, se asigna la constante de carácter `'x'` a la variable `c_a_x` de tipo `char`, la cual es miembro de la estructura.

Las líneas 60 a 75 dan la definición de la función `ExhibeDatos()` que imprime de nuevo en la pantalla la información introducida por el usuario. Los resultados que se presentan aquí son una muestra de lo que hice al ejecutar en mi máquina el programa del listado 20.5.

Definición de campos de bits mediante `struct`

En esta sección visitará de nuevo a una vieja amiga, la palabra reservada `struct`, para declarar un objeto muy pequeño. Luego utilizará el objeto con uniones.

Como usted sabe, `char` es el tipo de datos más pequeño en C. El tipo de datos `char` es de un byte de longitud. Sin embargo, con la ayuda de la palabra reservada `struct`, puede declarar un objeto más pequeño, un *campo de bits*, lo cual le permite acceder a un solo bit. Un bit puede contener sólo dos valores, `1` o `0`.

La forma general para declarar y definir campos de bits es

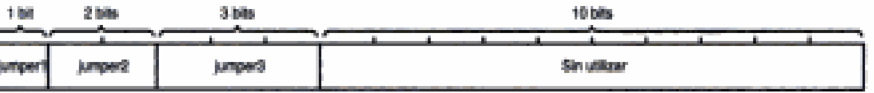
```
struct nombre_etiqueta {
   tipo_datos nombre1: longitud1;
   tipo_datos nombre2: longitud2;
   . . .
   tipo_datos nombreN: longitudN;
} lista_de_variables;
```

Aquí, la palabra reservada `struct` se usa para iniciar la declaración. *`nombre_etiqueta`* es el nombre de etiqueta del tipo de datos `struct`. *`tipo_datos`*, que puede ser `int`, un `signed int` o `signed int`, especifica el tipo de datos de los campos de bits. *`nombre1`*, *`nombre2`* y *`nombreN`* son los nombres de los campos de bits. *`longitud1`*, *`longitud2`* y *`longitudN`* indican las longitudes de los campos de bits, especificadas en bits. La longitud de cualquier campo de bits no debe exceder la longitud del tipo de datos `int`. `lista_de_variables` contiene los nombres de las variables del campo de bits.

Por ejemplo, la siguiente instrucción define una estructura denominada `jumpers` con tres miembros de campo de bits:

```
struct bf {
   int jumper1: 1;
   int jumper2: 2;
   int jumper3: 3;
} jumpers;
```

Aquí, `jumper1`, `jumper2` y `jumper3` son los tres campos de bits cuyas longitudes son 1, 2 y 3 bits, respectivamente. La figura 20.3 muestra las ubicaciones de memoria de los tres campos de bits.



GURA 20.3
a ubicación de emoria de `jumper1`, mper2 y `jumper3` en a `int` de 16 bits.

Materiał chroniony prawem autorskim

El programa del listado 20.6 es un ejemplo del uso de campos de bits definidos con `struct`. De hecho, este programa es una versión modificada del programa del listado 20.5.

TECLEE **LISTADO 20.6** Aplicación de campos de bits

```
1:  /* 20L06.c: Aplicación de campos de bits */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  struct campo_bits {
6:     int cable: 1;
7:     int antena: 1;
8:  };
9:
10: struct encuesta {
11:    char nombre[20];
12:    struct campo_bits c_a;
13:    int edad;
14:    int horas_por_semana;
15:    union {
16:        char servicio_cable[16];
17:        char servicio_antena[16];
18:    } proveedor;
19: };
20:
21: void CapturaDatos(struct encuesta *s);
22: void ExhibeDatos(struct encuesta *s);
23:
24: main(void)
25: {
26:    struct encuesta tv;
27:
28:    CapturaDatos(&tv);
29:    ExhibeDatos(&tv);
30:
31:    return 0;
32: }
33: /* definición de función */
34: void CapturaDatos(struct encuesta *aptr)
35: {
36:    char respuesta[3];
37:
38:    printf("¿Utiliza TV por cable en casa? (Sí o No)\n");
39:       gets(respuesta);
40:    if ((respuesta[0] == 'S') ||
41:        (respuesta[0] == 's')){
42:       printf("Escriba el nombre de su compañía de cable:\n");
43:       gets(aptr->proveedor.servicio_cable);
```

```
44:         aptr->c_a.cable = 1;
45:         aptr->c_a.antena = 0;
46:     } else {
47:         printf("¿Usa una antena parabólica? (Sí o No)\n");
48:            gets(respuesta);
49:         if ((respuesta[0] == 'S') ||
50:             (respuesta[0] == 's')){
51:            printf("Escriba el nombre de su compañía de antena parabólica:\n");
52:            gets(aptr->proveedor.servicio_antena);
53:            aptr->c_a.cable = 0;
54:            aptr->c_a.antena = 1;
55:         } else {
56:            aptr->c_a.cable = 0;
57:            aptr->c_a.antena = 0;
58:         }
59:     }
60:     printf("Escriba su nombre:\n");
61:         gets(aptr->nombre);
62:     printf("Su edad:\n");
63:         scanf("%d", &aptr->edad);
64:     printf("¿Cuántas horas dedica a la semana a ver TV?:\n");
65:         scanf("%d", &aptr->horas_por_semana);
66: }
67: /* definición de función */
68: void ExhibeDatos(struct encuesta *aptr)
69: {
70:     printf("\nÉstos son los datos que usted introdujo:\n");
71:     printf("Nombre: %s\n", aptr->nombre);
72:     printf("Edad:    %d\n", aptr->edad);
73:     printf("Horas por semana: %d\n", aptr->horas_por_semana);
74:     if (aptr->c_a.cable && !aptr->c_a.antena)
75:         printf("Su compañía de cable es: %s\n",
76:            aptr->proveedor.servicio_cable);
77:     else if (!aptr->c_a.cable && aptr->c_a.antena)
78:         printf("Su compañía de antena parabólica es: %s\n",
79:            aptr->proveedor.servicio_antena);
80:     else
81:         printf("Usted no tiene servicio de cable ni de antena parabólica.\n");
82:     printf("\n¡Gracias y adiós!\n");
83: }
```

Debido a que el programa del listado 20.6 es básicamente igual al del listado 20.5, obtuve los mismos resultados después de ejecutar el programa `20L06.exe` e introducir las mismas respuestas a la encuesta:

SALIDA

```
¿Utiliza TV por cable en casa? (Si o No)
No
¿Usa una antena parabólica? (Sí o No)
```

Materiał chroniony prawem autorskim

P ¿ Puede acceder a la misma ubicación de memoria con diferentes miembros de la unión?

R Sí. Debido a que todos los miembros de una unión comparten la misma ubicación de memoria, es posible acceder a dicha ubicación con diferentes miembros de la unión. Por ejemplo, en el programa del listado 20.4 se asignan dos constantes de carácter al almacenamiento en memoria de una unión a través de uno de sus miembros, y después los dos caracteres se guardan en la ubicación de memoria de la unión y se imprimen con la ayuda de otro miembro de la unión.

Taller

Para ayudarle a consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Cuál de las dos instrucciones siguientes es la declaración de una unión y cuál es la definición de variables de unión?

```
union una_union {
      int x;
      char y;
      };

union una_union x, y;
```

2. ¿Qué está mal en la siguiente declaración de unión?

```
union auto {
      int anio;
      char modelo[8]
      float peso;
      }
```

3. En la siguiente instrucción, ¿cuáles son los valores que contienen los dos miembros de la unión?

```
union u {
      int fecha;
      char anio;
} una_union = {1997};
```

Ejercicios

1. Reescriba el programa del listado 20.1, pero esta vez cambie el orden de las instrucciones de las líneas 15 y 17. ¿Qué es lo que obtiene después de ejecutar el programa modificado? ¿Por qué?
2. Reescriba el programa del listado 20.2. Esta vez imprima los valores contenidos por los miembro de la unión `info` cada vez que se asigne un valor a uno de los miembros.
3. Escriba un programa que solicite al usuario que introduzca su nombre. Luego pregunte al usuario si es ciudadano(a) de México. Si la respuesta es `Sí`, pídale que escriba el estado de donde es originario. En caso contrario, pídale que escriba el nombre del país de donde proviene. (Debe usar una unión en su programa.)
4. Modifique el programa que escribió en el ejercicio 3. Agregue un campo de bits y úselo como bandera. Si el usuario es ciudadano(a) de México, asigne `1` al campo de bits. Si no lo es, asígnele `0`. Imprima el nombre del usuario y el nombre del estado o país revisando el valor del campo de bits.

SINTAXIS

La sintaxis de la función `fclose()` es

```
#include <stdio.h>
int fclose(FILE *flujo);
```

Aquí, *`flujo`* es un apuntador de archivo que está asociado con un flujo al archivo abierto. Si `fclose()` cierra con éxito un archivo, devuelve `0`. En caso contrario, la función devuelve `EOF`. Por lo regular, la función `fclose()` sólo falla cuando se retira el disco antes de llamar a la función o cuando ya no queda espacio en el disco.

Debido a que todas las operaciones de E/S de alto nivel emplean búferes, la función `fclose()` desaloja los datos que quedan en el búfer para asegurarse de que no se pierda ningún dato antes de disociar un flujo específico del archivo abierto.

Observe que, después de la operación de E/S, se tiene que cerrar el archivo que esté abierto y asociado con un flujo . En caso contrario, podrían perderse los datos guardados en el archivo; podrían ocurrir algunos errores impredecibles durante la ejecución de su programa. Además, no cerrar un archivo cuando ya no se usa podría evitar que otros programas (o usuarios de su programa) tengan acceso al archivo más adelante.

El programa del listado 21.1 le muestra cómo abrir y cerrar un archivo de texto, y cómo verificar el valor del apuntador de archivo que devuelve `fopen()`.

TECLEE **LISTADO 21.1** Cómo abrir y cerrar un archivo de texto

```
1:  /* 21L01.c: Cómo abrir y cerrar un archivo */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO};
5:
6:  main(void)
7:  {
8:     FILE *aptrf;
9:     char nomarchivo[]= "haiku.txt";
10:    int valdevol = EXITO;
11:
12:    if ((aptrf = fopen(nomarchivo, "r")) == NULL){
13:       printf("No es posible abrir %s.\n", nomarchivo);
14:       valdevol = FRACASO;
15:    } else {
16:       printf("El valor de aptrf es: 0x%p\n", aptrf);
17:       printf("Listo para cerrar el archivo.");
18:       fclose(aptrf);
19:    }
20:
21:    return valdevol;
22: }
```

Materiał chroniony prawem autorskim

Después de ejecutar en mi computadora el archivo `21L01.exe` del programa del listado 21.1, se desplegaron en la pantalla los siguientes resultados (es probable que el valor de `aptrf` sea diferente en su máquina. No hay problema):

SALIDA

```
El valor de aptrf es: 0x013E
Listo para cerrar el archivo.
```

ANÁLISIS La finalidad del programa del listado 21.1 es mostrarle cómo abrir un archivo de texto. Puede ver en la expresión de la línea 12 que la función `fopen()` intenta abrir un archivo de texto para lectura con el nombre contenido en el arreglo de caracteres `nomarchivo`. El arreglo `nomarchivo` se define y se inicializa con el nombre `haiku.txt` en la línea 9.

La función `fopen()` devuelve un apuntador nulo si ocurre un error al intentar abrir el archivo de texto. La línea 13 imprime un mensaje de advertencia y la línea 14 le asigna a la variable `valdevol` de tipo `int` el valor representado por el nombre `enum FRACASO`. Debido a la declaración del tipo de datos `enum` de la línea 4, usted sabe que el valor de `FRACASO es 1`.

Sin embargo, si la función `fopen()` abre el archivo de texto con éxito, la instrucción de la línea 16 imprime el valor contenido por el apuntador de archivo `aptrf`. La línea 17 le indica al usuario que el programa está por cerrar el archivo, y la línea 18 lo cierra por medio de la llamada a la función `fclose()`.

La instrucción `return` de la línea 21 devuelve el valor de `valdevol`, que contiene `0` si se abrió con éxito el archivo, o `1` en caso contrario.

Puede observar en los resultados que aparecen en mi pantalla que, después de abrir el archivo de texto, el valor que contiene el apuntador de archivo `aptrf` es `0x013E`.

Lectura y escritura de archivos en disco

El programa del listado 21.1 no hace nada con el archivo de texto `haiku.txt`, excepto abrirlo y cerrarlo. De hecho, en ese archivo están almacenados dos haikus japoneses, escritos por Sodo y Chora. Pero, ¿cómo puede leerlos del archivo?

En C, puede realizar operaciones de E/S en las siguientes formas:

- Leer o escribir un carácter a la vez.
- Leer o escribir una línea de texto (es decir, una línea de caracteres) a la vez.
- Leer o escribir un bloque de caracteres a la vez.

Las tres secciones siguientes explican las tres formas de leer archivos en disco y escribir en ellos.

Un carácter a la vez

Entre las funciones de E/S de C, hay un par de ellas, `fgetc()` y `fputc()`, que se pueden usar para leer o escribir un carácter a la vez en un archivo en disco.

SINTAXIS

La sintaxis de la función `fgetc()` es

```
#include <stdio.h>
int fgetc(FILE *flujo);
```

Aquí, *flujo* es el apuntador de archivo asociado a un flujo. La función `fgetc()` extrae el siguiente carácter del flujo especificado por *flujo*. La función devuelve entonces un valor de tipo `int` que se convierte a partir del carácter. Si `fgetc()` encuentra el final del archivo, o si ocurre un error, devuelve `EOF`.

SINTAXIS

La sintaxis de la función `fputc()` es

```
#include <stdio.h>
int fputc(int c, FILE *flujo);
```

Aquí, `c` es un valor de tipo `int` que representa un carácter. De hecho, el valor `int` se convierte al tipo `unsigned char` antes de imprimirlo. *flujo* es el apuntador de archivo asociado a un flujo. Si tiene éxito, la función `fputc()` devuelve el carácter impreso; en caso contrario, devuelve `EOF`. Después de escribir un carácter, la función `fputc()` coloca adelante el apuntador de archivo asociado.

Para aprender cómo utilizar las funciones `fgetc()` y `fputc()`, veamos el listado 21.2, el cual contiene un programa que abre un archivo de texto y después lee e imprime un carácter a la vez.

TECLEE **LISTADO 21.2** Cómo leer y escribir un carácter a la vez

```
1:  /* 21L02.c: Cómo leer y escribir un carácter a la vez */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO};
5:
6:  void LeeEscribeCar(FILE *archEnt, FILE *archSal);
7:
8:  main(void)
9:  {
10:    FILE *aptrf1, *aptrf2;
11:    char nomarchivo1[]= "haikusal.txt";
12:    char nomarchivo2[]= "haiku.txt";
13:    int valdevol = EXITO;
14:
15:    if ((aptrf1 = fopen(nomarchivo1, "w")) == NULL){
16:       printf("No es posible abrir %s.\n", nomarchivo1);
17:       valdevol = FRACASO;
18:    } else if ((aptrf2 = fopen(nomarchivo2, "r")) == NULL){
19:       printf("No es posible abrir %s.\n", nomarchivo2);
```

continúa

Materiał chroniony prawem autorskim

LISTADO 21.2 continuación

```
20:         valdevol = FRACASO;
21:     } else {
22:         LeeEscribeCar(aptrf2, aptrf1);
23:         fclose(aptrf1);
24:         fclose(aptrf2);
25:     }
26:
27:     return valdevol;
28: }
29: /* definición de función */
30: void LeeEscribeCar(FILE *archEnt, FILE *archSal)
31: {
32:     int c;
33:
34:     while ((c=fgetc(archEnt)) != EOF){
35:         fputc(c, archSal);  /* escribe en un archivo */
36:         putchar(c);       /* coloca el carácter en la pantalla */
37:     }
38: }
```

Después de ejecutar en mi máquina el archivo `21L02.exe`, obtuve los siguientes resultados:

SALIDA

```
Llevándome con ella
mi sombra regresa a casa
tras mirar la luna.
--- Sodo
    (1641-1716)

Sopla un viento de tormenta
de entre las hierbas
crece la luna llena.
--- Chora
    (1729-1781)
```

ANÁLISIS La finalidad del programa del listado 21.2 es leer un carácter de un archivo, escribirlo en otro archivo y desplegarlo en la pantalla. Antes de ejecutar el programa, necesita crear un archivo de texto denominado `haiku.txt`, con el contenido que se muestra arriba, y guardarlo en el mismo directorio del programa ejecutable.

En el listado 21.2 hay una función llamada `LeeEscribeCar()`, la cual tiene dos apuntadores como argumentos (vea la declaración de la función en la línea 6).

La instrucción de la línea 10 define dos apuntadores de archivo, `aptrf1` y `aptrf2`, los cuales se usan más adelante en el programa. Las líneas 11 y 12 definen dos arreglos de caracteres, `nomarchivo1` y `nomarchivo2`, y los inicializan con dos cadenas que contienen nombres de archivo, `haikusal.txt` y `haiku.txt`.

En la línea 15 se abre para escritura el archivo llamado `haikusal.txt`. `haikusal.txt` es la cadena contenida por el arreglo `nomarchivo1`. El apuntador `aptrf1` se asocia al archivo. Si la función `fopen()` devuelve `NULL`, lo que significa que ocurrió un error, en la línea 16 se imprime un mensaje de advertencia. Además, en la línea 17 se le asigna `1` a la variable `valdevol` y se representa por medio del nombre `enum FRACASO`.

Si se abre con éxito el archivo `haikusal.txt`, en la línea 18 se abre otro archivo de texto para lectura llamado `haiku.txt`. El apuntador de archivo `aptrf2` se asocia al archivo de texto abierto.

Si no ocurre ningún error, en la línea 22 se llama a la función `LeeEscribeCar()` y se le pasan dos apuntadores de archivo como argumentos, `aptrf1` y `aptrf2`. Puede ver en la definición de la función `LeeEscribeCar()`, en las líneas 30 a 38, que hay un ciclo `while` que continúa llamando a la función `fgetc()` para leer el siguiente carácter del archivo de texto `haiku.txt`, hasta que la función llega al final del archivo (vea la línea 34).

Dentro del ciclo `while`, la función `fputc()` de la línea 35 escribe en el archivo de texto `haikusal.txt`, al cual apunta `archSal`, cada carácter leído del archivo `haiku.txt`. Además, en la línea 36 se llama a `putchar()` para desplegar en la pantalla el carácter devuelto por la función `fgetc()`.

Después de que la función `LeeEscribeCar()` termina su labor, los dos archivos abiertos, que están asociados a `aptrf1` y `aptrf2`, se cierran con una llamada a la función `fclose()` de las líneas 23 y 24, respectivamente.

Como se mencionó antes, el archivo `haiku.txt` contiene dos haikus japoneses escritos por Sodo y Chora. Si el programa del listado 21.2 se ejecuta con éxito, usted verá los dos haikus en la pantalla, y también se escribirán en el archivo `haikusal.txt`. Para confirmar que el contenido de `haiku.txt` se haya copiado correctamente a `haikusal.txt`, puede ver este último archivo en un editor de texto.

Una línea a la vez

Además de leer o escribir un carácter a la vez, también puede leer o escribir una línea de caracteres a la vez. En C hay un par de funciones de E/S, `fgets()` y `fputs()`, que le permiten hacerlo.

SINTAXIS

La sintaxis de la función `fgets()` es

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *flujo);
```

Aquí, *s* hace referencia a un arreglo de caracteres que se usa para almacenar los caracteres leídos del archivo abierto al que apunta el apuntador de archivo *flujo*. *n* especifica el número máximo de elementos del arreglo. Si tiene éxito, la función `fgets()` devuelve

Materiał chroniony prawem autorskim

Mientras tanto, cada línea leída por la función `fgets()` se escribe en otro archivo de texto abierto llamado `haikusal.txt` que está asociado al apuntador de archivo `archSal`. Esto se hace llamando a la función `fputs()` de la línea 35.

La instrucción de la línea 36 imprime en la pantalla el contenido de cada cadena para que usted pueda ver los dos versos japoneses después de ejecutar el programa del listado 21.3. También puede ver el archivo de texto `haikusal.txt` en un editor de texto para asegurarse de que el contenido del archivo `haiku.txt` se haya copiado al archivo `haikusal.txt`.

En la hora anterior utilizamos la función `gets()` para leer entrada de datos desde el teclado. Dado que `gets()` no conoce el tamaño del arreglo de caracteres que se le pasa, simplemente lee datos hasta encontrar un carácter de línea nueva. Esto es en realidad muy peligroso, debido a que el usuario podría con facilidad teclear más caracteres de los que puede contener el arreglo. Como resultado, podrían sobreescribirse las otras variables y el programa sufriría un colapso o, en el mejor de los casos, se comportaría de manera impredecible.

Por fortuna, `fgets()` proporciona una forma mucho más segura para leer entradas desde el teclado, si le pasa `stdin` como el flujo de entrada para leer.

El siguiente código usa `fgets()` para leer entradas desde `stdin` dentro de una cadena `cad`.

```
int longitud = 80;
char cad[80];
fgets(cad, longitud, stdin);
if (cad[(longitud = strlen(cad) - 1)] == '\n')
  cad[longitud] = '\0';  /* reemplaza el carácter '\n' */
else
  while (getchar() != '\n')
    ;  /* descarta la entrada hasta el carácter de línea nueva */
```

Como puede ver, este método implica un poco más de trabajo que `gets()`, ya que tenemos que deshacernos del carácter de línea nueva que almacena `fgets()` en nuestro arreglo. Sin embargo, esto bien vale la pena pues es una forma mucho más segura y limpia de trabajar con entradas del usuario que utilizando `gets()`.

Un bloque a la vez

Si lo desea, también puede leer o escribir un bloque de datos a la vez. En C hay dos funciones de E/S, `fread()` y `fwrite()`, que se pueden utilizar para realizar operaciones de E/S en bloque. Las funciones `fread()` y `fwrite()` se reflejan entre sí.

SINTAXIS

La sintaxis de la función `fread()` es

```
#include <stdio.h>
size_t fread(void *aptr, size_t tamaño, size_t n, FILE *flujo);
```

Aquí, *aptr* es un apuntador a un arreglo en el que se almacenan los datos. *tamaño* indica el tamaño de cada elemento del arreglo. *n* especifica el número de elementos a leer.

Materiał chroniony prawem autorskim

El programa del listado 21.4 muestra cómo leer y escribir un bloque de caracteres a la vez llamando a las funciones `fread()` y `fwrite()`. De hecho, el programa del listado 21.4 es otra versión modificada del programa del listado 21.2.

TECLEE **LISTADO 21.4** Cómo leer y escribir un bloque de caracteres a la vez

```
1:  /* 21L04.c: Cómo leer y escribir un bloque a la vez */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO, MAX_LEN = 80};
5:
6:  void LeeEscribeBloque(FILE *archEnt, FILE *archSal);
7:  int MsjError(char *cadena);
8:
9:  main(void)
10: {
11:    FILE *aptrf1, *aptrf2;
12:    char nomarchivo1[]= "haikusal.txt";
13:    char nomarchivo2[]= "haiku.txt";
14:    int valdevol = EXITO;
15:
16:    if ((aptrf1 = fopen(nomarchivo1, "w")) == NULL){
17:       valdevol = MsjError(nomarchivo1);
18:    } else if ((aptrf2 = fopen(nomarchivo2, "r")) == NULL){
19:       valdevol = MsjError(nomarchivo2);
20:    } else {
21:       LeeEscribeBloque(aptrf2, aptrf1);
22:       fclose(aptrf1);
23:       fclose(aptrf2);
24:    }
25:
26:    return valdevol;
27: }
28: /* definición de función */
29: void LeeEscribeBloque(FILE *archEnt, FILE *archSal)
30: {
31:    int num;
32:    char bufer[MAX_LEN + 1];
33:
34:    while (!feof(archEnt)){
35:       num = fread(bufer, sizeof(char), MAX_LEN, archEnt);
36:       bufer[num * sizeof(char)] = '\0';  /* agrega un carácter nulo */
37:       printf("%s", bufer);
38:       fwrite(bufer, sizeof(char), num, archSal);
39:    }
40: }
41: /* definición de función */
42: int MsjError(char *cadena)
43: {
```

Resumen

En esta lección aprendió los siguientes conceptos y funciones importantes en C con respecto a la entrada y salida de archivos en disco:

- En C, un archivo puede hacer referencia a un archivo en disco, una terminal, una impresora o una unidad de cinta.
- Al flujo de datos que transfiere de su programa a un archivo, o viceversa, se le denomina flujo.
- Un flujo es una serie de bytes ordenados.
- A diferencia de un archivo, un flujo es independiente del dispositivo.
- Hay dos formatos de flujo: flujo de texto y flujo binario.
- El indicador de posición del archivo de la estructura `FILE` apunta a la posición de un archivo en donde a continuación se leerán o escribirán datos.
- La función `fopen()` se usa para abrir un archivo y asociarle un flujo.
- Usted puede especificar diferentes modos de abrir un archivo.
- La función `fclose()` es responsable de cerrar un archivo abierto y disociarlo de un flujo.
- Las funciones `fgetc()` y `fputc()` leen o escriben un carácter a la vez.
- Las funciones `fgets()` y `fputs()` leen o escriben una línea a la vez.
- Las funciones `fread()` y `fwrite()` leen o escriben un bloque de datos a la vez.
- La función `feof()` puede determinar cuándo se llegó al final de un archivo.
- Para detectar el EOF en un archivo binario, se debe usar la función `feof()`.

En la siguiente lección aprenderá más acerca de las operaciones de E/S de archivos en disco, en C.

Preguntas y respuestas

P ¿Cuáles son las diferencias entre un flujo de texto y un flujo binario?

R Un flujo de texto es una secuencia de caracteres que podría no tener una relación uno a uno con los datos del dispositivo. Los flujos de texto se usan normalmente para datos textuales, los cuales tienen una apariencia consistente de un ambiente a otro, o de una máquina a otra. Por esta razón, los datos de un archivo de texto se pueden interpretar de modo que aparezcan correctamente en la pantalla. Por otra parte, un flujo binario es una secuencia de bytes que tiene una correspondencia uno a uno con los bytes del dispositivo. Los flujos binarios se emplean principalmente para datos no textuales que son necesarios para mantener el contenido exacto del dispositivo.

P ¿Por qué es necesario un apuntador de archivo?

R Un apuntador de archivo se utiliza para asociar un flujo con un archivo abierto para fines de lectura o escritura. A un apuntador de tipo `FILE` se le llama apuntador

Material chroniony prawem autorskim

de archivo. `FILE` es un `typedef` de una estructura que contiene información de sobrecarga acerca de un archivo en disco. Un apuntador de archivo juega un papel muy importante en la comunicación entre programas y archivos en disco.

P ¿Qué hace la función `fclose()` antes de cerrar un archivo abierto?

R Como usted sabe, todas las operaciones de E/S de alto nivel utilizan búferes. Una de la tareas de la función `fclose()` es desalojar los datos que quedan en el búfer para confirmar las operaciones de E/S y asegurar que no se pierdan datos. Por ejemplo, al terminar de escribir varios bloques de caracteres en un archivo de texto abierto, usted llama a la función `fclose()` para disociar un flujo especificado y cerrar el archivo de texto. La función `fclose()` primero desalojará todos los caracteres que queden en el búfer y los escribirá en el archivo de texto antes de cerrarlo. En esta forma, todos los caracteres que escriba en el archivo se guardarán en forma adecuada.

P ¿Cuál es la diferencia entre `fgets()` y `gets()`?

R La principal diferencia entre las funciones `fgets()` y `gets()` es que la función `fgets()` incluye un carácter de línea nueva en el arreglo si durante la operación de lectura se encuentra la línea nueva, mientras que la función `gets()` sólo reemplaza el carácter de línea nueva con un carácter nulo. Además, `fgets()` es mucho más seguro que `gets()`, pues le permite especificar el tamaño del arreglo en el que se almacena la cadena.

Taller

Para ayudarle a consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Qué hace la siguiente expresión?

```
fopen("prueba.bin", "r+b")
fopen("prueba.txt" "a")
fopen("prueba.ini", "w+")
```

2. ¿Qué está mal en el siguiente segmento de código?

```
FILE *aptrf;
int c;
if ((aptrf = fopen("prueba1.txt", "r")) == NULL){
   while ((c=fgetc(aptrf)) != EOF){
      putchar(c);
   }
}
fclose(aptrf);
```

HORA 22

Funciones de archivo especiales

Espacio en disco: la última frontera.

—El hermano menor del capitán Kirk

En la lección anterior aprendió los aspectos básicos de lectura y escritura de archivos de datos en disco. En esta lección aprenderá más acerca de la comunicación con los archivos de datos en disco. Los principales temas que trataremos en esta hora son los siguientes:

- Archivos de acceso aleatorio
- Lectura y escritura de datos binarios
- Redireccionamiento de los flujos estándar

Además, en esta lección se presentan las siguientes funciones de E/S de C:

- Las funciones `fseek()`, `ftell()` y `rewind()`
- Las funciones `fscanf()` y `fprintf()`
- La función `freopen()`

Materiał chroniony prawem autorskim

Acceso aleatorio a archivos en disco

Hasta ahora ha aprendido a leer o escribir datos en forma secuencial en un archivo de disco abierto, lo que se conoce como *acceso secuencial*. En otras palabras, comenzó por el primer byte y siguió leyendo o escribiendo en orden cada byte sucesivo. Sin embargo, en muchos casos necesita acceder a datos específicos que se encuentran a la mitad del archivo. Una forma de hacerlo es seguir leyendo datos del archivo hasta extraer los datosespecíficos. Obviamente, ésta no es una forma eficiente, en especial cuando el archivo contiene muchos elementos de datos.

El *acceso aleatorio* es otra forma de leer o escribir datos en archivos en disco. En el acceso aleatorio es posible acceder a elementos específicos del archivo en un orden aleatorio (es decir, sin leer todos los datos precedentes).

En C hay dos funciones de E/S, `fseek()` y `ftell()`, que están diseñadas para llevar a cabo el acceso aleatorio.

Las funciones `fseek()` y `ftell()`

Como acabo de mencionar, usted necesita funciones que le permitan acceder a archivos en forma aleatoria. Las funciones `fseek()` y `ftell()` le proporcionan esta capacidad.

En la lección anterior aprendió que uno de los miembros de la estructura `FILE` se denomina *indicador de posición del archivo*. Antes de poder leer o escribir en un archivo, el indicador de posición del archivo debe apuntar a la posición deseada en el mismo. Puede emplear la función `fseek()` para mover el indicador de posición del archivo al punto en el que desea acceder al archivo.

SINTAXIS

La sintaxis de la función `fseek()` es

```
#include <stdio.h>
int fseek(FILE *flujo, long desplazamiento, int desde);
```

Aquí, *`flujo`* es el apuntador de archivo asociado a un archivo abierto. *`desplazamiento`* indica el número de bytes desde una posición fija, especificada por *`desde`*, que puede tener uno de los siguientes valores integrales representados por `SEEK_SET`, `SEEK_CUR` y `SEEK_END`. Si tiene éxito, la función `fseek()` devuelve `0`; en caso contrario, devuelve un valor diferente de cero.

Puede encontrar en el archivo de encabezado `stdio.h` los valores representados por `SEEK_SET`, `SEEK_CUR` y `SEEK_END`.

Si se elige `SEEK_SET` como el tercer argumento para la función `fseek()`, el desplazamiento se cuenta a partir del principio del archivo, y el valor del desplazamiento debe ser mayor o igual a cero. Sin embargo, si utiliza `SEEK_END`, el desplazamiento inicia a partir del final del archivo, y el valor del desplazamiento debe ser negativo. Cuando se pasa `SEEK_CUR` a

la función `fseek()`, el desplazamiento se calcula desde el valor actual del indicador de posición del archivo.

SINTAXIS

Puede obtener el valor actual del indicador de posición del archivo por medio de la llamada a la función `ftell()`.

La sintaxis de la función `ftell()` es

```
#include <stdio.h>
long ftell(FILE *flujo);
```

Aquí, *`flujo`* es el apuntador de archivo asociado a un archivo abierto. La función `ftell()` devuelve el valor actual del indicador de posición del archivo.

El valor que regresa la función `ftell()` representa el número de bytes desde el principio del archivo hasta la posición actual a la que apunta el indicador de posición del archivo.

Si falla, la función `ftell()` devuelve `-1L` (es decir, un valor `long` de menos `1`). Algo que puede hacer que la función `ftell()` falle, es que el archivo sea una terminal o algún otro tipo para el cual el indicador de posición del archivo no sea significativo.

El programa del listado 22.1 muestra cómo acceder de manera aleatoria a un archivo en disco por medio de las funciones `fseek()` y `ftell()`.

TECLEE **LISTADO 22.1** Acceso aleatorio a un archivo

```
1:  /* 22L01.c: Acceso aleatorio a un archivo */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO, MAX_LEN = 80};
5:
6:  void AptrSeek(FILE *aptrf);
7:  long AptrTell(FILE *aptrf);
8:  void LeeDatos(FILE *aptrf);
9:  int MsjError(char *cad);
10:
11: main(void)
12: {
13:     FILE *aptrf;
14:     char nomarchivo[]= "haiku.txt";
15:     int valdevol = EXITO;
16:
17:     if ((aptrf = fopen(nomarchivo, "r")) == NULL){
18:         valdevol = MsjError(nomarchivo);
19:     } else {
20:         AptrSeek(aptrf);
21:         fclose(aptrf);
22:     }
23:
```

continúa

Materiał chroniony prawem autorskim

LISTADO 22.1 continuación

```
24:     return valdevol;
25: }
26: /* definición de función */
27: void AptrSeek(FILE *aptrf)
28: {
29:     long desplaz1, desplaz2, desplaz3;
30:
31:     desplaz1 = AptrTell(aptrf);
32:     LeeDatos(aptrf);
33:     desplaz2 = AptrTell(aptrf);
34:     LeeDatos(aptrf);
35:     desplaz3 = AptrTell(aptrf);
36:     LeeDatos(aptrf);
37:
38:     printf("\nRelee el haiku:\n");
39:     /* relee el tercer verso del haiku */
40:     fseek(aptrf, desplaz3, SEEK_SET);
41:     LeeDatos(aptrf);
42:     /* relee el segundo verso del haiku */
43:     fseek(aptrf, desplaz2, SEEK_SET);
44:     LeeDatos(aptrf);
45:     /* relee el primer verso del haiku */
46:     fseek(aptrf, desplaz1, SEEK_SET);
47:     LeeDatos(aptrf);
48: }
49: /* definición de función */
50: long AptrTell(FILE *aptrf)
51: {
52:     long valdevol;
53:
54:     valdevol = ftell(aptrf);
55:     printf("El apuntador aptrf está en %ld\n", valdevol);
56:
57:     return valdevol;
58: }
59: /* definición de función */
60: void LeeDatos(FILE *aptrf)
61: {
62:     char buff[MAX_LEN];
63:
64:     fgets(buff, MAX_LEN, aptrf);
65:     printf("---%s", buff);
66: }
67: /* definición de función */
68: int MsjError(char *cad)
69: {
70:     printf("No es posible abrir %s.\n", cad);
71:     return FRACASO;
72: }
```

Después de ejecutar el archivo `22L01.exe` del programa del listado 22.1, se mostraron en la pantalla de mi computadora los siguientes resultados:

SALIDA

```
El apuntador aptrf está en 0
---Llevándome con ella
El apuntador aptrf está en 21
---mi sombra regresa a casa
El apuntador aptrf está en 46
---tras mirar la luna.

Relee el haiku:
---tras mirar la luna.
---mi sombra regresa a casa
---Llevándome con ella
```

ANÁLISIS La finalidad del programa del listado 22.1 es mover el indicador de posición para leer diferentes versos del archivo `haiku.txt`.

Dentro de la función `main()`, en la línea 13 se define el apuntador de archivo `aptrf`, y en la línea 14 se asigna el nombre del archivo `haiku.txt` al arreglo llamado `nomarchivo`. Luego, en la línea 17, se intenta abrir el archivo `haiku.txt` para lectura por medio de la llamada a la función `fopen()`. Si tiene éxito, en la línea 20 se invoca a la función `AptrSeek()` con el apuntador de archivo `aptrf` como argumento.

La definición de la primera función, `AptrSeek()`, se muestra en las líneas 27 a 48. La instrucción de la línea 31 obtiene el valor original del apuntador de archivo `aptrf` por medio de la llamada a la otra función, `AptrTell()`, la cual se define en las líneas 50 a 58. La función `AptrTell()` encuentra e imprime el valor del indicador de posición del archivo con ayuda de la función `ftell()`. En la línea 31, el valor original del indicador de posición del archivo contenido por `aptrf` se asigna a la variable `desplaz1` de tipo `long`.

En la línea 32 se llama a la tercera función, `LeeDatos()`, para leer una línea de caracteres del archivo abierto e imprimirla en la pantalla. La línea 33 obtiene el valor del indicador de posición del archivo, `aptrf`, justo después de la lectura, y asigna el valor a otra variable `long`, `desplaz2`.

Después, la función `LeeDatos()` de la línea 34 lee la segunda línea de caracteres del archivo abierto. La línea 35 obtiene el valor del indicador de posición del archivo que apunta al primer byte del tercer verso y asigna el valor a la tercera variable `long`, `desplaz3`. La línea 36 llama a la función `LeeDatos()` para leer el tercer verso e imprimirlo en la pantalla.

Por lo tanto, en la primera parte de los resultados puede ver los tres diferentes valores del indicador de posición del archivo, así como los tres versos del haiku escrito por Sodo, en tres posiciones distintas. Los tres valores del indicador de posición del archivo se guardan en `desplaz1`, `desplaz2` y `desplaz3`, respectivamente.

fopen(). Por ejemplo, la siguiente instrucción intenta abrir un archivo binario existente para lectura:

```
aptrf = fopen("prueba.bin", "rb");
```

Observe que se usa el modo "rb" para indicar que el archivo que va a abrir para lectura es un archivo binario.

El listado 22.2 contiene un ejemplo de lectura y escritura de datos binarios.

TECLEE **LISTADO 22.2** Lectura y escritura de datos binarios

```
1:  /* 22L02.c: Lectura y escritura de datos binarios */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO, MAX_NUM = 3};
5:
6:  void EscribeDatos(FILE *archSal);
7:  void LeeDatos(FILE *archEnt);
8:  int MsjError(char *cadena);
9:
10: main(void)
11: {
12:    FILE *aptrf;
13:    char nomarchivo[]= "doble.bin";
14:    int valdevol = EXITO;
15:
16:    if ((aptrf = fopen(nomarchivo, "wb+")) == NULL){
17:       valdevol = MsjError(nomarchivo);
18:    } else {
19:       EscribeDatos(aptrf);
20:       rewind(aptrf);  /* restablece aptrf */
21:       LeeDatos(aptrf);
22:       fclose(aptrf);
23:    }
24:
25:    return valdevol;
26: }
27: /* definición de función */
28: void EscribeDatos(FILE *archSal)
29: {
30:    int i;
31:    double bufer[MAX_NUM] = {
32:                123.45,
33:                567.89,
34:                100.11};
35:
```

continúa

Materiał chroniony prawem autorskim

SINTAXIS

La sintaxis de la función `fprintf()` es

```
#include <stdio.h>
int fprintf(FILE *flujo, const char *formato, ...);
```

Aquí, *`flujo`* es el apuntador de archivo asociado a un archivo abierto. *`formato`*, cuyo uso es el mismo que en la función `printf()`, es un apuntador de tipo `char` que apunta a una cadena que contiene los especificadores de formato. Si tiene éxito, la función `fprintf()` devuelve el número de expresiones formateadas. En caso contrario, devuelve un valor negativo.

Para conocer más acerca de las funciones `fprintf()` y `fscanf()`, puede revisar las explicaciones sobre las funciones `printf()` y `scanf()` de la hora 5, "Manejo de la entrada y salida estándar", y de la hora 13, "Cadenas".

El programa del listado 22.3 muestra cómo utilizar las funciones `fscanf()` y `fprintf()` para leer y escribir elementos de datos de diferentes tipos.

TECLEE **LISTADO 22.3** Uso de las funciones `fscanf()` y `fprintf()`

```
1:  /* 22L03.c: Uso de las funciones fscanf() y fprintf() */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO,
5:        MAX_NUM = 3,
6:        LONG_CAD = 23};
7:
8:  void EscribeDatos(FILE *archSal);
9:  void LeeDatos(FILE *archEnt);
10: int MsjError(char *cadena);
11:
12: main(void)
13: {
14:    FILE *aptrf;
15:    char nomarchivo[]= "numycad.mix";
16:    int valdevol = EXITO;
17:
18:    if ((aptrf = fopen(nomarchivo, "w+")) == NULL){
19:       valdevol = MsjError(nomarchivo);
20:    } else {
21:       EscribeDatos(aptrf);
22:       rewind(aptrf);
23:       LeeDatos(aptrf);
24:       fclose(aptrf);
25:    }
26:
27:    return valdevol;
28: }
29: /* definición de función */
30: void EscribeDatos(FILE *archSal)
```

```
31: {
32:    int i;
33:    char ciudades[MAX_NUM][LONG_CAD] = {
34:                "St.Louis->Houston:",
35:                "Houston->Dallas:",
36:                "Dallas->Philadelphia:"};
37:    int millas[MAX_NUM] = {
38:                845,
39:                243,
40:                1459};
41:
42:    printf("Los datos escritos son:\n");
43:    for (i=0; i<MAX_NUM; i++){
44:       printf("%-23s  %d millas\n", ciudades[i], millas[i]);
45:       fprintf(archSal, "%s  %d", ciudades[i], millas[i]);
46:    }
47: }
48: /* definición de función */
49: void LeeDatos(FILE *archEnt)
50: {
51:    int i;
52:    int millas;
53:    char ciudades[LONG_CAD];
54:
55:    printf("\nLos datos leídos son:\n");
56:    for (i=0; i<MAX_NUM; i++){
57:       fscanf(archEnt, "%s%d", ciudades, &millas);
58:       printf("%-23s  %d millas\n", ciudades, millas);
59:    }
60: }
61: /* definición de función */
62: int MsjError(char *cadena)
63: {
64:    printf("No es posible abrir %s.\n", cadena);
65:    return FRACASO;
66: }
```

Después de crear y ejecutar en mi computadora el archivo `22L03.exe`, se mostraron en la pantalla los siguientes resultados:

SALIDA

```
Los datos escritos son:
St.Louis->Houston:      845 millas
Houston->Dallas:        243 millas
Dallas->Philadelphia:  1459 millas

Los datos leídos son:
St.Louis->Houston:      845 millas
Houston->Dallas:        243 millas
Dallas->Philadelphia:  1459 millas
```

Materiał chroniony prawem autorskim

ANÁLISIS La finalidad del programa del listado 22.3 es escribir en un archivo elementos de datos de diferentes tipos con la ayuda de la función `fprintf()`, y leerlos de nuevo en el mismo formato por medio de la llamada a la función `fscanf()`. En realidad, las dos funciones declaradas en las líneas 8 y 9, `EscribeDatos()` y `LeeDatos()`, realizan la escritura y la lectura.

La instrucción de la línea 18 de la función `main()` intenta crear y abrir un archivo de texto llamado `numycad.mix`, para lectura y escritura, especificando el segundo argumento para la función `fopen()` como `"w+"`. Si `fopen()` no devuelve un apuntador nulo, en la línea 21 se llama a la función `EscribeDatos()` para escribir cadenas y elementos de datos de tipo `int` en el archivo `numycad.mix`. Observe que la función `fprintf()` se invoca dentro de la función `EscribeDatos()` de la línea 45, para escribir en el archivo de texto los datos formateados.

Puede ver en la definición de la función `EscribeDatos()`, en las líneas 30 a 47, que hay dos arreglos, `ciudades` y `millas`. El arreglo `ciudades` contiene tres cadenas que indican tres pares de ciudades, y el arreglo `millas` tiene tres valores `int` que representan las distancias correspondientes entre las ciudades que muestra el arreglo `ciudades`. Por ejemplo, en el arreglo `millas`, `845` es la distancia (en millas) entre las dos ciudades expresadas por la cadena `St.Louis->Houston:` del arreglo `ciudades`.

En la línea 22 se llama a la función `rewind()` para restablecer el indicador de posición del archivo al inicio del archivo `numycad.mix`. Luego, la función `LeeDatos()` de la línea 23 lee de nuevo, con la ayuda de la función `fscanf()`, lo que se guardó en `numycad.mix`. La definición de la función `LeeDatos()` se muestra en las líneas 49 a 60.

Puede ver en este ejemplo que es conveniente utilizar juntas las funciones `fprintf()` y `fscanf()` para realizar operaciones de E/S formateadas en archivos en disco.

Redirección de los flujos estándar mediante `freopen()`

En la hora 5 aprendió cómo leer o escribir en la E/S estándar. También se le mencionó que ciertas funciones de C, como `getc()`, `gets()`, `putc` y `printf()`, dirigen en forma automática sus operaciones de E/S a `stdin` o a `stdout`.

En esta sección aprenderá cómo redirigir los flujos estándar, como `stdin` y `stdout`, a archivos en disco. También utilizará una nueva función de C llamada `freopen()`, la cual puede asociar un flujo estándar a un archivo en disco.

SINTAXIS La sintaxis de la función `freopen()` es

```
#include <stdio.h>
FILE *freopen(const char *nomarchivo, const char *modo, FILE *flujo);
```

Aquí, *`nomarchivo`* es un apuntador de tipo `char` que hace referencia al nombre de un archivo que usted necesita asociar al flujo estándar representado por *`flujo`*. *`modo`* es otro

apuntador de tipo `char` que apunta a una cadena que define la forma de abrir un archivo. Los valores que puede tener *modo* en `freopen()` son los mismos que en la función `fopen()`. (En la hora 21 se proporcionaron las definiciones de todos los valores de modo.)

Si ocurre un error, la función `freopen()` devuelve un apuntador nulo. En caso contrario, la función devuelve el flujo estándar que se asoció al archivo en disco identificado por *nomarchivo*.

El listado 22.4 demuestra un ejemplo de redireccionamiento de la salida estándar, `stdout`, con ayuda de la función `freopen()`.

TECLEE **LISTADO 22.4** Redireccionamiento del flujo estándar `stdout`

```
1:  /* 22L04.c: Redireccionamiento de un flujo estándar */
2:  #include <stdio.h>
3:
4:  enum {EXITO, FRACASO,
5:        NUM_CAD = 4};
6:
7:  void ImprimeCadena(char **cad);
8:  int MsjError(char *cad);
9:
10: main(void)
11: {
12:    char *cad[NUM_CAD] = {
13:             "Inclínate y quedarás erguido.",
14:             "Vacíate y quedarás lleno.",
15:             "Desgástate y quedarás renovado.",
16:             "--- por Lao Tzu"};
17:    char nomarchivo[]= "LaoTzu.txt";
18:    int valdevol = EXITO;
19:
20:    ImprimeCadena(cad);
21:    if (freopen(nomarchivo, "w", stdout) == NULL){
22:       valdevol = MsjError(nomarchivo);
23:    } else {
24:       ImprimeCadena(cad);
25:       fclose(stdout);
26:    }
27:    return valdevol;
28: }
29: /* definición de función */
30: void ImprimeCadena(char **cad)
31: {
32:    int i;
33:
```

continúa

Materiał chroniony prawem autorskim

LISTADO 22.4 continuación

```
34:     for (i=0; i<NUM_CAD; i++)
35:        printf("%s\n", cad[i]);
36: }
37: /* definición de función */
38: int MsjError(char *cad)
39: {
40:     printf("No es posible abrir %s.\n", cad);
41:     return FRACASO;
42: }
```

Después de crear y ejecutar el archivo `22L04.exe`, se desplegaron los siguientes resultados en la pantalla de mi computadora:

SALIDA

```
Inclínate y quedarás erguido.
Vacíate y quedarás lleno.
Desgástate y quedarás renovado.
--- por Lao Tzu
```

ANÁLISIS La finalidad del programa del listado 22.4 es guardar en un archivo de texto, `LaoTzu.txt`, un párrafo del *Tao Te Ching* escrito por el filósofo chino Lao Tzu. Para ello, después de redireccionar el flujo predeterminado, `stdout`, de la función `printf()` para que apunte al archivo de texto, usted llama a la función `printf()` en lugar de la función `fprintf()` u otra función de E/S de archivo en disco.

La función que en realidad hace la impresión es `ImprimeCadena()`, la cual llama a la función `printf()` de C para enviar las cadenas de caracteres formateadas al flujo de salida. (Vea la definición de `ImprimeCadena()` en las líneas 30 a 36.)

Dentro de la función `main()`, en la línea 20 se llama a la función `ImprimeCadena()` antes de redireccionar `stdout` al archivo `LaoTzu.txt`. No es ninguna sorpresa ver que se imprime en la pantalla el párrafo tomado del *Tao Te Ching*, ya que la función `printf()` envía automáticamente el párrafo a `stdout`, que de manera predeterminada lo dirige a la pantalla.

Después, en la línea 21 se redirecciona `stdout` al archivo de texto `LaoTzu.txt` por medio de la llamada a la función `freopen()`. En ella, `"w"` se usa como el modo que indica que se abra el archivo de texto para escritura. Si `freopen()` tiene éxito, entonces en la línea 24 se llama a la función `ImprimeCadena()`. Sin embargo, esta vez la función `ImprimeCadena()` escribe el párrafo en el archivo de texto abierto, `LaoTzu.txt`. La razón es que ahora `stdout` está asociado al archivo de texto, no a la pantalla, así que las cadenas que envía la llamada a `printf()` que está dentro de `ImprimeCadena()` se dirigen al archivo de texto.

Después de la ejecución del programa del listado 22.4, puede abrir el archivo `LaoTzu.txt` en un editor de texto y ver que el párrafo del *Tao Te Ching* se haya guardado en el archivo.

> Como mencioné antes, los flujos de E/S emplean búferes de manera predeterminada. En ocasiones podría necesitar desactivar los búferes para que pueda procesar de inmediato la entrada. En C hay dos funciones, `setbuf()` y `setvbuf()`, que usted puede utilizar para desactivar el uso de búferes, aunque dicho tema excede el alcance de este libro.
>
> También hay un conjunto de funciones de E/S de bajo nivel, como `open()`, `create()`, `close()`, `read()`, `write()`, `lseek()` y `tell()`, mismas que el estándar ANSI de C no maneja. Tal vez aún las vea en algunos programas de C dependientes de alguna plataforma. Para usarlos, necesita leer el manual de consulta de su compilador de C, para ver si están disponibles.

Resumen

En esta lección aprendió los siguientes conceptos y funciones importantes con respecto a la entrada y salida de archivos en disco de C:

- Se puede restablecer el indicador de posición del archivo mediante la función `fseek()`.
- La función `ftell()` puede indicarle el valor actual del indicador de posición del archivo.
- La función `rewind()` puede asignar el indicador de posición del archivo al inicio del mismo.
- Después de especificar el modo de la función `fopen()` para un archivo binario, puede usar las funciones `fread()` o `fwrite()` para realizar operaciones de E/S sobre datos binarios.
- Además del hecho de que las funciones `fscanf()` y `fprintf()` pueden hacer el mismo trabajo que las funciones `scanf()` y `printf()`, las primeras también pueden permitirle al programador especificar flujos de E/S.
- Con la ayuda de la función `freopen()`, es posible redireccionar los flujos estándar, como `stdin` y `stdout`, a un archivo en disco.

En la siguiente lección aprenderá acerca del preprocesador de C.

Preguntas y respuestas

P ¿Por qué es necesario el acceso aleatorio a un archivo en disco?

R Cuando necesita extraer una pieza de información de un archivo que contiene una enorme cantidad de datos, el acceso aleatorio al archivo resulta más eficiente que el acceso secuencial. Las funciones que realizan el acceso aleatorio pueden colocar el indicador de posición del archivo en el lugar justo, y entonces usted puede

Materiał chroniony prawem autorskim

simplemente comenzar a extraer desde ahí la información requerida. En C, `fseek()` y `ftell()` son dos útiles funciones para ayudarle a llevar a cabo la operación de acceso aleatorio.

P ¿Cómo especifica usted el formato de un nuevo archivo en disco que se va a crear por medio de una llamada a `fopen()`?

R Tiene que agregar `b` en el argumento de modo para la función `fopen()` para especificar que el archivo que se va a crear es un archivo binario. Puede usar `"wb"` para crear un nuevo archivo para escritura, y `"wb+"` para crear uno para lectura y escritura. Sin embargo, si el archivo a crear es un archivo de texto, no es necesaria la `b` en el argumento de modo.

P ¿Cuál es la diferencia entre las funciones `printf()` y `fprintf()`?

R Básicamente, las funciones `printf()` y `fprintf()` pueden realizar el mismo trabajo: enviar elementos de datos formateados a los flujos de salida. Sin embargo, la función `printf()` envía automáticamente datos formateados a `stdout`, mientras que a la función `fprintf()` se le puede asignar un apuntador de archivo que esté asociado a un flujo de salida específico.

P ¿Es posible redireccionar un flujo estándar a un archivo en disco?

R Sí. Con la ayuda de la función `freopen()`, puede redireccionar un flujo estándar y asociarlo a un archivo en disco.

Taller

Para ayudarle a consolidar la comprensión de la lección de esta hora, le recomiendo responder el cuestionario y terminar los ejercicios de este taller antes de pasar a la siguiente lección. Las respuestas y sugerencias se presentan en el apéndice B, "Respuestas a los cuestionarios y ejercicios".

Cuestionario

1. ¿Son equivalentes las siguientes instrucciones?

```
rewind(aptrf);
fseek(aptrf, 0L, SEEK_SET);
```

2. ¿Son equivalentes las siguientes instrucciones?

```
rewind(aptrf);
fseek(aptrf, 0L, SEEK_CUR);
```

3. Después de ejecutar con éxito la instrucción

```
freopen("prueba.txt", "r", stdin);
```

Materiał chroniony prawem autorskim

Qué es el preprocesador de C

Si una constante aparece en varias partes de su programa, es una buena idea asociarle un nombre simbólico a la constante y utilizar después dicho nombre para reemplazarla a lo largo del programa. Realizar esto tiene dos ventajas. Primero, su programa será más legible. Segundo, será más fácil darle mantenimiento. Por ejemplo, si necesita cambiar el valor de la constante, sólo tiene que encontrar la instrucción que asocia a la constante con el nombre simbólico y reemplazarla con una nueva. Si no utiliza el nombre simbólico, tiene que buscar la constante en todo el programa para reemplazarla. Esto suena muy bien, pero ¿cómo puede hacer esto en C?

Pues bien, C tiene un programa especial llamado preprocesador de C que le permite definir y asociar nombres simbólicos a constantes. De hecho, el preprocesador de C utiliza la terminología de *nombres de macro y cuerpo de macro* para referirse a los nombres simbólicos y a las constantes. El preprocesador de C se ejecuta antes que el compilador. Durante el preproceso, la operación de reemplazar un nombre de macro con su cuerpo de macro asociado se denomina *sustitución de macros o expansión de macros*.

Puede colocar una definición de macro en cualquier parte de su programa. Sin embargo, tiene que definir el nombre de macro antes de que pueda usarlo en su programa.

Además, el preprocesador de C le proporciona la capacidad de incluir otros archivos de código fuente. Por ejemplo, en los programas de este libro hemos venido utilizando la directiva de preprocesador `#include` para incluir archivos de encabezado de C, como `stdio.h`, `stdlib.h` y `string.h`. El preprocesador de C también le permite compilar diferentes secciones de su programa bajo condiciones específicas.

El preprocesador de C en comparación con el compilador

Algo importante que debe recordar es que el preprocesador no es parte del compilador.

El preprocesador de C utiliza una sintaxis diferente. Todas las directivas del preprocesador de C comienzan con un signo de numeral (#). En otras palabras, el signo de numeral indica el inicio de una directiva del preprocesador, y debe ser el primer carácter de la línea que no sea un espacio.

El preprocesador de C está orientado a la línea. Toda instrucción de macro termina con un carácter de línea nueva, no con un punto y coma. (Sólo las instrucciones de C terminan con punto y coma.) Uno de los errores más comunes que comete el programador es colocar un punto y coma al final de una instrucción de macro. Por fortuna, muchos compiladores de C pueden detectar dichos errores.

Las siguientes secciones describen algunas de las directivas de uso más frecuente, como `#define`, `#undef`, `#if`, `#elif`, `#else`, `#ifdef`, `#ifndef` y `#endif`.

```
17:       arreglo[i] = (i + 1) * NUM_NEGATIVO;
18:       printf("arreglo[%d]: %d\n", i, arreglo[i]);
19:    }
20:
21:    printf("\nValores aplicando la macro %s:\n", cadena);
22:    for (i=0; i<LONG_MAX; i++){
23:       printf("ABS(%d): %3d\n", arreglo[i], ABS(arreglo[i]));
24:    }
25:
26:    return 0;
27: }
```

Después de ejecutar el archivo `23L01.exe` del programa del listado 23.1, se desplegaron en la pantalla de mi computadora los siguientes resultados:

SALIDA

```
Valores originales del arreglo:
arreglo[0]: -10
arreglo[1]: -20
arreglo[2]: -30
arreglo[3]: -40
arreglo[4]: -50
arreglo[5]: -60
arreglo[6]: -70
arreglo[7]: -80

Valores aplicando la macro ABS:
ABS(-10):  10
ABS(-20):  20
ABS(-30):  30
ABS(-40):  40
ABS(-50):  50
ABS(-60):  60
ABS(-70):  70
ABS(-80):  80
```

ANÁLISIS La finalidad del programa del listado 23.1 es definir diferentes nombres de macros, incluyendo una macro de tipo función, y utilizarlos en el programa.

En las líneas 4 a 7 se definen, con la directiva `#define`, cuatro nombres de macro, `METODO`, `ABS`, `LONG_MAX` y `NUM_NEGATIVO`. Entre ellas, `ABS` puede aceptar un argumento. La definición de `ABS` en la línea 5 verifica el valor del argumento y devuelve el número absoluto del mismo. Observe que se utiliza el operador condicional `?:` para encontrar el valor absoluto del argumento entrante. (El operador `?:` se presentó en la hora 8, "Uso de operadores condicionales".)

Luego, dentro de la función `main()`, en la línea 11 se define el apuntador `cadena` de tipo `char` y se le asigna `METODO`. Como puede ver, `METODO` está asociado a la cadena `"ABS"`. En la línea 12 se define el arreglo de tipo `int` denominado `arreglo`, con el número de elementos que especifica `LONG_MAX`.

Compilación de su código bajo condiciones

Puede seleccionar partes de su programa de C que desee compilar, utilizando un conjunto de directivas del preprocesador. A esta técnica se le conoce como *compilación condicional*. Esto resulta muy útil, en especial si está probando una parte de código nuevo o depurando una porción de código.

Las directivas `#ifdef` y `#endif`

Las directivas `#ifdef` y `#endif` determinan si se incluirá como parte de su programa un grupo dado de instrucciones.

La forma general para utilizar las directivas `#ifdef` y `#endif` es

```
#ifdef nombre_macro
   instrucción1
   instrucción2
   . . .
   instrucciónN
#endif
```

Aquí, *`nombre_macro`* es cualquier cadena de caracteres que pueda ser definida por una directiva `#define`. *`instrucción1`*, *`instrucción2`* e *`instrucciónN`* son las instrucciones que se incluyen en el programa solamente si ya se definió *`nombre_macro`*. Si no se ha definido, se saltan *`instrucción1`*, *`instrucción2`* y todo lo demás hasta llegar a *`instrucciónN`*.

A diferencia de una instrucción `if` de C, las instrucciones bajo el control de la directiva `#ifdef` no se encierran entre llaves; en su lugar se debe utilizar la directiva `#endif` para marcar el final de un bloque `#ifdef`.

Por ejemplo, la directiva `#ifdef` del siguiente segmento de código:

```
. . .
#ifdef DEPURAR
   printf("El contenido de la cadena a la que apunta cad es: %s\n", cad);
#endif
. . .
```

indica que si se definió el nombre de macro `DEPURAR`, entonces se incluye en el programa la función `printf()` que está en la instrucción que sigue a la directiva `#ifdef`. El compilador compilará la instrucción para que imprima el contenido de la cadena a la que apunta `cad`. Sin embargo, si no se ha definido `DEPURAR`, la llamada a `printf()` quedará completamente fuera del programa compilado.

La directiva `#ifndef`

La directiva `#ifndef` le permite definir el código que se ejecutará cuando no esté definido un nombre de macro en particular.

Materiał chroniony prawem autorskim

El formato general para usar `#ifndef` es el mismo que para `#ifdef`:

```
#ifndef nombre_macro
   instrucción1
   instrucción2
   . . .
   instrucciónN
#endif
```

Aquí, *`nombre_macro`*, *`instrucción1`*, *`instrucción2`* e *`instrucciónN`* tienen el mismo significado que en la forma de `#ifdef` que se presentó en la sección anterior. Aquí también se necesita la directiva `#endif` para marcar el final del bloque `#ifndef`.

El listado 23.2 contiene un programa que muestra cómo usar juntas las directivas `#ifdef`, `#ifndef` y `#endif`.

TECLEE **LISTADO 23.2** Uso de las directivas `#ifdef`, `#ifndef` y `#endif`

```
1:  /* 23L02.c: Uso de #ifdef, #ifndef y #endif */
2:  #include <stdio.h>
3:
4:  #define MAYUSCULAS  0
5:  #define SIN_ERROR   0
6:
7:  main(void)
8:  {
9:     #ifdef MAYUSCULAS
10:      printf("ESTA LÍNEA SE IMPRIME,\n");
11:      printf("PORQUE MAYUSCULAS ESTÁ DEFINIDA.\n");
12:    #endif
13:    #ifndef MINUSCULAS
14:      printf("\nEsta línea se imprime,\n");
15:      printf("porque MINUSCULAS no está definida.\n");
16:    #endif
17:
18:    return SIN_ERROR;
19: }
```

Después de crear y ejecutar en mi computadora el archivo `23L02.exe`, se desplegaron en la pantalla los siguientes resultados:

SALIDA

```
ESTA LÍNEA SE IMPRIME,
PORQUE MAYUSCULAS ESTÁ DEFINIDA.

Esta linea se imprime,
porque MINUSCULAS no está definida.
```

ANÁLISIS La finalidad del programa del listado 23.2 es usar las directivas `#ifdef` e `#ifndef` para determinar si se despliega un mensaje.

En las líneas 4 y 5 se definen dos nombres de macros, `MAYUSCULAS` y `SIN_ERROR`.

La directiva `#ifdef` de la línea 9 verifica que se haya definido el nombre de macro `MAYUSCULAS`. Debido a que el nombre de la macro se definió en la línea 4, se incluyen en el programa compilado las dos instrucciones de las líneas 10 y 11 (hasta que la directiva `#endif` de la línea 12 marque el final del bloque `#ifdef`).

En la línea 13, la directiva `#ifndef` le indica al preprocesador que incluya en el programa las dos instrucciones de las líneas 14 y 15 si no se ha definido el nombre de macro `MINUSCULAS`. Como puede ver, `MINUSCULAS` no está definida en el programa. Por lo tanto, las instrucciones de las líneas 14 y 15 se compilan como parte del programa.

Los resultados de la ejecución del programa del listado 23.2 muestran que las funciones `printf()` de las líneas 10, 11, 14 y 15 se compilan y de acuerdo con ello se ejecutan, bajo el control de las directivas `#ifdef` e `#ifndef`. Puede intentar modificar el programa cambiando la línea 4 para que defina `MINUSCULAS` en vez de `MAYUSCULAS`. Entonces, las directivas `#ifdef` e `#ifndef` quitarán del programa las cuatro llamadas a `printf()`.

Las directivas #if, #elif y #else

La directiva `#if` especifica que se incluirán ciertas instrucciones sólo si el valor representado por la expresión condicional es diferente de cero. La expresión condicional puede ser una expresión aritmética.

La forma general para usar la directiva `#if` es

```
#if expresión
    instrucción1
    instrucción2
    . . .
    instrucciónN
#endif
```

Aquí, *expresión* es la expresión condicional a evaluar. *instrucción1*, *instrucción2* e *instrucciónN* representan el código a incluir si *expresión* es diferente de cero.

Observe que al final de la definición se incluye la directiva `#endif` para señalar el final del bloque `#if`, como sucede con los bloques `#ifdef` e `#ifndef`.

Además, la directiva `#else` proporciona una alternativa a elegir. La siguiente forma general usa la directiva `#else` para incluir en el programa las instrucciones *instrucción_1*, *instrucción_2* e *instrucción_N*, si *expresión* es cero:

```
#if expresión
    instrucción1
    instrucción2
    . . .
```

El preprocesador de C tiene otra directiva, `#elif`, que significa "else if". Puede emplear `#if` y `#elif` juntas para construir una cadena `if-else-if` para una compilación de condiciones múltiples.

El programa que se muestra en el listado 23.3 es un ejemplo del uso de las directivas `#if`, `#elif` y `#else`.

TECLEE **LISTADO 23.3** Uso de las directivas `#if`, `#elif` y `#else`

```
1:  /* 23L03.c: Uso de #if, #elif y #else */
2:  #include <stdio.h>
3:
4:  #define LENG_C     'C'
5:  #define LENG_B     'B'
6:  #define SIN_ERROR  0
7:
8:  main(void)
9:  {
10:    #if LENG_C == 'C' && LENG_B == 'B'
11:      #undef LENG_C
12:      #define LENG_C "Yo conozco el lenguaje C.\n"
13:      #undef LENG_B
14:      #define LENG_B "Yo conozco BASIC.\n"
15:      printf("%s%s", LENG_C, LENG_B);
16:    #elif LENG_C == 'C'
17:      #undef LENG_C
18:      #define LENG_C "Yo sólo conozco el lenguaje C.\n"
19:      printf("%s", LENG_C);
20:    #elif LENG_B == 'B'
21:      #undef LENG_B
22:      #define LENG_B "Yo sólo conozco BASIC.\n"
23:      printf("%s", LENG_B);
24:    #else
25:      printf("Yo no conozco ni C ni BASIC.\n");
26:    #endif
27:
28:    return SIN_ERROR;
29: }
```

Después de crear y ejecutar el archivo `23L03.exe`, se desplegaron en la pantalla de mi computadora los siguientes resultados.

SALIDA

```
Yo conozco el lenguaje C.
Yo conozco BASIC.
```

ANÁLISIS La finalidad del programa del listado 23.3 es utilizar las directivas `#if`, `#elif` y `#else` para seleccionar las partes del código que se van a compilar.

```
   #endif
#else
   printf("No se definió ningún nombre de macro.\n");
#endif
```

Aquí, la directiva `#if` está anidada a tres niveles. Observe que cada `#else` o `#endif` está asociada con la directiva `#if` más cercana.

Veamos ahora otro ejemplo en el listado 23.4, en el cual las directivas `#if` están anidadas.

TECLEE **LISTADO 23.4** Cómo anidar directivas `#if`

```
1:  /* 23L04.c: Cómo anidar #if */
2:  #include <stdio.h>
3:
4:  /* definición de macros */
5:  #define CERO      0
6:  #define UNO       1
7:  #define DOS       (UNO + UNO)
8:  #define TRES      (UNO + DOS)
9:  #define PRUEBA_1     UNO
10: #define PRUEBA_2     DOS
11: #define PRUEBA_3     TRES
12: #define MAX_NUM    TRES
13: #define SIN_ERROR  CERO
14: /* declaración de función */
15: void ImpCadena(char **aptr_s, int max);
16: /* la función main() */
17: main(void)
18: {
19:    char *cadena[MAX_NUM] = {"La elección de un punto de vista",
20:                             "es el acto inicial de la cultura.",
21:                             "--- por Ortega y Gasset"};
22:
23:    #if PRUEBA_1 == 1
24:       #if PRUEBA_2 == 2
25:          #if PRUEBA_3 == 3
26:             ImpCadena(cadena, MAX_NUM);
27:          #else
28:             ImpCadena(cadena, MAX_NUM - UNO);
29:          #endif
30:       #else
31:         ImpCadena(cadena, MAX_NUM - DOS);
32:       #endif
33:    #else
34:       printf("No se estableció ninguna macro PRUEBA.\n");
35:    #endif
```

continúa

4. Qué mensaje desplegará el siguiente segmento de código?

```
#define NOMBRE_MACRO  0
#ifdef NOMBRE_MACRO
   printf("Bajo #ifdef.\n");
#endif
#ifndef NOMBRE_MACRO
   printf("Bajo #ifndef.\n");
#endif
```

Ejercicios

1. En la hora 18, "Tipos de datos y funciones especiales", aprendió cómo definir datos `enum`. Reescriba el programa del listado 18.1 con la directiva `#define`.
2. Defina un nombre de macro que pueda multiplicar dos argumentos. Escriba un programa que calcule la multiplicación de `2` por `3`, con ayuda de la macro. Imprima el resultado de la multiplicación.
3. Reescriba el programa del listado 23.2 con las directivas `#if`, `#elif` y `#else`.
4. Reescriba el programa del listado 23.3 con directivas `#if` anidadas.

LISTADO 24.1 continuación

```
55:         /* busca el último nodo de la lista */
56:         for (aptr=aptr_inicial;
57:              aptr->aptr_siguiente != NULL;
58:              aptr=aptr->aptr_siguiente)
59:            ; /* aquí no hace nada */
60:         /* enlaza al último nodo */
61:         aptr->aptr_siguiente = nuevo_aptr;
62:     }
63: }
64: /**
65:  ** elimina_nodo_lista()
66:  **/
67: int elimina_nodo_lista(void)
68: {
69:     NODO *aptr, *aptr_guardado;
70:     unsigned long id;
71:     int eliminado = 0;
72:     int valdevol = 0;
73:
74:     if (aptr_inicial == NULL){
75:         printf("No hay nada por eliminar.\n");
76:         valdevol = 1;
77:     } else {
78:         printf("Escriba el ID del estudiante: ");
79:         scanf("%ld", &id);
80:
81:         if (aptr_inicial->id == id){
82:             aptr_guardado = aptr_inicial->aptr_siguiente;
83:             free(aptr_inicial);
84:             aptr_inicial = aptr_guardado;
85:             if (aptr_inicial == NULL){
86:                 printf("Se eliminaron todos los nodos.\n");
87:                 valdevol = 1;
88:             }
89:         } else {
90:             for (aptr=aptr_inicial;
91:                  aptr->aptr_siguiente != NULL;
92:                  aptr=aptr->aptr_siguiente){
93:                 if (aptr->aptr_siguiente->id == id){
94:                     aptr_guardado = aptr->aptr_siguiente->aptr_siguiente;
95:                     free(aptr->aptr_siguiente);
96:                     aptr->aptr_siguiente = aptr_guardado;
97:                     eliminado = 1;
98:                     break;
99:                 }
100:            }
101:            if (!eliminado){
102:                printf("No se encuentra el ID del estudiante.\n");
103:            }
```

Materiał chroniony prawem autorskim

Materiał chroniony prawem autorskim

No hay una salida directa del programa de módulo del listado 24.1.

ANÁLISIS La finalidad del programa del listado 24.1 es proporcionar un programa de módulo que contenga todas las funciones cohesivas para la creación de una lista enlazada, así como la incorporación y eliminación de nodos. La figura 24.2 muestra las tareas que realizan las funciones del programa, como son `crea_nodo_lista()`, `agrega_nodo_lista()` y `elimina_nodo_lista()`.

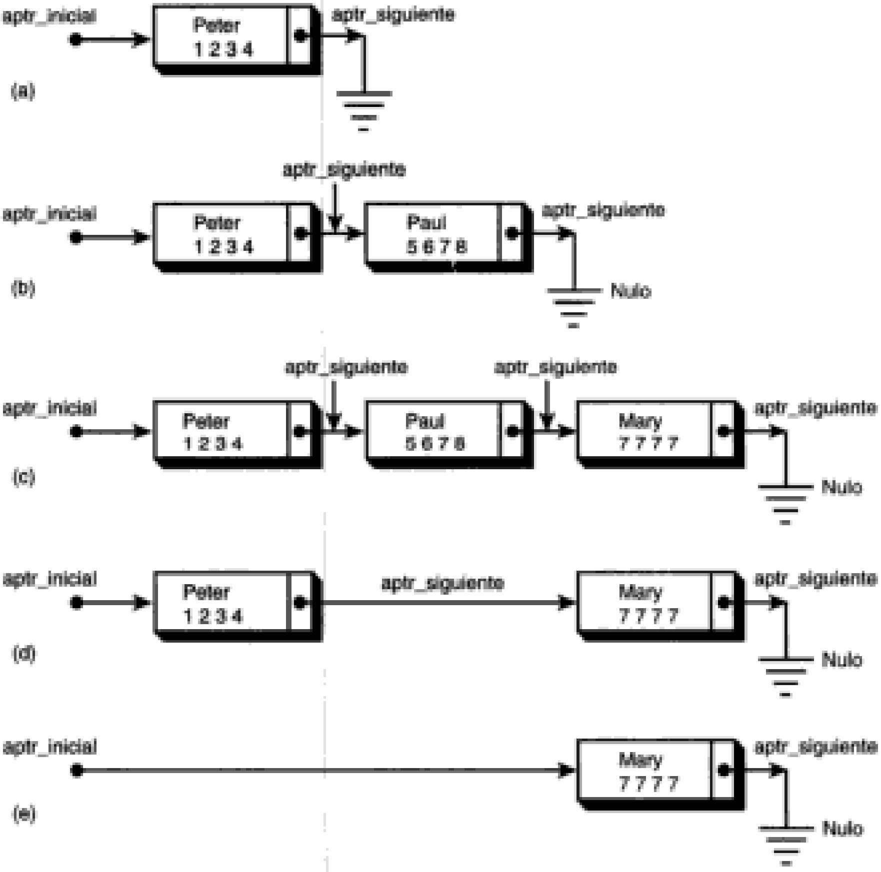


FIGURA 24.2
Uso de las funciones definidas en 24L01.c (los siguientes párrafos le proporcionan una explicación acerca de esta figura).

Como puede ver en la figura 24.2 (a), el primer nodo de la lista enlazada se crea llamando a la función `crea_nodo_lista()`, y los elementos de datos se agregan con la ayuda de la función `agrega_nodo_lista()`. Además, el apuntador `aptr_inicial` apunta al nodo. Aquí, el nombre del estudiante es `Peter`; su número de ID es `1234`. Debido a que no existen más nodos enlazados, el apuntador `aptr_siguiente` del primero nodo se asigna como nulo.

En la figura 24.2 (b) se incorpora otro nodo a la lista enlazada, con `Paul` como nombre del estudiante y `5678` como el número de ID. Observe que el apuntador `aptr_siguiente` del primer nodo se reasigna para apuntar al segundo nodo, mientras que el apuntador `aptr_siguiente` del segundo nodo se asigna como nulo.

De la misma manera, en la figura 24.2 (c) se agrega el tercer nodo a la lista enlazada. El apuntador `aptr_siguiente` del tercer nodo es un apuntador nulo. El apuntador del segundo nodo se reasigna para que apunte al tercer nodo.

Si necesita eliminar uno de los nodos, puede llamar a la función `elimina_nodo_lista()`. Como se muestra en la figura 24.2 (d), el segundo nodo es eliminado, así que el apuntador del primer nodo tiene que ser reasignado para que apunte al que antes era el tercer nodo, mismo que contiene el nombre de estudiante `Mary` y su número de ID, `7777`.

En la figura 24.2 (e) se elimina el primer nodo aplicando de nuevo la función `elimina_nodo_lista()`. Sólo queda un nodo en la lista enlazada. El apuntador `aptr_inicial` debe ser reasignado para que apunte al último nodo.

En el listado 24.2 se muestra el archivo de encabezado `24L02.h`, mismo que se incluye en el programa de módulo `24L01.c`. (Este archivo de encabezado también se incluye en el programa conductor del listado 24.3.)

TECLEE **LISTADO 24.2** Colocación de declaraciones de datos y prototipos de funciones en el archivo de encabezado

```
1:  /* 24L02.h: el archivo de encabezado */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:
5:  #ifndef ENC_LISTA_ENL
6:  #define ENC_LISTA_ENL
7:  #define BANDERA_ERR  1
8:  #define LONG_MAX      16
9:
10: struct estructura_lista_enl
11: {
12:    char nombre[LONG_MAX];
13:    unsigned long id;
14:    struct estructura_lista_enl *aptr_siguiente;
15: };
16:
17: typedef struct estructura_lista_enl NODO;
18:
19: NODO *crea_nodo_lista(void);
20: void agrega_nodo_lista(void);
21: int elimina_nodo_lista(void);
22: void imprime_nodo_lista(void);
23: void nodo_lista_libre(void);
24: void TerminaError(char *);
25: void interfaz_principal(int);
26:
27: #endif /* de ENC_LISTA_ENL */
```

Materiał chroniony prawem autorskim

No hay una salida directa del programa del listado 24.2.

ANÁLISIS La finalidad del programa del listado 24.2 es declarar una estructura con el nombre de etiqueta `estructura_lista_enl` (vea las líneas 10 a 15), y definir un nuevo nombre de variable, `NODO`, para la estructura (vea la línea 17).

En las líneas 19 a 25 se listan los prototipos de las funciones definidas en el programa de módulo del listado 24.1, como son `crea_nodo_lista()`, `agrega_nodo_lista()` y `elimina_nodo_lista()`.

Observe que en las líneas 5 y 27 se usan las directivas de preprocesador `#ifndef` y `#endif`. Las declaraciones y definiciones colocadas entre las dos directivas se compilan solamente si no ha sido definido el nombre de macro `ENC_LISTA_ENL`. Además, la línea 6 define el nombre de macro si aún no ha sido definido. Es una buena idea poner las directivas `#ifndef` y `#endif` en un archivo de encabezado a fin de evitar inclusiones cruzadas al incluir el archivo de encabezado en más de un archivo de código fuente. En este caso, las declaraciones y definiciones del archivo de encabezado `24L02.h` no se incluirán más de una vez.

El programa de módulo del listado 24.3 proporciona una interfaz que el usuario puede emplear para llamar a las funciones almacenadas en el archivo fuente (`24L01.c`).

TECLEE **LISTADO 24.3** Cómo llamar a las funciones almacenadas en el programa de módulo.

```
1:  /* 24L03.c: El archivo conductor */
2:  #include "24L02.h"     /* incluye el archivo de encabezado */
3:
4:  main(void)
5:  {
6:     int ch;
7:
8:     printf("Introduzca a para agregar, e para eliminar,\n");
9:     printf("x para exhibir y s para salir:\n");
10:    while ((ch=getchar()) != 'q'){
11:       interfaz_principal(ch);   /* procesa la entrada del usuario */
12:    }
13:
14:    nodo_lista_libre();
15:    printf("\n¡Hasta luego!\n");
16:
17:    return 0;
18: }
```

Compilé por separado los archivos fuente `24L01.c` y `24L03.c`, con Microsoft Visual C++, y después enlacé sus archivos objeto con las funciones de biblioteca de C para producir un solo programa ejecutable denominado `24L03.exe`. Obtuve los siguientes resultados

Debe tener cuidado al usar operadores de C que utilizan los mismos símbolos, en especial con el operador de asignación (=) y el operador condicional (==), ya que cualquier uso equivocado de estos dos operadores puede conducir a resultados inesperados y dificultar mucho la depuración.

Evite usar la instrucción `goto`; en su lugar, utilice instrucciones de control de flujo en donde sea necesario.

Utilice en su programa constantes con nombre en vez de constantes numéricas, ya que las primeras pueden hacer más legible su programa y usted sólo tendrá que ir a un lugar para actualizar los valores de las constantes.

Para evitar efectos colaterales, debe encerrar entre paréntesis cada expresión o argumento constante definido por una directiva de preprocesador.

Además debe establecer una regla razonable para el espaciado y el sangrado, de manera que pueda seguir dicha regla en forma consistente en todos los programas que escriba. La regla debe ayudarle a escribir programas fáciles de leer.

Programación modular

No es una buena práctica de programación tratar de resolver un problema complejo con una sola función. La forma adecuada de afrontarlo, es dividir el problema en varios problemas más pequeños y simples que se puedan entender con más detalle, y luego comenzar a definir y construir las funciones para resolverlos. Tenga presente que cada una de sus funciones debe hacer sólo una tarea, pero hacerla bien.

Cuando su programa se haga más y más grande, debe considerar dividirlo en varios archivos de código fuente, y que cada uno de éstos contenga un grupo reducido de funciones cohesivas. A dichos archivos de código fuente se les conoce también como *módulos*. Ponga las declaraciones de datos y los prototipos de funciones dentro de archivos de encabezado, para que cualquier cambio a las declaraciones o prototipos pueda aparecer en forma automática en todos los archivos de código fuente que incluyan el archivo de encabezado.

Por ejemplo, en la sección "Creación de una lista enlazada", todas las funciones que se pueden utilizar para crear una lista enlazada y agregar o eliminar un nodo se colocan dentro del mismo módulo (`24L01.c`). La estructura de datos y las declaraciones de variables y prototipos de funciones se guardan dentro de un archivo de encabezado (`24L02.h`). La función `main()` y la interfaz se guardan dentro de otro módulo (`24L03.c`).

Para reducir la complejidad de programación, puede emplear una técnica de ingeniería de software conocida como *ocultamiento de información*. Hablando llanamente, el ocultamiento de información requiere que un módulo no proporcione información a otro módulo, a menos de que sea muy necesario.

TABLA 24.1 continuación

<table>
<tr><th>Operador</th><th>Descripción</th></tr>
<tr><td><</td><td>Menor que</td></tr>
<tr><td>>=</td><td>Mayor o igual que</td></tr>
<tr><td><=</td><td>Menor o igual que</td></tr>
<tr><td>sizeof</td><td>Tamaño de</td></tr>
<tr><td>&&</td><td>Operador lógico AND</td></tr>
<tr><td>||</td><td>Operador lógico OR</td></tr>
<tr><td>!</td><td>Operador lógico NOT</td></tr>
<tr><td>&</td><td>Operador AND a nivel de bits</td></tr>
<tr><td>|</td><td>Operador OR a nivel de bits</td></tr>
<tr><td>^</td><td>Operador OR exclusivo (XOR) a nivel de bits</td></tr>
<tr><td>~</td><td>Operador de complemento a nivel de bits</td></tr>
<tr><td>>></td><td>Operador de desplazamiento a la derecha a nivel de bits</td></tr>
<tr><td><<</td><td>Operador de desplazamiento a la izquierda a nivel de bits</td></tr>
<tr><td>?:</td><td>Operador condicional</td></tr>
</table>

Constantes

Las *constantes* son elementos cuyos valores no cambian en el programa. Hay varios tipos de constantes en C.

Constantes enteras

Las *constantes enteras* son números decimales. Puede ponerle los sufijos `u` o `U` a una constante entera para especificar que es del tipo de datos `unsigned`. Una constante entera con los sufijos `l` o `L` es una constante de tipo `long int`.

A una constante entera se le pone el prefijo `0` (cero) para indicar que está en formato octal. Si a una constante entera se le ponen los prefijo `0X` o `0x`, entonces ésta es un número hexadecimal.

Constantes de carácter

Una constante de carácter es un carácter encerrado entre comillas sencillas. Por ejemplo, `'C'` es una constante de carácter.

En C hay varias constantes de carácter que representan ciertos caracteres especiales (vea la tabla 24.2).

Materiał chroniony prawem autorskim

TABLA 24.2 Caracteres especiales de C

Carácter	*Significado*
`\a`	Alerta audible
`\b`	Retroceso
`\f`	Avance de página
`\n`	Nueva línea
`\r`	Retorno de carro
`\t`	Tabulador horizontal
`\v`	Tabulador vertical
`\"`	Comillas dobles
`\'`	Comilla sencilla
`\0`	Nulo
`\\`	Diagonal invertida
`\N`	Constante octal (aquí, `N` es una constante octal)
`\xN`	Constante hexadecimal (aquí, `N` es una constante hexadecimal)

Constantes de punto flotante

Las *constantes de punto flotante* son números decimales que pueden tener los sufijos `f`, `F`, `l` o `L` para especificar que son de los tipos de datos `float` o `long double`. Una constante de punto flotante sin sufijo es, de manera predeterminada, del tipo de datos `double`. Por ejemplo, las siguientes instrucciones declaran e inicializan una variable `float` (`num_flt`) y una variable `double` (`num_dbl`):

```
float   num_flt = 1234.56f;
double  num_dbl = 1234.56;
```

Un número de punto flotante también se puede representar en la notación científica.

Constantes de cadena

Una constante de cadena es una secuencia de caracteres encerrados ente comillas dobles. Por ejemplo, `"Ésta es una constante de cadena."` es una constante de cadena. Observe que las comillas dobles no son parte del contenido de la cadena. Además, el compilador de C agrega automáticamente un carácter nulo (`\0`) al final de una constante de cadena para indicar el final de la cadena.

Tipos de datos

Los *tipos de datos* básicos que proporciona el lenguaje C son `char`, `int`, `float` y `double`. También están los tipos de datos `array`, `enum`, `struct` y `union` que usted puede declarar y usar en sus programas de C.

Aquí, `struct` es la palabra reservada que se usa en C para iniciar una declaración de estructura. *`etiq_estructura`* es el nombre de etiqueta de la estructura. *`variable1`*, *`variable2`* y *`variable3`* son los miembros de la estructura. Sus tipos de datos se especifican, respectivamente, mediante *`tipo_datos1`*, *`tipo_datos2`* y *`tipo_datos3`*. En la declaración de la estructura, las declaraciones de los miembros tienen que estar encerradas entre llaves de apertura y de cierre ({ y }), y se debe incluir un punto y coma (;) al final de la declaración.

El siguiente es un ejemplo de una declaración de estructura:

```
struct auto {
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
      };
```

Aquí, `struct` se usa para iniciar una declaración de estructura. `auto` es el nombre de etiqueta de la estructura. En este ejemplo hay tres tipos de variables, `char`, `int` y `float`. Las variables tienen sus propios nombres, como `anio`, `modelo`, `potencia_motor` y `peso`. Todos son miembros de la estructura, y están declarados con las llaves ({ y }).

El tipo de datos `union`

Una *unión* es un bloque de memoria que se usa para contener elementos de datos de diferentes tipos. En C, una unión es similar a una estructura, excepto que los elementos de datos que se guardan en la unión están sobrepuestos, a fin de compartir la misma ubicación de memoria. La sintaxis para declarar una unión es parecida a la de una estructura. La forma general para declarar una unión es

```
union etiq_union {
      tipo_datos1 variable1;
      tipo_datos2 variable2;
      tipo_datos3 variable3;
      .
      .
      .
      };
```

Aquí, `union` es la palabra clave que se usa en C para iniciar una declaración de unión. *`etiq_union`* es el nombre de etiqueta de la unión. *`variable1`*, *`variable2`* y *`variable3`* son los miembros de la unión. Sus tipos de datos respectivos están especificados por *`tipo_datos1`*, *`tipo_datos2`* y *`tipo_datos3`*. La declaración de la unión termina con un punto y coma (;).

El siguiente es un ejemplo de una declaración de unión:

```
union auto {
      int anio;
      char modelo[8];
      int potencia_motor;
      float peso;
};
```

Materiał chroniony prawem autorskim

Aquí, `union` especifica el tipo de datos de la unión. `auto` es el nombre de etiqueta de la unión. Las variables, como `anio`, `modelo`, `potencia_motor` y `peso`, son los miembros de la unión y están declarados dentro de las llaves ({ y }).

Definición de nuevos nombres de tipos de datos con typedef

Usted puede crear sus propios nombres de tipos de datos con ayuda de la palabra reservada `typedef`, y usar dichos nombres como sinónimos de los tipos de datos. Por ejemplo, puede declarar `NUMERO` como un sinónimo del tipo de datos `int`:

```
typedef  int  NUMERO;
```

Luego puede comenzar a emplear `NUMERO` para declarar variables enteras de la siguiente manera:

```
NUMERO i, j;
```

lo que equivale a

```
int i, j;
```

Recuerde que debe hacer la definición de `typedef` antes de utilizar en cualquier declaración de su programa el sinónimo creado en la definición.

Expresiones e instrucciones

Una *expresión* es una combinación de constantes o variables que se utiliza para denotar cálculos.

Por ejemplo,

```
(2 + 3) * 10
```

es una expresión que primero suma `2` y `3`, y luego multiplica por `10` el resultado de la suma.

En el lenguaje C, una *instrucción* es una entidad completa que termina con un punto y coma. En muchos casos, puede convertir una expresión en una instrucción con sólo agregar un punto y coma al final de la expresión.

Una *instrucción nula* está representada por un punto y coma aislado.

Un grupo de instrucciones puede formar un *bloque* que inicie con una llave de apertura ({) y termine con una llave de cierre (}). El compilador de C trata a un bloque de instrucciones como a una sola instrucción.

Instrucciones de control de flujo

En C hay un conjunto de instrucciones de control de flujo que se pueden dividir en dos categorías: ciclos y ramificación (o bifurcación) condicional.

Los ciclos `for`, `while` y `do-while`

La forma general de la instrucción `for` es

```
for (expresión1; expresión2; expresión3) {
    instrucción1;
    instrucción2;
    .
    .
    .
}
```

La instrucción `for` primero evalúa a *expresión1*, la cual es, por lo general, una expresión que inicializa una o más variables. La segunda expresión, *expresión2*, es la parte condicional que evalúa y prueba la instrucción `for` para cada iteración. Si *expresión2* se evalúa como un valor diferente de cero, se ejecutan las expresiones que están entre las llaves, *instrucción1* e *instrucción2*. Si *expresión2* se evalúa como `0` (cero), el ciclo se detiene y concluye la ejecución de la instrucción `for`. La tercera expresión de la instrucción `for`, *expresión3*, se evalúa después de cada iteración, antes de que la instrucción regrese a probar de nuevo a *expresión2*. Esta tercera expresión se usa a menudo para incrementar el valor de una variable de índice.

La siguiente instrucción `for` crea un ciclo infinito debido a que las tres expresiones están vacías:

```
for ( ; ; ){
   /* bloque de instrucciones */
}
```

La forma general de la instrucción `while` es

```
while (expresión) {
    instrucción1;
    instrucción2;
    .
    .
    .
}
```

Aquí, *expresión* es la expresión condicional de la instrucción `while`. Primero se evalúa la expresión. Si se evalúa como un valor diferente de cero, el ciclo continúa; es decir, se ejecutan las instrucciones que están dentro del bloque, *instrucción1* e *instrucción2*. Después de la ejecución, se evalúa de nuevo la expresión. Si ésta aún se evalúa como diferente de cero, las instrucciones se ejecutan una vez más. El proceso se repite una y otra vez hasta que la expresión se evalúe como cero.

Materiał chroniony prawem autorskim

También puede hacer un ciclo `while` infinito poniendo `1` (uno) en el campo de expresión, como se indica a continuación:

```
while (1) {
   /* bloque de instrucciones */
}
```

La forma general de la instrucción `do-while` es

```
do {
   instrucción1;
   instrucción2;
   .
   .
   .
} while (expresión);
```

Aquí, *expresión* es la expresión condicional que se evalúa a fin de determinar si las instrucciones que están dentro del bloque se ejecutan una vez más. Si la expresión se evalúa como un valor diferente de cero, el ciclo `do-while` continúa; en caso contrario, la iteración se detiene. Observe que la instrucción `do-while` termina con un punto y coma, lo que es una diferencia importante. Se garantiza que las instrucciones que controla la instrucción `do-while` se ejecuten por lo menos una vez antes de evaluar la expresión condicional.

Ramificación condicional

Las instrucciones `if`, `if-else`, `switch`, `break`, `continue` y `goto` están dentro de la categoría de ramificación condicional.

La forma general de la instrucción `if` es

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

Aquí, *expresión* es la expresión condicional. Si *expresión* se evalúa como diferente de cero, se ejecutan las instrucciones que están dentro de las llaves, *instrucción1* e *instrucción2*. Si *expresión* se evalúa como `0`, dichas instrucciones se saltan.

Como una ampliación de la instrucción `if`, la instrucción `if-else` tiene la siguiente forma:

```
if (expresión) {
   instrucción1;
   instrucción2;
   .
   .
   .
```

```
}
else {
   instrucción_A;
   instrucción_B;
   .
   .
   .
}
```

Aquí, si `expresión` se evalúa como diferente de cero, se ejecutan las instrucciones que controla el `if`, incluyendo a `instrucción1` e `instrucción2`. En caso contrario, se ejecutan las instrucciones `instrucción_A` e `instrucción_B`, que están dentro del bloque que sigue a la palabra reservada `else`.

La forma general de la instrucción `switch` es

```
switch (expresión) {
   case expresión1:
         instrucción1;
   case expresión2:
         instrucción2;
   .
   .
   .
   default:
         instrucción_predeterminada;
}
```

Aquí se evalúa primero la expresión condicional `expresión`. Si el valor que produce es igual a la expresión constante `expresión1`, la ejecución comienza en la instrucción `instrucción1`. Si el valor de `expresión` es el mismo que el de `expresión2`, entonces la ejecución comienza en `instrucción2`. Sin embargo, si el valor de `expresión` no es igual a ninguno de los valores de las expresiones constantes etiquetadas por la palabra reservada `case`, se ejecuta la instrucción `instrucción_predeterminada` que sigue a la palabra reservada `default`.

Si necesita salir de la estructura de `switch` después de ejecutar las instrucciones que están dentro de un caso seleccionado, puede agregar una instrucción `break` al final de la lista de instrucciones que sigue a cada etiqueta `case`.

La instrucción `break` también se puede usar para interrumpir un ciclo infinito.

En ocasiones necesitará permanecer en un ciclo, pero saltarse algunas de las instrucciones que están dentro de éste. Para hacerlo, puede utilizar la instrucción `continue`.

La siguiente es la forma general de la instrucción `goto`:

```
nombre_etiqueta:
   instrucción1;
   instrucción2;
```

Materiał chroniony prawem autorskim


```
.
.
.
goto  nombre_etiqueta;
```

Aquí, *nombre_etiqueta* es un nombre de etiqueta que le indica a la instrucción goto a dónde saltar. Debe colocar el *nombre_etiqueta* en dos lugares: en el sitio al que va a saltar la instrucción goto y después de la palabra reservada goto. Además, el lugar al que va a saltar la instrucción goto puede estar antes o después de la propia instrucción. Observe que, en el lugar al que saltará la instrucción goto, el nombre de etiqueta debe ir seguido de dos puntos (:). Sin embargo, no se recomienda el uso de goto debido a que hace que su programa sea difícil de depurar.

Apuntadores

Un *apuntador* es una variable cuyo valor se utiliza para apuntar a otra variable. La forma general de la declaración de un apuntador es

```
tipo_de_datos  *nombre_del_apuntador;
```

Aquí, *tipo_de_datos* especifica el tipo de datos a los que apunta el apuntador. *nombre_del_apuntador* es el nombre de la variable de apuntador, el cual puede ser cualquier nombre de variable válido en C. Cuando el compilador ve en la declaración el asterisco (*) como prefijo del nombre de la variable, toma nota de que la variable se puede utilizar como un apuntador.

Por lo regular, la dirección asociada a un nombre de variable se conoce como *valor izquierdo* de la variable. Cuando a una variable se le asigna un valor, éste se almacena como contenido dentro de la ubicación de memoria reservada de la variable. Al contenido también se le llama *valor derecho* de la variable.

Se dice que un apuntador es nulo cuando se le asigna el valor derecho de 0. Recuerde que un apuntador nulo nunca puede apuntar a datos válidos, por lo que puede usarlo para probar la validez de un apuntador.

El operador de indirección (*) es un operador unario que sólo requiere un operando. Por ejemplo, la expresión *nombre_aptr produce el valor al que apunta la variable de apuntador nombre_aptr, en donde nombre_aptr puede ser cualquier nombre de variable válido en C.

Al operador & se le conoce como el *operador de dirección*, debido a que puede devolver la dirección (es decir, el valor izquierdo) de una variable.

Varios apuntadores pueden apuntar a la misma ubicación de memoria. En C, usted puede mover la posición de un apuntador sumando o restando enteros al apuntador.

Observe que para apuntadores de diferentes tipos de datos, los enteros que se suman o se restan a los apuntadores tienen tamaños escalares distintos.

Cómo apuntar a objetos

Es posible acceder a un elemento de un arreglo mediante un apuntador. Por ejemplo, dado el arreglo `un_arreglo` y el apuntador `aptr_arreglo`, si `un_arreglo` y `aptr_arreglo` son del mismo tipo de datos, y a `aptr_arreglo` se le asigna la dirección de inicio del arreglo, como se presenta a continuación:

```
aptr_arreglo = un_arreglo;
```

la expresión

```
un_arreglo[n]
```

equivale a la expresión

```
*(aptr_arreglo + n)
```

Aquí, `n` es un índice del arreglo.

En muchos casos, resulta útil declarar un arreglo de apuntadores y acceder al contenido al que apunta el arreglo mediante la desreferenciación de cada apuntador. Por ejemplo, la siguiente es una declaración de un arreglo de apuntadores:

```
int *aptr_int[3];
```

En otras palabras, la variable `aptr_int` es un arreglo de tres elementos de apuntadores de tipo `int`.

Además, puede definir un apuntador de tipo `struct` y hacer referencia a un elemento de la estructura por medio del apuntador. Por ejemplo, dada la siguiente declaración de estructura:

```
struct computadora {
   float costo;
   int anio;
   int velocidad_cpu;
   char tipo_cpu [16];
};
```

se puede definir un apuntador de la siguiente manera:

```
struct computadora *aptr_s;
```

Entonces, se puede acceder a todos los elementos de la estructura desreferenciando el apuntador. Por ejemplo, para asignar el valor `1997` a la variable `anio` de tipo `int` de la estructura `computadora`, puede emplear la siguiente instrucción de asignación:

```
(*aptr_s).anio = 1997;
```

O bien, puede usar el operador de flecha (`->`) para la asignación, de la siguiente manera:

```
aptr_s->anio = 1997;
```

Materiał chroniony prawem autorskim

Observe que el operador de flecha (->) se usa por lo regular para hacer referencia a un miembro de una estructura asociado con un apuntador que apunta a la estructura.

Funciones

Las *funciones* son los bloques de construcción de los programas de C. Además de las funciones estándar de la biblioteca de C, también puede usar en su programa otras funciones creadas por usted u otro programador. La llave de apertura ({) indica el inicio del cuerpo de una función, mientras que la llave de cierre (}) marca el final del cuerpo de una función.

De acuerdo con el estándar ANSI, la *declaración* de una variable o función especifica la interpretación y atributos de un conjunto de identificadores. La *definición*, por otra parte, requiere que el compilador de C reserve espacio de almacenamiento para la variable o función nombrada por un identificador.

De hecho, una declaración de variable es una definición. Pero esto mismo no es cierto para las funciones. Una *declaración de función* hace referencia a una función que está definida en alguna otra parte, y especifica qué tipo de valor devuelve la función. Una *definición de función* define lo que hace la función, así como el número y tipo de argumentos que se le pasan.

El estándar ANSI permite que el número y los tipos de argumentos que se pasan a una función se agreguen dentro de la declaración de la función. El número y los tipos de argumentos se llaman *prototipo de la función*.

La forma general de una declaración de función, incluyendo su prototipo, es la siguiente:

```
especificador_de_tipo_de_datos  nombre_de_función(
      especificador_de_tipo_de_datos nombre_de_argumento1,
      especificador_de_tipo_de_datos nombre_de_argumento2,
      especificador_de_tipo_de_datos nombre_argumento3,
      .
      .
      .
      especificador_de_tipo_de_datos nombre_de_argumentoN,
);
```

Aquí, `especificador_de_tipo_de_datos` determina el tipo de valor devuelto por la función, o especifica los tipos de datos de los argumentos, como *nombre_de_argumento1*, *nombre_de_argumento2*, *nombre_de_argumento3* y *nombre_de_argumentoN*, que se pasan a la función denominada *nombre_de_función*.

La finalidad de usar un prototipo de función es ayudarle al compilador a verificar que los tipos de datos de los argumentos que se pasan a una función coincidan con los que ésta espera. Si no coinciden, el compilador emite un mensaje de error. El tipo de datos `void` es necesario en la declaración de una función que no toma ningún argumento.

Para declarar una función que toma un número variable de argumentos, debe especificar por lo menos el primer argumento y usar los puntos suspensivos (...) para representar el resto de los argumentos que se pasan a la función.

Una llamada a una función es una expresión que se puede utilizar como una sola instrucción o dentro de otras expresiones o instrucciones.

A menudo es más eficiente pasar a una función la dirección de un argumento en lugar de una copia del mismo, a fin de que la función pueda acceder al valor original del argumento y manipularlo. Por lo tanto, es una buena idea pasar el nombre de un apuntador, que apunte a un arreglo, como argumento a una función, en vez del arreglo de apuntadores mismo.

También se puede llamar a una función por medio de un apuntador que contenga la dirección de la función.

Entrada y salida (E/S)

En C, un *archivo* se refiere a un archivo en disco, una terminal, una impresora o una unidad de cinta. En otras palabras, un archivo representa un dispositivo concreto con el que usted desea intercambiar información. Por otra parte, un *flujo* es una serie de bytes a través de los cuales puede leer o escribir datos en un archivo. A diferencia de un archivo, un flujo es independiente de un dispositivo. Todos los flujos tienen el mismo comportamiento.

Además, hay tres flujos de archivo que están preabiertos para usted:

- `stdin`—La entrada estándar para lectura.
- `stdout`—La salida estándar para escritura.
- `stderr`—El flujo de error estándar para escribir mensajes de error.

Por lo regular, la entrada estándar `stdin` está asociada al teclado, mientras que la salida estándar `stdout` y el flujo de error estándar `stderr` están direccionados a la pantalla. Además, varios sistemas operativos le permiten redireccionar estos flujos de archivo.

De manera predeterminada, los flujos de E/S utilizan búferes. A la E/S con búferes se le conoce también como *E/S de alto nivel*.

La estructura `FILE` es la estructura de control de archivos definida en el archivo de encabezado `stdio.h`. Un apuntador de tipo `FILE` se llama *apuntador de archivo* y hace referencia a un archivo en disco. Un flujo emplea un apuntador de archivo para conducir la operación de las funciones de E/S. Por ejemplo, a continuación tenemos la definición de un apuntador de archivo llamado `aptrf`:

```
FILE *aptrf;
```

En la estructura `FILE` hay un miembro llamado *indicador de posición del archivo*, el cual apunta a la posición en la que se leerán o escribirán datos.

El lenguaje C proporciona un rico conjunto de funciones de biblioteca para realizar operaciones de E/S. Dichas funciones pueden leer o escribir en archivos cualquier tipo de datos.

Materiał chroniony prawem autorskim

En este libro le he presentado las siguientes funciones: `fopen()`, `fclose()`, `fgetc()`, `fputc()`, `fgets()`, `fputs()`, `fread()`, `fwrite()`, `feof()`, `fscanf()`, `fprintf()`, `fseek()`, `ftell()`, `rewind()` y `freopen()`.

El preprocesador de C

El *preprocesador de C* no es parte del compilador de C. El preprocesador de C se ejecuta antes que el compilador. Durante la etapa de preprocesamiento, todas las ocurrencias de un nombre de macro se reemplazan por el cuerpo de la macro asociada a dicho nombre. Observe que una instrucción de macro termina con un carácter de línea nueva, no con un punto y coma.

El preprocesador de C también le permite incluir archivos de código fuente adicionales en el programa o compilar secciones de código de C de manera condicional.

La directiva `#define` le indica al preprocesador que reemplace todas las ocurrencias de un nombre de macro definido por ésta, con el cuerpo de la macro asociado a ese nombre. Puede especificar uno o más argumentos para un nombre de macro definido por la directiva `#define`.

La directiva `#undef` se utiliza para remover la definición de un nombre de macro definido con anterioridad.

La directiva `#ifdef` determina si se incluirá como parte del programa un grupo determinado de instrucciones. La directiva `#ifndef` es la directiva opuesta a `#ifdef`; le permite definir el código que se incluirá cuando no esté definido un nombre de macro en particular.

Las directivas `#if`, `#elif` y `#else` le permiten eliminar porciones del código a compilar. `#endif` se utiliza para marcar el final de un bloque `#ifdef`, `#ifndef` o `#if`, ya que las instrucciones bajo el control de estas directivas del preprocesador no se encierran entre llaves.

El camino a seguir...

Creo que una vez que, por medio de este libro, ha aprendido a caminar a través del mundo de la programación del lenguaje C, ahora puede empezar a correr. Aunque aprenda por su cuenta, no está solo. Puede visitar de nuevo este libro siempre que lo considere necesario. Le recomiendo los siguientes libros, los cuales también pueden guiarle en su continua jornada por el mundo de C:

The C Programming Language

de Brian Kernighan y Dennis Ritchie, editado por Prentice Hall

C Interfaces and Implementations

de David Hanson, editado por Addison-Wesley

Practical C Programming
de Steve Oualline, editado por O'Reilly & Associates, Inc.

No Bugs!—Delivering Error-Free Code in C and C++
de David Thielen, editado por Addison-Wesley

Software Engineering
de Ian Sommerville, editado por Addison-Wesley

The Mythical Man-Month: Essays on Software Engineering
de F. P. Brooks, Jr., editado por Addison-Wesley

Dynamics of Software Development: "Don't Flip the Bozo Bit" and 53 More Rules for Delivering Great Software on Time
de Jim McCarthy, editado por Microsoft Press
ISBN: 1-55615-823-8

Code Complete: A Practical Handbook of Software Construction
de Steve McConnell, editado por Microsoft Press
ISBN: 1-55615-484-4

Resumen

Antes de cerrar este libro, quisiera agradecerle por la paciencia y el esfuerzo que dedicó a aprender las bases del lenguaje C durante las pasadas 24 horas. (Creo que usted habría podido dedicar más de 24 horas. Es muy normal. Aún quedan muchas más cosas acerca de la programación en C que aprenderá sobre la marcha. Pero si ya domina los conceptos básicos, operadores y funciones que se enseñaron en este libro, puede aprender cosas nuevas con rapidez.) Ahora es su turno de aplicar lo que aprendió en este libro resolviendo problemas de la vida real. ¡Buena suerte!

Materiał chroniony prawem autorskim

PARTE VI
Apéndices

Hora

A Archivos de encabezado del estándar ANSI

B Respuestas a los cuestionarios y ejercicios

Obraz chroniony prawem autorskim

Materiał chroniony prawem autorskim

APÉNDICE A

Archivos de encabezado del estándar ANSI

Como aprendió en las últimas 24 horas, la biblioteca estándar de C comprende un conjunto de archivos de inclusión denominados *archivos de encabezado*. Estos archivos contienen las declaraciones de las macros y funciones de biblioteca de C, así como de tipos de datos relevantes. Siempre que invoque una función de C, tiene que incluir en su programa el archivo o archivos con los que está asociada dicha función.

Los siguientes son los archivos de encabezado del estándar ANSI:

Archivo	*Descripción*
`assert.h`	Contiene funciones de diagnóstico.
`ctype.h`	Contiene funciones de prueba y asociación de caracteres.
`errno.h`	Contiene constantes para procesar errores.
`float.h`	Contiene constantes para valores de punto flotante.
`limits.h`	Contiene valores dependientes de la implementación.

continúa

4. Necesitamos el estándar ANSI de C para garantizar la portabilidad de los programas escritos en este lenguaje. La mayoría de los fabricantes de compiladores de C manejan el estándar ANSI. Si escribe su programa siguiendo las reglas establecidas por el estándar ANSI, puede portar su programa a cualquier máquina simplemente recompilándolo con un compilador que soporte esas máquinas.

Hora 2, "Su primer programa de C"

Cuestionario

1. No. De hecho, el preprocesador dejará fuera todos los comentarios que ponga en su programa antes de que el compilador pueda verlos. Usted escribe los comentarios para usted mismo y para otros programadores que vean su programa.
2. Después de que el compilador de C compila un programa, se crea un archivo `.obj`. Sin embargo, todavía se necesita un enlazador para enlazar todos los archivos `.obj` con otros archivos de biblioteca a fin de formar el archivo ejecutable final.
3. No, la función `exit()` no devuelve ningún valor. Sin embargo, la instrucción `return` sí lo hace. En la función `main()`, si la instrucción `return` devuelve un valor `0`, esto le indica al sistema operativo que el programa terminó en forma normal; en caso contrario, ocurre un error.
4. En C, un archivo requerido por la directiva `#include` y que termina con la extensión `.h` se denomina *archivo de encabezado*. Más adelante, en este libro, aprenderá que un archivo de encabezado contiene los datos o declaraciones de funciones.

Ejercicios

1. No. Los paréntesis angulares (< y >) de la expresión `#include <stdio.h>` le solicitan al preprocesador de C que busque el archivo de encabezado en un directorio distinto al actual. Por otra parte, la expresión `#include "stdio.h"` le indica al preprocesador de C que primero busque el archivo de encabezado `stdio.h` en el directorio actual, y que después lo busque en otro directorio.
2. La siguiente es una posible solución:

```
/* 02A02.c */
#include <stdio.h>

main()
{
   printf ("Es divertido escribir mi propio programa de C.\n");
   return 0;
}
```

Materiał chroniony prawem autorskim

Hora 4, "Tipos de datos y palabras reservadas"

Cuestionario

1. Sí. Tanto `134/100` como `17/10` dan el mismo resultado, es decir, `1`.
2. Sí. Los resultados de `3000 + 1.0` y de `3000/1.0` son valores de punto flotante.
3. Tenemos las siguientes expresiones en la notación científica:
   - `3.5e3`
   - `3.5e-3`
   - `-3.5e-3`
4. Entre los cuatro nombres, `7o_Calculo` y `Método_de_Tom` no son nombres válidos en C.

Ejercicios

1. La siguiente es una posible solución:

```
/* 04A01.c */
#include <stdio.h>

main()
{
   char c1;
   char c2;

   c1 = 'Z';
   c2 = 'z';
   printf("El valor numérico de Z es: %d.\n", c1);
   printf("El valor numérico de z es: %d.\n", c2);
   return 0;
}
```

SALIDA Los resultados del programa son:

```
El valor numérico de Z es: 90.
El valor numérico de z es: 122.
```

2. La siguiente es una posible solución:

```
/* 04A02.c */
#include <stdio.h>

main()
{
   char c1;
```

Materiał chroniony prawem autorskim


```
    char c2;

    c1 = 72;
    c2 = 104;
    printf("El carácter de 72 es: %c\n", c1);
    printf("El carácter de 104 es: %c\n", c2);
    return 0;
}
```

SALIDA Los resultados del programa son:

```
El carácter de 72 es: H
El carácter de 104 es: h
```

3. El número 72368 está fuera del rango del tipo de datos `int` de 16 bits. Si asigna un valor demasiado grande para el tipo de datos, el valor resultante se redondeará y el resultado será incorrecto.

4. La siguiente es una posible solución:

```
/* 04A04.c */
#include <stdio.h>

main()
{
    double num_dbl;

    num_dbl = 123.456;
    printf("El formato de punto flotante de 123.456 es: %f\n",
           num_dbl);
    printf("El formato de notación científica de 123.456 es: %e\n",
           num_dbl);

    return 0;
}
```

SALIDA En mi máquina, los resultados del programa son los siguientes:

```
El formato de punto flotante de 123.456 es: 123.456000
El formato de notación científica de 123.456 es: 1.234560e+002
```

5. La siguiente es una posible solución:

```
/* 04A05.c */
#include <stdio.h>

main()
{
    char ch;

    ch = '\n';
    printf("El valor numérico del carácter de nueva línea es: %d\n", ch);

    return 0;
}
```

```
    putchar(ch);

    return 0;
}
```

5. Es probable que obtenga dos mensajes de error (advertencias); uno que indique que `getchar()` está indefinida y otro que diga que `putchar()` está indefinida. La razón es que el archivo de encabezado `stdio.h` se omitió en el código.

Hora 6, "Manejo de datos"

Cuestionario

1. El operador `=` es un operador de asignación que asigna el valor del operando que está a la derecha del operador, al operando que está a su izquierda. Por otra parte, `==` es uno de los operadores relacionales; este operador sólo verifica los valores de los dos operandos que están a ambos lados y determina si son iguales.
2. En la expresión `x + - y - - z`, el primero y el tercero de los signos de menos son operadores de negación; el segundo signo de menos es un operador de resta.
3. `15/4` se evalúa como `3`. `(float)15/4` se evalúa como `3.750000`.
4. No. De hecho, la expresión `y *= x + 5` es igual a la expresión `y = y * (x + 5)`.

Ejercicios

1. La siguiente es una posible solución:

```
/* 06A01.c */
#include <stdio.h>

main()
{
    int x, y;

    x = 1;
    y = 3;
    x += y;
    printf("El resultado de x += y es: %d\n", x);

    x = 1;
    y = 3;
    x += -y;
    printf("El resultado de x += -y es: %d\n", x);

    x = 1;
    y = 3;
```

```
    x -= y;
    printf("El resultado de x -= y es: %d\n", x);

    x = 1;
    y = 3;
    x -= -y;
    printf("El resultado de x -= -y es: %d\n", x);

    x = 1;
    y = 3;
    x *= y;
    printf("El resultado de x *= y es: %d\n", x);

    x = 1;
    y = 3;
    x *= -y;
    printf("El resultado de x *= -y es: %d\n", x);

    return 0;
}
```

SALIDA Los resultados del programa son:

```
El resultado de x += y es: 4
El resultado de x += -y es: -2
El resultado de x -= y es: -2
El resultado de x -= -y es: 4
El resultado de x *= y es: 3
El resultado de x *= -y es: -3
```

2. Después de evaluar la expresión `z=x*y==18`, el valor de `z` es `1` (uno).
3. La siguiente es una posible solución:

```
/* 06A03.c */
#include <stdio.h>

main()
{
    int x;

    x = 1;
    printf("x++ produce:    %d\n", x++);
    printf("Ahora x contiene: %d\n", x);

    return 0;
}
```

SALIDA Los resultados del programa son:

```
x++ produce:    1
Ahora x contiene: 2
```

Materiał chroniony prawem autorskim

4. La siguiente es una posible solución:

```
/* 06A04.c */
#include <stdio.h>

main()
{
   int x;

   x = 1;
   printf("x = x++ produce: %d\n", x = x++);
   printf("Ahora x contiene:   %d\n", x);

   return 0;
}
```

En las dos llamadas `printf()` de este programa obtuve 1 y 1. La razón es que, en la expresión `x = x++`, el valor original de x se copia primero en una ubicación temporal, y luego x se incrementa en 1. Por último, el valor guardado en la ubicación temporal se asigna de nuevo a x. Es por eso que el valor final almacenado en x sigue siendo 1.

5. El programa usa en forma incorrecta un operador de asignación =, en vez de un operador relacional "igual a" (==).

Hora 7, "Ciclos"

Cuestionario

1. No.
2. Sí. El ciclo `do-while` imprime el carácter `d`, cuyo valor numérico es `100`.
3. Sí. Ambos ciclos iteran 8 veces.
4. Sí.

Ejercicios

1. El primer ciclo contiene una instrucción

```
printf("%d  +  %d  =  %d\n", i, j, i+j);
```

Pero el segundo ciclo `for` tiene un punto y coma justo después de la instrucción `for`. Ésta es una instrucción nula (solamente un punto y coma), lo cual significa que es una instrucción que no hace nada.

2. La siguiente es una posible solución:

```
/* 07A02.c */
#include <stdio.h>

main()
{
```

Materiał chroniony prawem autorskim


```
   int i, j;

   for (i=0, j=1; i<8; i++, j++)
      printf("%d  +  %d  =  %d\n", i, j, i+j);

   printf("\n");
   for (i=0, j=1; i<8; i++, j++);
      printf("%d  +  %d  =  %d\n", i, j, i+j);

   return 0;
}
```

SALIDA Los resultados del programa son:

```
0 + 1 = 1
1 + 2 = 3
2 + 3 = 5
3 + 4 = 7
4 + 5 = 9
5 + 6 = 11
6 + 7 = 13
7 + 8 = 15
8 + 9 = 17
```

3. La siguiente es una posible solución:

```
/* 07A03.c */
#include <stdio.h>

main()
{
   int c;

   printf("Introduzca un carácter:\n(introduzca K para terminar)\n");
   c = ' ';

   while( c != 'K' ) {
      c = getc(stdin);
      putchar(c);
   }
   printf("\nFuera del ciclo for. ¡Adiós!\n");

   return 0;
}
```

4. La siguiente es una posible solución:

```
/* 07A04.c: Uso de un ciclo for */
#include <stdio.h>

main()
{
   int i;

   i = 65;
   for (i=65; i<72; i++){
```

```
        printf("El valor numérico de %c es %d.\n", i, i);
    }

    return 0;
}
```

5. La siguiente es una posible solución:

```
/* 07A05.c */
#include <stdio.h>

main()
{
    int i, j;

    i = 1;
    while (i<=3) {    /* ciclo externo */
        printf("Inicio de la iteración %d del ciclo externo.\n", i);
        j = 1;
        do{    /* ciclo interno */
            printf("    Iteración %d del ciclo interno.\n", j);
            j++;
        } while (j<=4);
        i++;
        printf("Fin de la iteración %d del ciclo externo.\n", i);
    }

    return 0;
}
```

Hora 8, "Uso de operadores condicionales"

Cuestionario

1. La expresión `(x=1)&&(y=10)` devuelve `1`; `(x=1)&(y=10)` devuelve `0`.
2. En la expresión `!y ? x == z : y`, `!y` produce `0`; de este modo, el valor del tercer operando y se toma como el valor de la expresión. Es decir, la expresión se evalúa como `1`.
3. `1100111111000110` y `0011000000111001`.
4. La expresión `(x%2==0)||(x%3==0)` produce `1`, y la expresión `(x%2==0)&&(x%3==0)` se evalúa como `0`.
5. Sí. `8 >> 3` equivale a $8/2^3$. Por otra parte, `1 << 3` equivale a $2^3= 8$.

Ejercicios

1. -x produce `0x1000` debido a que `-0xEFFF` equivale a `-1110111111111111` (en binario), lo cual produce `0001000000000000` (en binario), (es decir, `0x1000` en formato hexadecimal). De la misma manera, ~y se evalúa como `0xEFFF` debido a que `~0x1000` equivale a `~0001000000000000` (en binario), lo cual produce `1110111111111111` (en binario) (es decir, `0xEFFF` en formato hexadecimal).

Material chroniony prawem autorskim

2. La siguiente es una posible solución:

```
/* 08A02.c */
#include <stdio.h>

int main()
{
   int x, y;

   x = 0xEFFF;
   y = 0x1000;

   printf("!x produce: %d (es decir, %u)\n", !x, !x);
   printf("!y produce: %d (es decir, %u)\n", !y, !y);

   return 0;
}
```

Los resultados del programa son:

```
!x produce: 0 (es decir, 0)
!y produce: 0 (es decir, 0)
```

3. La siguiente es una posible solución:

```
/* 08A03.c */
#include <stdio.h>

int main()
{
   int x, y;

   x = 123;
   y = 4;
   printf("x << y produce: %d\n", x << y);
   printf("x >> y produce: %d\n", x >> y);

   return 0;
}
```

SALIDA Los resultados del programa son:

```
x << y produce: 1968
x >> y produce: 7
```

4. La siguiente es una posible solución:

```
/* 08A04.c */
#include <stdio.h>

int main()
{
   printf("0xFFFF ^ 0x8888 produce: 0x%X\n",
```

```
            0xFFFF ^ 0x8888);
    printf("0xABCD & 0x4567 produce: 0x%X\n",
            0xABCD & 0x4567);
    printf("0xDCBA ¦ 0x1234 produce: 0x%X\n",
            0xDCBA ¦ 0x1234);

    return 0;
}
```

SALIDA Los resultados del programa son:

```
0xFFFF ^ 0x8888 produce: 0x7777
0xABCD & 0x4567 produce: 0x145
0xDCBA ¦ 0x1234 produce: 0xDEBE
```

5. La siguiente es una posible solución:

```
/* 08A05.c */
#include <stdio.h>

main()
{
    int x;

    printf("Introduzca un carácter:\n(introduzca q para terminar)\n");
    for ( x=' '; x != 'q' ? 1 : 0; ) {
        x = getc(stdin);
        putchar(x);
    }
    printf("\nFuera del ciclo for. ¡Adiós!\n");

    return 0;
}
```

Hora 9, "Modificadores de datos y funciones matemáticas"

Cuestionario

1. No. x contiene un número negativo que no es igual al número contenido por la variable y de tipo `unsigned int`.
2. Puede usar el modificador `long`, el cual incrementa el rango de valores que puede contener `int`. Por otro lado, si almacena un número positivo, también puede usar el modificador `unsigned` para almacenar un valor más grande.
3. `%lu`.
4. El archivo de encabezado es `math.h`.

Materiał chroniony prawem autorskim

Ejercicios

1. La siguiente es una posible solución:

```
/* 09A01 */
#include <stdio.h>

main()
{
   int x;
   unsigned int y;

   x = 0xAB78;
   y = 0xAB78;

   printf("El valor decimal de x es %d.\n", x);
   printf("El valor decimal de y es %u.\n", y);

   return 0;
}
```

SALIDA En mi máquina, los resultados del programa son:

```
El valor decimal de x es -21640.
El valor decimal de y es 43896.
```

2. La siguiente es una posible solución:

```
/* 09A02 */
#include <stdio.h>

main()
{
   printf("El tamaño de short int es %d.\n",
          sizeof(short int));
   printf("El tamaño de long int es %d.\n",
          sizeof(long int));
   printf("El tamaño de long double es %d.\n",
          sizeof(long double));

   return 0;
}
```

3. La siguiente es una posible solución:

```
/* 09A03 */
#include <stdio.h>

main()
{
   int x, y;
   long int resultado;

   x = 7000;
```

Materiał chroniony prawem autorskim


```
main()
{
   double x;

   x = (double)0x19A1;
   printf("La raíz cuadrada de x es: %2.0f\n", sqrt(x));

   return 0;
}
```

Hora 10, "Control de flujo del programa"

Cuestionario

1. No.
2. Después de la ejecución de los tres casos, '-', '*' y '/', el resultado final almacenado en x es 2.
3. El resultado final almacenado en x es 2. Esta vez sólo se ejecuta el caso del operador = '-' debido a la instrucción break.
4. El resultado almacenado en x es 27.

Ejercicios

1. La siguiente es una posible solución:

```
/* 10A01.c  Uso de la instrucción if */
#include <stdio.h>

main()
{
   int i;

   printf("Enteros que se pueden dividir entre 2 y entre 3\n");
   printf("(dentro del rango de 0 a 100):\n");
   for (i=0; i<=100; i++)
      if (i%6 == 0)
         printf("   %d\n", i);

   return 0;
}
```

2. La siguiente es una posible solución:

```
/* 10A02.c  Uso de la instrucción if */
#include <stdio.h>
```

```
main()
{
   int i;

   printf("Enteros que se pueden dividir entre 2 y entre 3\n");
   printf("(dentro del rango de 0 a 100):\n");
   for (i=0; i<=100; i++)
      if (i%2 == 0)
        if (i%3 == 0)
          printf("   %d\n", i);

   return 0;
}
```

3. La siguiente es una posible solución:

```
/* 10A03.c */
#include <stdio.h>

main()
{
   int letra;

   printf("Introduzca una letra:\n");
   letra = getchar();
   switch (letra){
      case 'A':
         printf("El valor numérico de A es: %d\n", 'A');
         break;
      case 'B':
         printf("El valor numérico de B es: %d\n", 'B');
         break;
      case 'C':
         printf("El valor numérico de C es: %d\n", 'C');
         break;
      default:
         break;
   }

   return 0;
}
```

4. La siguiente es una posible solución:

```
/* 10A04.c */
#include <stdio.h>

main()
{
   int c;
```

Materiał chroniony prawem autorskim


```
    printf("Introduzca un carácter:\n(introduzca q para terminar)\n");
    while ((c = getc(stdin)) != 'q') {

       /* no hay instrucciones dentro del ciclo while */
    }
    printf("\n¡Adiós!\n");

    return 0;
}
```

5. La siguiente es una posible solución:

```
/* 10A05.c */
#include <stdio.h>

main()
{
    int i, suma;

    suma = 0;
    for (i=1; i<8; i++){
        if ((i%2 == 0) && (i%3 == 0))
            continue;
        suma += i;
    }
    printf("La suma es: %d\n", suma);
    return 0;
}
```

Hora 11, "Apuntadores"

Cuestionario

1. Utilizando el operador de dirección, `&`. Es decir, la expresión `&ch` da el valor izquierdo (la dirección) de la variable de carácter `ch`.
2. Las respuestas son las siguientes:
    - Operador de indirección o de desreferenciación
    - Operador de multiplicación
    - Operador de multiplicación
    - El primero y el tercer asteriscos son operadores de indirección; el segundo asterisco es un operador de multiplicación.
3. `aptr_int` produce el valor de la dirección `0x1A38`; `*aptr_int` produce el valor `10`.
4. `x` contiene ahora el valor `456`.

Ejercicios

1. La siguiente es una posible solución:

```
/* 11A01.c */
#include <stdio.h>

main()
{
   int x, y, z;

   x = 512;
   y = 1024;
   z = 2048;

   printf("Los valores izquierdos de x, y y z son:\n");
   printf("0x%p,  0x%p,  0x%p\n", &x, &y, &z);
   printf("Los valores derechos de x, y y z son:\n");
   printf("%d,  %d,  %d\n", x, y, z);

   return 0;
}
```

2. La siguiente es una posible solución:

```
/* 11A02.c */
#include <stdio.h>

main()
{
   int *aptr_int;
   char *aptr_ch;

   aptr_int = 0; /* apuntador nulo */
   aptr_ch = 0;  /* apuntador nulo */

   printf("El valor izquierdo de aptr_int es: 0x%p\n",
           aptr_int);
   printf("El valor derecho de aptr_int es: %d\n",
           *aptr_int);
   printf("El valor izquierdo de aptr_ch es: 0x%p\n",
           aptr_ch);
   printf("El valor derecho de aptr_ch es: %d\n",
           *aptr_ch);

   return 0;
}
```

3. La siguiente es una posible solución:

```
/* 11A03.c */
#include <stdio.h>
```

Materiał chroniony prawem autorskim


```
main()
{
   char ch;
   char *aptr_ch;

   ch = 'A';
   printf("El valor derecho de ch es: %c\n",
           ch);
   aptr_ch = &ch;
   *aptr_ch = 'B';  /* decimal 66 */
   /* demuestra que se actualizó ch */
   printf("El valor izquierdo de ch es: 0x%p\n",
           &ch);
   printf("El valor derecho de aptr_ch es: 0x%p\n",
           aptr_ch);
   printf("El valor derecho de ch es: %c\n",
           ch);

   return 0;
}
```

4. La siguiente es una posible solución:

```
/* 11A04.c */
#include <stdio.h>

main()
{
   int x, y;
   int *aptr_x, *aptr_y;

   x = 5;
   y = 6;
   aptr_x = &x;
   aptr_y = &y;

   *aptr_x *= *aptr_y;

   printf("El resultado es: %d\n",
           *aptr_x);

   return 0;
}
```

Hora 12, "Arreglos"

Cuestionario

1. Declara un arreglo de tipo `int` denominado `arreglo_ent` con cuatro elementos. Esta instrucción también inicializa el arreglo con cuatro enteros, 12, 23, 9 y 56.

Materiał chroniony prawem autorskim

Ejercicios

1. La siguiente es una posible solución:

```
/* 13A01.c: Copia una cadena a otra */
#include <stdio.h>
#include <string.h>

main()
{
   int i;
   char cadena1[] = "Éste es el ejercicio 1.";
   char cadena2[24];

   /* Método I */
   strcpy(cadena2, cadena1);

   /* confirma el copiado */
   printf("del Método I: %s\n", cadena2);

   /* Método II */
   for (i=0; cadena1[i]; i++)
      cadena2[i] = cadena1[i];
   cadena2[i] = '\0';
   /* confirma el copiado */
   printf("del método II: %s\n", cadena2);

   return 0;
}
```

2. La siguiente es una posible solución:

```
/* 13A02.c: Cómo medir la longitud de una cadena */
#include <stdio.h>
#include <string.h>

main()
{
   int i, longitud_cad;
   char cadena[] = "Éste es el ejercicio 2.";

   /* Método I */
   longitud_cad = 0;
   for (i=0; cadena[i]; i++)
      longitud_cad++;
   printf("La longitud de la cadena es %d.\n", longitud_cad);

   /* Método II */
   printf("La longitud de la cadena es %d.\n",
          strlen(cadena);

   return 0;
}
```

3. La siguiente es una posible solución:

```
/* 13A03.c: Uso de gets() y puts() */
#include <stdio.h>

main()
{
   char cadena[80];
   int i, delt;

   printf("Introduzca una cadena de menos de 80 caracteres:\n");
   gets( cadena);
   delt = 'a' - 'A';
   i = 0;
   while (cadena[i]){
     if ((cadena[i] >= 'A') && (cadena[i] <= 'Z'))
        cadena[i] += delt;  /* convierte a minúsculas */
     ++i;
   }
   printf("La cadena introducida (en minúsculas) es:\n");
   puts(cadena);

   return 0;
}
```

4. La siguiente es una posible solución:

```
/* 13A04.c: Uso de scanf() */
#include <stdio.h>

main()
{
   int x, y, suma;

   printf("Introduzca dos enteros:\n");
   scanf("%d%d", &x, &y);
   suma = x + y;
   printf("La suma es %d\n", suma);

   return 0;
}
```

Hora 14, "Alcance y clases de almacenamiento"

Cuestionario

1. La variable `x` de tipo `int` y la variable `y` de tipo `float`, declaradas fuera de la función `miFuncion()`, son variables globales. Las variables `int`, `i` y `j`, y la variable `y`

Materiał chroniony prawem autorskim

Hora 15, "Funciones"

Cuestionario

1. Las respuestas son las siguientes:
   - `int funcion_1(int x, float y);` es una declaración de función con un número fijo de argumentos.
   - `void funcion_2(char *cadena);` es una declaración de función con un número fijo de argumentos.
   - `char *asctime(const struct tm *timeptr);` es una declaración de función con un número fijo de argumentos.
   - `int funcion_3(void);` es una declaración de función sin argumentos.
   - `char funcion_4(char c, ...);` es una declaración de función con un número variable de argumentos.
   - `void funcion_5(void);` es una declaración de función sin argumentos.
2. La segunda expresión es una definición de función; es decir,

   `int funcion_2(int x, int y) {return x+y;}`
3. El tipo de datos `int` es el tipo predeterminado devuelto por una función si se omite el especificador de tipo.
4. La tercera, `char función_3(...);`, es ilegal.

Ejercicios

1. La siguiente es una posible solución:

```
/* 15A01.c: */
#include <stdio.h>
#include <time.h>

void FechaHora(void);

main()
{
   printf("Antes de llamar a la función FechaHora().\n");
   FechaHora();
   printf("Después de llamar a la función FechaHora().\n");
   return 0;
}
/* Definición de FechaHora()*/
void FechaHora(void)
{
   time_t ahora;
   int i;
```

```
    char *cadena;

    printf("Dentro de FechaHora().\n");
    time(&ahora);
    cadena = asctime(localtime(&ahora));
    printf("La fecha y hora actuales son: ");
    for (i=0; cadena[i]; i++)
       printf("%c", cadena[i]);
}
```

2. La siguiente es una posible solución:

```
/* 15A02.c */
#include <stdio.h>

int MultiDos(int x, int y);

main ()
{
    printf("El resultado que devuelve MultiDos() es: %d\n",
       MultiDos(32, 10));
    return 0;
}

/* definición de función */
int MultiDos(int x, int y)
{
    return x * y;
}
```

3. La siguiente es una posible solución:

```
/* 15A03.c */
#include <stdio.h>
#include <stdarg.h>

int MultiEnt(int x, ...);

main ()
{
    int d1 = 1;
    int d2 = 2;
    int d3 = 3;
    int d4 = 4;

    printf("Dado un argumento: %d\n", d1);
    printf("El resultado que devuelve MultiEnt() es: %d\n\n",
       MultiEnt(1, d1));
    printf("Dados los argumentos: %d, %d, %d, y %d\n", d1, d2, d3, d4);
    printf("El resultado que devuelve MultiEnt() es: %d\n\n",
       MultiEnt(4, d1, d2, d3, d4));

    return 0;
```

```
double SumaDobles(int x, ...)
{
    va_list   listaarg;
    int i;
    double argumento, resultado = 0.0;

    printf("El número de argumentos es: %d\n", x);
    va_start (listaarg, x);
    for (i=0; i<x; i++){
       argumento = va_arg(listaarg, double);
       printf("Argumento pasado a esta función: %f\n", argumento);
       resultado += argumento;
    }

    va_end (listaarg);

    return resultado;
}
```

Hora 16, "Uso de apuntadores"

Cuestionario

1. En mi máquina obtuve las siguientes respuestas:
   - 4 bytes
   - 4 bytes
   - 4 bytes
   - 12 bytes
   - 12 bytes
   - 12 bytes
2. Debido a que `0x100A - 0x1006` da 4, y un `int` ocupa 2 bytes, `aptr1` y `aptr2` son dos enteros diferentes. Por lo tanto, la respuesta es 2.
3. `0x0230` y `0x0260`.
4. Las respuestas son las siguientes:
   - `*(aptr + 3)` extrae `'A'`.
   - `aptr - ch` da 1.
   - `*(aptr - 1)` extrae `'a'`.
   - `*aptr = 'F'` reemplaza `'b'` con `'F'`.

```
    int *aptr_int;
    int i, suma;
    int max = 0;
    int terminacion = 0;

    printf("Escriba el número total de enteros:\n");
       scanf("%d", &max);
    /* llama a malloc() */
    aptr_int = malloc(max * sizeof(int));
    if (aptr_int == NULL){
      printf("Falló la función malloc().\n");
      terminacion = 1;
    }
    else{
      for (i=0; i<max; i++)
        aptr_int[i] = i + 1;
    }
    suma = 0;
    for (i=0; i<max; i++)
      suma += aptr_int[i];
    printf("La suma es %d.\n", suma);
    free(aptr_int);

    return terminacion;
}
```

2. La siguiente es una posible solución:

```
/* 17A02.c */
#include <stdio.h>
#include <stdlib.h>

/* función main() */
main()
{
    float *aptr_flt;
    int terminacion = 0;

    /* llama a calloc() */
    aptr_flt = calloc(100, sizeof(float));
    if (aptr_flt == NULL){
      printf("Falló la función calloc().\n");
      terminacion = 1;
    }
    else{
      aptr_flt = realloc(aptr_flt, 150 * sizeof(float));
      if (aptr_flt == NULL){
        printf("Falló la función realloc().\n");
        terminacion = 1;
      }
```

Materiał chroniony prawem autorskim


```
      else
        free(aptr_flt);
   }
   printf("¡Concluido!\n");

   return terminacion;
}
```

3. La siguiente es una posible solución:

```
/* 17A03.c */
#include <stdio.h>
#include <stdlib.h>

/* función main() */
main()
{
   float *aptr1, *aptr2;
   int i;
   int terminacion = 1;
   int max = 0;

   printf("Escriba el número total:\n");
      scanf("%d", &max);

   aptr1 = malloc(max * sizeof(float));
   aptr2 = calloc(max, sizeof(float));

   if (aptr1 == NULL)
      printf("Falló malloc().\n");
   else if (aptr2 == NULL)
      printf("Falló calloc().\n");
   else{
      for (i=0; i<max; i++)
         printf("aptr1[%d]=%5.2f,  aptr2[%d]=%5.2f\n",
          i, *(aptr1 + i), i, *(aptr2 + i));
      free(aptr1);
      free(aptr2);
      terminacion = 0;
   }
   printf ("\n¡Adiós!\n");

   return terminacion;
}
```

4. La siguiente es una posible solución:

```
/* 17A04.c: Uso de la función realloc() */
#include <stdio.h>
#include <stdlib.h>
```

```
main(void)
{
   typedef char WORD;
   typedef int SHORT;
   typedef long LONG;
   typedef float FLOAT;
   typedef double DFLOAT;

   printf("El tamaño de WORD es: %d-byte\n", sizeof(WORD));
   printf("El tamaño de SHORT es: %d-byte\n", sizeof(SHORT));
   printf("El tamaño de LONG es: %d-byte\n", sizeof(LONG));
   printf("El tamaño de FLOAT es: %d-byte\n", sizeof(FLOAT));
   printf("El tamaño de DFLOAT es: %d-byte\n", sizeof(DFLOAT));

   return 0;
}
```

3. La siguiente es una posible solución:

```
/* 18A03.c */
#include <stdio.h>

enum con{MIN_NUM = 0,
         MAX_NUM = 100};

int fRecur(int n);

main()
{
   int i, suma1, suma2;

   suma1 = suma2 = 0;
   for (i=1; i<=MAX_NUM; i++)
     suma1 += i;
   printf("El valor de suma1 es %d.\n", suma1);
   suma2 = fRecur(MIN_NUM);
   printf("El valor que devuelve fRecur() es %d.\n", suma2);

   return 0;
}

int fRecur(int n)
{
   if (n > MAX_NUM)
     return 0;
   return  fRecur(n + 1) + n;
}
```

4. La siguiente es una posible solución:

```
/* 18A04.c: Argumentos de la línea de comandos */
#include <stdio.h>

main (int argc, char *argv[])
```

```
            200,
            2345.67};

    printf("año: %d\n", sedan.anio);
    printf("modelo: %s\n", sedan.modelo);
    printf("potencia_motor: %d\n", sedan.potencia_motor);
    printf("peso: %6.2f\n", sedan.peso);

    return 0;
}
```

2. La siguiente es una posible solución:

```
/* 19A02.c */
#include <stdio.h>

struct empleado {
    int id;
    char nombre[32];
};

void Exhibe(struct empleado s);

main(void)
{
    /* inicialización de la estructura */
    struct empleado info = {
        0001,
        "B. Smith"
        };

    printf("ésta es una muestra:\n");
    Exhibe(info);

    printf("¿Cuál es su nombre?\n");
        gets(info.nombre);
    printf("¿Cuál es su número de ID?\n");
        scanf("%d", &info.id);

    printf("\nEsto es lo que introdujo:\n");
    Exhibe(info);

    return 0;
}
/* definición de función */
void Exhibe(struct empleado s)
{
    printf("Nombre: %s\n", s.nombre);
    printf("No. de ID #: %04d\n\n", s.id);
}
```

```
   char autor[16];
   char cadena1[32];
   char cadena2[32];
   char cadena3[32];
};

typedef struct haiku HK;

void ExhibeDatos(HK *aptr_s);

main(void)
{
   HK poema[2] = {
     { 1641,
       1716,
       "Sodo",
       "Llevándome con ella",
       "mi sombra regresa a casa",
       "tras mirar la luna."
     },
     { 1729,
       1781,
       "Chora",
       "Sopla un viento de tormenta",
       "de entre las hierbas",
       "crece la luna llena."
     }
   };

   /* define un arreglo de apuntadores con HK */
   HK *aptr_poema[2] = {&poema[0], &poema[1]};
   int i;

   for (i=0; i<2; i++)
      ExhibeDatos(aptr_poema[i]);

   return 0;
}

void ExhibeDatos(HK *aptr_s)
{
   printf("%s\n", aptr_s->cadena1);
   printf("%s\n", aptr_s->cadena2);
   printf("%s\n", aptr_s->cadena3);
   printf("--- %s\n", aptr_s->autor);
   printf("    (%d-%d)\n\n", aptr_s->nacimiento, aptr_s->muerte);
}
```

Materiał chroniony prawem autorskim

Hora 20, "Uniones"

Cuestionario

1. La primera instrucción es la declaración de una unión con el nombre de etiqueta `una_union`. La segunda instrucción define dos variables de unión, `x` y `y`, con el tipo de datos `una_union`.
2. Falta el punto y coma (`;`) en dos lugares: al final de la declaración de `char modelo[8]` y al final de la declaración de la unión.
3. Los dos miembros de la unión tienen el mismo valor, `1997`.

Ejercicios

1. La siguiente es la versión modificada. El contenido del arreglo `nombre` se sobreescribe parcialmente con el valor asignado a la variable `double`, `precio`.

```
/* 20A01.c */
#include <stdio.h>
#include <string.h>

main(void)
{
   union menu {
      char nombre[23];
      double precio;
   } platillo;

   printf("Contenido asignado a la unión por separado:\n");
   /* acceso al nombre */
   strcpy(platillo.nombre, "Pollo agridulce");
   /* acceso al precio */
   platillo.precio = 9.95;
   printf("Nombre del platillo: %s\n", platillo.nombre);
   printf("Precio del platillo: %5.2f\n", platillo.precio);

   return 0;
}
```

2. La siguiente es una posible solución:

```
/* 20A02.c */
#include <stdio.h>

union empleado {
   int anio_inicial;
   int cve_depto;
   int num_id;
};

void ExhibeDatos(union empleado u);
```

```
Esto es lo que usted introdujo:
Su nombre es Antonio.
Usted es de Tlaxcala.

¡Gracias!
```

4. La siguiente es una posible solución:

```
/* 20A04.c */
#include <stdio.h>
#include <string.h>

struct campo_bits {
   int si: 1;
};

struct encuesta {
   struct campo_bits bandera;
   char nombre[20];
   union {
       char estado[32];
       char pais[32];
   } lugar;
};

void CapturaDatos(struct encuesta *s);
void ExhibeDatos(struct encuesta *s);

main(void)
{
   struct encuesta ciudadano;

   CapturaDatos(&ciudadano);
   ExhibeDatos(&ciudadano);

   return 0;
}

/* definición de función */
void CapturaDatos(struct encuesta *aptr)
{
   char respuesta[4];

   printf("Escriba su nombre:\n");
      gets(aptr->nombre);

   printf("¿Es usted ciudadano de México? (Sí o No)\n");
      gets(respuesta);
   if ((respuesta[0] == 'S') ||
       (respuesta[0] == 's')){
      printf("Escriba el nombre del estado:\n");
      gets(aptr->lugar.estado);
```

```
enum {EXITO, FRACASO};

int LeeCar(FILE *archEnt);

main(void)
{
   FILE *aptrf;
   char nomarchivo[]= "haiku.txt";
   int valdevol = EXITO;

   if ((aptrf = fopen(nomarchivo, "r")) == NULL){
      printf("No es posible abrir %s.\n", nomarchivo);
      valdevol = FRACASO;
   } else {
      printf("\nEl número total de caracteres es %d.\n",
          LeeCar(aptrf));
      fclose(aptrf);
   }

   return valdevol;
}

/* definición de LeeCar() */
int LeeCar(FILE *archEnt)
{
   int c, num;

   num = 0;
   while ((c=fgetc(archEnt)) != EOF){
      putchar(c);
      ++num;
   }

   return num;
}
```

2. La siguiente es una posible solución:

```
/* 21A02.c */
#include <stdio.h>
#include <string.h>

enum {EXITO, FRACASO, MAX_LEN = 80};

void EscribeLinea(FILE *archSal, char *cadena);

main(void)
{
   FILE *aptrf;
   char cadena[MAX_LEN+1];
   char nomarchivo[32];
   int valdevol = EXITO;
```

```
/* definición de función */
void EscribeCar(FILE *archSal, char *cadena)
{
   int i, c;

   i = 0;
   while ((c=cadena[i]) != '\0'){
      putchar(c);
      fputc(c, archSal);
      i++;
   }
}
```

4. La siguiente es una posible solución:

```
/* 21A04.c */
#include <stdio.h>
#include <string.h>

enum {EXITO, FRACASO};

void EscribeBloque(FILE *archSal, char *cadena);

main(void)
{
   FILE *aptrf;
   char nomarchivo[]= "prueba_21.txt";
   char cadena[]= "La E/S de archivos en disco tiene sus trucos.";
   int valdevol = EXITO;

   if ((aptrf = fopen(nomarchivo, "w")) == NULL){
      printf("No es posible abrir %s.\n", nomarchivo);
      valdevol = FRACASO;
   } else {
      EscribeBloque(aptrf, cadena);
      fclose(aptrf);
   }

   return valdevol;
}

/* definición de función */
void EscribeBloque(FILE *archSal, char *cadena)
{
   int num;

   num = strlen(cadena);
   fwrite(cadena, sizeof(char), num, archSal);
   printf("%s\n", cadena);
}
```

Materiał chroniony prawem autorskim

Hora 22, "Funciones de archivo especiales"

Cuestionario

1. Sí. Las dos instrucciones son equivalentes.
2. No. Estas instrucciones no son equivalentes, a menos que el indicador de posición del archivo esté, efectivamente, al principio del archivo.
3. La función `scanf()` lee desde el archivo `prueba.txt`, en lugar de hacerlo desde el flujo de entrada predeterminado, debido a que la función `freopen()` redireccionó el flujo de entrada y lo asoció al archivo `prueba.txt`.
4. Si el tamaño del tipo de datos `double` es de ocho bytes de longitud, los cuatro elementos de datos `double` ocuparán en conjunto 32 bytes en el archivo binario.

Ejercicios

1. La siguiente es una posible solución:

```
/* 22A01.c */
#include <stdio.h>

enum {EXITO, FRACASO, MAX_LEN = 80};

void AptrSeek(FILE *aptrf);
long AptrTell(FILE *aptrf);
void LeeDatos(FILE *aptrf);
int MsjError(char *cadena);

main(void)
{
    FILE *aptrf;
    char nomarchivo[]= "LaoTzu.txt";
    int valdevol = EXITO;

    if ((aptrf = fopen(nomarchivo, "r")) == NULL){
       valdevol = MsjError(nomarchivo);
    } else {
       AptrSeek(aptrf);
       fclose(aptrf);
    }

    return valdevol;
}

/* definición de función */
void AptrSeek(FILE *aptrf)
{
    long desplaz1, desplaz2, desplaz3;
```

```
    desplaz1 = AptrTell(aptrf);
    LeeDatos(aptrf);
    desplaz2 = AptrTell(aptrf);
    LeeDatos(aptrf);
    desplaz3 = AptrTell(aptrf);
    LeeDatos(aptrf);
    printf("\nRelee el párrafo:\n");

    /* relee la tercera oración */
    fseek(aptrf, desplaz3, SEEK_SET);
    LeeDatos(aptrf);

    /* relee la segunda oración */
    fseek(aptrf, desplaz2, SEEK_SET);
    LeeDatos(aptrf);

    /* relee la primera oración */
    fseek(aptrf, desplaz1, SEEK_SET);
    LeeDatos(aptrf);
}

/* definición de función */
long AptrTell(FILE *aptrf)
{
    long valdevol;

    valdevol = ftell(aptrf);
    printf("El apuntador aptrf está en %ld\n", valdevol);

    return valdevol;
}

/* definición de función */
void LeeDatos(FILE *aptrf)
{
    char bufer[MAX_LEN];

    fgets(bufer, MAX_LEN, aptrf);
    printf("%s", bufer);
}

/* definición de función */
int MsjError(char *cadena)
{
    printf("No es posible abrir %s.\n", cadena);

    return FRACASO;
}
```

Materiał chroniony prawem autorskim

2. La siguiente es una posible solución:

```
/* 22A02.c */
#include <stdio.h>

enum {EXITO, FRACASO, MAX_LEN = 80};

void AptrSeek(FILE *aptrf);
long AptrTell(FILE *aptrf);
void LeeDatos(FILE *aptrf);
int MsjError(char *cadena);

main(void)
{
     FILE *aptrf;
     char nomarchivo[]= "LaoTzu.txt";
     int valdevol = EXITO;

     if ((aptrf = fopen(nomarchivo, "r")) == NULL){
        valdevol = MsjError(nomarchivo);
     } else {
        AptrSeek(aptrf);
        fclose(aptrf);
     }

     return valdevol;
}

/* definición de función */
void AptrSeek(FILE *aptrf)
{
     long desplaz1, desplaz2, desplaz3;

     desplaz1 = AptrTell(aptrf);
     LeeDatos(aptrf);
     desplaz2 = AptrTell(aptrf);
     LeeDatos(aptrf);
     desplaz3 = AptrTell(aptrf);
     LeeDatos(aptrf);
     printf("\nRelee el párrafo:\n");
     /* relee la tercera oración */
     fseek(aptrf, desplaz3, SEEK_SET);
     LeeDatos(aptrf);
     /* relee la segunda oración */
     fseek(aptrf, desplaz2, SEEK_SET);
     LeeDatos(aptrf);
     /* relee la primera oración */
     rewind(aptrf);  /* restablece al inicio el indicador de posición del
                        archivo */
     LeeDatos(aptrf);
}
```

```
/* definición de función */
long AptrTell(FILE *aptrf)
{
    long valdevol;

    valdevol = ftell(aptrf);
    printf("El apuntador aptrf está en %ld\n", valdevol);

    return valdevol;
}

/* definición de función */
void LeeDatos(FILE *aptrf)
{
    char bufer[MAX_LEN];

    fgets(bufer, MAX_LEN, aptrf);
    printf("%s", bufer);
}

/* definición de función */
int MsjError(char *cadena)
{
    printf("No es posible abrir %s.\n", cadena);
    return FRACASO;
}
```

3. En mi máquina, el archivo binario `datos.bin` es de 10 bytes. La siguiente es una posible solución:

```
/* 22A03.c */
#include <stdio.h>

enum {EXITO, FRACASO};

void EscribeDatos(FILE *archSal);
void LeeDatos(FILE *archEnt);
int MsjError(char *cadena);

main(void)
{
   FILE *aptrf;
   char nomarchivo[]= "datos.bin";
   int valdevol = EXITO;

   if ((aptrf = fopen(nomarchivo, "wb+")) == NULL){
      valdevol = MsjError(nomarchivo);
   } else {
      EscribeDatos(aptrf);
      rewind(aptrf);
      LeeDatos(aptrf);
```

Materiał chroniony prawem autorskim


```
        fclose(aptrf);
    }

    return valdevol;
}

/* definición de función */
void EscribeDatos(FILE *archSal)
{
    double numd;
    int numi;

    numd = 123.45;
    numi = 10000;

    printf("%5.2f\n", numd);
    fwrite(&numd, sizeof(double), 1, archSal);
    printf("%d\n", numi);
    fwrite(&numi, sizeof(int), 1, archSal);
}

/* definición de función */
void LeeDatos(FILE *archEnt)
{
    double x;
    int y;

    printf("\nLee de nuevo desde el archivo binario:\n");
    fread(&x, sizeof(double), (size_t)1, archEnt);
    printf("%5.2f\n", x);
    fread(&y, sizeof(int), (size_t)1, archEnt);
    printf("%d\n", y);
}

/* definición de función */
int MsjError(char *cadena)
{
    printf("No es posible abrir %s.\n", cadena);
    return FRACASO;
}
```

4. La siguiente es una posible solución:

```
/* 22A04.c */
#include <stdio.h>

enum {EXITO, FRACASO,
      MAX_NUM = 3,
      LONG_CAD = 23};

void LeeDatos(FILE *archEnt);
int MsjError(char *cadena);
```

```
main(void)
{
   FILE *aptrf;
   char nomarchivo[]= "numycad.mix";
   int valdevol = EXITO;

   if ((aptrf = freopen(nomarchivo, "r", stdin)) == NULL){
      valdevol = MsjError(nomarchivo);
   } else {
      LeeDatos(aptrf);
      fclose(aptrf);
   }

   return valdevol;
}

/* definición de función */
void LeeDatos(FILE *archEnt)
{
   int i;
   int kilometros;
   char ciudades[LONG_CAD];

   printf("Los datos leídos son:\n");
   for (i=0; i<MAX_NUM; i++){
      scanf("%s%d", ciudades, &kilometros);
      printf("%-23s  %d\n", ciudades, kilometros);
   }
}

/* definición de función */
int MsjError(char *cadena)
{
   printf("No es posible abrir %s.\n", cadena);
   return FRACASO;
}
```

Hora 23, "Compilación: el preprocesador de C"

Cuestionario

1. No se debe incluir el punto y coma (`;`) en la definición de la macro, ya que ésta termina con un carácter de nueva línea, no con un punto y coma.
2. El valor `82` se asigna a `resultado` debido a la expresión de asignación `resultado = 1 + 9 * 9`.

3. Se imprime el mensaje `Bajo #else`.
4. Se imprime el mensaje `Bajo #ifdef`.

Ejercicios

1. La siguiente es una posible solución:

```
/* 23A01.c */
#include <stdio.h>
/* función main() */
main()
{
   #define humano       100
   #define animal       50
   #define computadora  51
   #define DOM          0
   #define LUN          1
   #define MAR          2
   #define MIE          3
   #define JUE          4
   #define VIE          5
   #define SAB          6


   printf("humano: %d,  animal: %d,  computadora: %d\n",
      humano, animal, computadora);
   printf("DOM: %d\n", DOM);
   printf("LUN: %d\n", LUN);
   printf("MAR: %d\n", MAR);
   printf("MIE: %d\n", MIE);
   printf("JUE: %d\n", JUE);
   printf("VIE: %d\n", VIE);
   printf("SAB: %d\n", SAB);

   return 0;
}
```

2. La siguiente es una posible solución:

```
/* 23A02.c */
#include <stdio.h>

#define MULTIPLICA(val1, val2)  ((val1) * (val2))
#define SIN_ERROR  0

main(void)
{
   int resultado;

   resultado = MULTIPLICA(2, 3);
```

Material chroniony prawem autorskim

```
   printf("MULTIPLICA(2, 3) produce el valor de %d.\n", resultado);

   return SIN_ERROR;
}
```

3. La siguiente es una posible solución:

```
/* 23A03.c */
#include <stdio.h>

#define MAYUSCULAS  0
#define SIN_ERROR   0

main(void)
{
   #if MAYUSCULAS
     printf("ESTA LÍNEA SE IMPRIME,\n");
     printf("PORQUE MAYUSCULAS ESTÁ DEFINIDA.\n");
   #elif MINUSCULAS
     printf("Esta línea se imprime,\n");
     printf("porque MINUSCULAS está definida.\n");
   #else
     printf("Esta línea se imprime,\n");
     printf("porque ni MAYUSCULAS ni MINUSCULAS están definidas.\n");
   #endif

   return SIN_ERROR;
}
```

4. La siguiente es una posible solución:

```
/* 23A04.c:   */
#include <stdio.h>

#define LENG_C    'C'
#define LENG_B    'B'
#define SIN_ERROR  0

main(void)
{
   #if LENG_C == 'C'
      #if LENG_B == 'B'
        #undef LENG_C
        #define LENG_C "Yo conozco el lenguaje C.\n"
        #undef LENG_B
        #define LENG_B "Además, conozco BASIC.\n"
        printf("%s%s", LENG_C, LENG_B);

     #else
        #undef LENG_C
        #define LENG_C "Yo sólo conozco el lenguaje C.\n"
        printf("%s", LENG_C);
```

Materiał chroniony prawem autorskim

ÍNDICE

Símbolos

& (ampersand)
- operador AND a nivel de bits, 131
- operador de dirección, 177-179

* (asterisco)
- apuntadores, 180
- determinación de significado, 182
- operador de indirección, 182
- operador de multiplicación, 182

\ (carácter de escape), 59

\0 (carácter nulo), 198

" (comillas dobles), 32-33, 59

' (comillas sencillas), 59

[] (corchetes), 190

% (especificador de formato), 79

{} (llaves), 45, 48
- instrucción if, 156
- instrucción if-else, 159

/* (marca de apertura de comentario), 29

// (marca de apertura de comentario), 30

*/ (marca de cierre de comentario), 29

:? (operador condicional), 135-136

= (operador de asignación), 92

/= (operador de asignación de división), 93

*= (operador de asignación de multiplicación), 93

%= (operador de asignación de residuo), 93

-= (operador de asignación de resta), 93

+= (operador de asignación de suma), 93

~ (operador de complemento a nivel de bits), 131

— (operador de decremento), 96-98

>> (operador de desplazamiento a la derecha), 133-135

<< (operador de desplazamiento a la izquierda), 133-135

-> (operador de flecha), uniones, 335, 351

++ (operador de incremento), 96-98

Materiał chroniony prawem aut

- (operador de negación), 95
. (operador de punto), uniones, 335-337, 351
% (operador de residuo), 43
- (operador de resta), 96
!= (operador diferente de), 98
== (operador igual a), 98
&& (operador lógico AND), 124-126
! (operador lógico NOT), 128-129
|| (operador lógico OR), 126-127
>= (operador mayor o igual a), 98
> (operador mayor que), 98
<= (operador menor o igual a), 98
< (operador menor que), 98
| (operador OR a nivel de bits), 131
^ (operador XOR a nivel de bits), 131
( ) (paréntesis)
  colocación alrededor de expresiones, 419
  instrucción if, 156
<> (paréntesis angulares), 32
; (punto y coma), 28
 # (signo de numeral), 392
  caracteres de nueva línea, 392
  undef, directiva, 393-394, 406, 434
  comparación con el compilador de C, 392-393, 405
  compilación condicional anidada, 402-404
  define, directiva, 393, 434
  definición de macros de tipo funciones, 394-396
  definiciones de macros, 400
  definiciones de macros anidadas, 396
  ejemplo de código, 394-396, 398-399, 401-402
  elif, directiva, 401-402, 434
  else, directiva, 399-402, 434
  endif, directiva, 397-402, 406, 416
  expresiones, 396
  expresiones aritméticas, 406
  if, directiva, 434
  ifdef, directiva, 397-399, 434
  ifndef, directiva, 397-399, 416
  sintaxis, 393, 397, 399
  macro, cuerpo, 392, 396
  macros, nombres, 392-393
  macros, sustitución de, 392-396
19L02.exe, archivo ejecutable, 318

A

acceso
  aleatorio
    archivos en disco, 374-377, 387
    código de ejemplo, 375-378
    función fseek(), 374-378
    función ftell(), 374-378
  archivos secuenciales en disco, 374
  arreglos, mediante apuntadores, 264-266
  elementos de arreglos, índices, 190
  secuencial, archivos en disco, 374
activación de optimizadores, 235
actualización de definiciones, 301
agotamiento de recursos de pila, 305
agrupamiento de variables con estructuras, 314

de encabezado, 32
ANSI, 439
assert.h, 439
ctype.h, 439
errno.h, 439
float.h, 439
limits.h, 439
locale.h, 440
math.h, 440
setjmp.h, 440
signal.h, 440
stdarg.h, 440
stddef.h, 301
stdio.h, 32
stdlib.h, 302
string.h, 302
time.h, 440
definición, 356
denominación, sensibilidad a mayúsculas y minúsculas, 32
ejecutables, 24, 34-35
estructura FILE, 357, 374-375, 378
función fclose()
cierre de, 358-360, 371
ejemplo de código, 359-360
sintaxis, 359
función feof(), 367-369
función fgetc()
ejemplo de código, 361-363
lectura de, 360-363
sintaxis, 361
función fgets()
comparación con la función gets(), 364-366, 371
ejemplo de código, 364-366
lectura de archivos, 363-366
lectura de entrada desde el teclado, 366
sintaxis, 363
función fopen(), 357, 381
apertura de, 358
ejemplo de código, 359-360
formateo, 388
modos, 357-358, 388
sintaxis, 357
función fputc(), 360-363
función fputs(), 364-366
función fread(), 381
ejemplo de código, 368-369
lectura de, 366-369
sintaxis, 366
función fseek(), 374-378
función ftell(), 374-378
función fwrite(), 381
ejemplo de código, 368-369
escritura de archivos, 366-369
sintaxis, 367
función rewind(), 378
ejemplo de código, 379-384
sintaxis, 378
objeto, 34
argumentos
argc, 306
argv, 306
integrados
denominación, 308
funciones main(), 306
reemplazo, 308
línea de comandos, 305
recepción, 306-308
listas de, 47
paso a funciones, 47-48, 305
variables, procesamiento de, 252-254
array, tipo de datos, 424
arreglo, operador de subíndices del ([ ]), 190
arreglos, 190
acceso mediante apuntadores, 264-266
apuntadores
declaración, 272
referencia con, 195-196
cadenas, 272-274
de caracteres, 190, 210-211
sin especificación de tamaño, 209
de estructuras, 324-327
declaración, 190
despliegue, 196-198

Materiał chroniony prawem autorskim

elementos, 190
acceso, 190
enteros, 190
inicialización, 191-192
inicialización, 208-209
listado, 325-326
multidimensionales
declaración, 199
despliegue, 200-201
inicialización, 199-201
sin especificación de tamaño
paso a funciones, 266-267, 270-272
sin especificación de tamaño
cálculo, 201
en comparación con estructuras, 314
tamaño
cálculo, 192-194
especificación, 267

arreglos, referencias a elementos de, 191

ASCII, códigos de caracteres, 57

asctime(), función, 250-251

asignación
de apuntadores nulos, 183
de cadenas de caracteres a apuntadores, 210-211
de constantes de carácter a apuntadores, 210
de enteros a estructuras, 315
de memoria
función calloc(), 286-288
función malloc(), 280-283
de valores
a apuntadores, 180-181
a tipos de datos enum, 299
de valores enteros a tipos de datos enum, 296, 300

assert.h, archivo de encabezado, 439

asteriscos (*)
apuntadores, 180
determinación de significado, 182
operador de indirección, 182
operador de multiplicación, 182

aumento de la portabilidad del programa, 234

auto
especificador, variables, 229
palabra reservada, 56

B

\b, carácter de retroceso, 60

bibliotecas, 14

big-endian, formato, 343

binario
código, 14, 24
formato, 13

binarios
archivos
escritura, 378-381
lectura, 378-381
números
conversión de números decimales a, 129-130
negativos, 142

bits, 14, 58, 142
campos de
declaración, 347
definición, 347
ejemplo de código, 348-350
operadores de manipulación de, 130
AND (&) a nivel de bits, 131
complemento (~) a nivel de bits, 131
desplazamiento a la derecha (>>), 133-135
desplazamiento a la izquierda (<<), 133-135
OR (|) a nivel de bits, 131
XOR (^) a nivel de bits, 131

bloques
de instrucciones, 45-46

Materiał chroniony prawem aut

marcar como comentarios, 31

Borland C++, compilador, 21-24

ejecución, 23

inicio, 21

break

instrucciones, 155, 164-165

interrupción de ciclos infinitos, 166-167

listado, 164-165

salida de instrucciones switch, 164

ubicación, 164

palabra reservada, 56

búferes, 356

desalojar, 356

E/S de alto nivel, 357

E/S de bajo nivel, 357, 387

funciones setbuf(), 387

funciones setvbuf(), 387

bytes, 14, 58, 192-194

C

C

compilador de, comparación con el preprocesador de C, 392-393, 405

el lenguaje de programación, 15

historia, 12

portabilidad, 13

preprocesador de, 392

programación estructurada, 169

programas

legibilidad, mejoramiento de, 296, 300

mantenimiento, 296

ventajas, 12-14

%c, especificador de formato (carácter), 60-62, 74, 79

.c, extensión de nombre de archivo 28

C++, 14

cadena, especificador de formato (%s), 79

cadenas, 208

apuntadores, arreglos, 272-274

de caracteres, 196, 210-211

copiado, 213-215

inicialización, 208-211

lectura, 217

longitud, medición, 212-213

calificadores. Vea modificadores (variables)

calloc(), función, 286-288

comparada con funciones malloc(), 292

listado, 287

cambio de valores de variables mediante apuntadores, 183-184

CapturaDatos(), función, 346

CapturaInf(), función, 329

carácter

especificador de formato (%c), 60-62, 74, 79

tipo de datos. Vea char, tipo de datos

caracteres

arreglos, 190, 196-198

conversión de valores numéricos a, 61

escritura

al flujo de salida estándar, 215-216

desde el flujo de salida estándar, 217

función puts(), 215-217

ilegales (identificadores), 44

impresión, 60-62

lectura

desde el flujo de entrada estándar, 215-217

función gets(), 215-217

legales (identificadores), 44

nulos, 198. Vea también cadenas

valores numéricos

conversión, 61

muestra, 63-64

case, palabra reservada, 56, 162

char
palabra reservada, 56
tipo de datos, 47, 57
ciclos, 105, 427-428
anidación, 116-117
control del flujo, 155
do-while, 107-109
for, 109-112
expresiones complejas en, 113-115
instrucciones nulas, 112-113
infinitos, 166-167
instrucción continue, 171
saltos, 167
while, 106-107
cierre de archivos, función fclose(), 358-360, 371
clases, almacenamiento, 229
codificación, estilo de, 418-419
código, 14. Vea también listados
binario, 14
comentarios, 29-31, 418
anidados, 31
marcar como, 31
rendimiento, 30
división de líneas, 28
escritura
Borland C++, 21
Visual C++, 18
espaciado, 419
espacio en blanco, 29
espagueti, 169
fuente, 34
guardar
Borland C++, 23
Visual C++, 19
sangrado, 28, 419
sintaxis, verificación de, 36
códigos de caracteres (ASCII), 57
códigos, ejecución de la instrucción if, 156
combinación de declaraciones y definiciones, 315
comentarios, 29-31, 418
anidados, 31
marcar código como, 31
rendimiento del programa, 30
comillas dobles ("), 32-33, 59
comillas sencillas ('), 59
compatibilidad, error de, 280-281
compilación condicional, 397
anidada, 402-404
directiva #elif, 401-402
directiva #else, 399-402
directiva #endif, 397-402, 406
ejemplo de código, 398-399
sintaxis, 397
directiva #if
definiciones de macros, 400
ejemplo de código, 401-402
expresiones aritméticas, 406
sintaxis, 399
directiva #ifdef, 397-399
directiva #ifndef, 397-399
compiladores, 13
acceso, 17
Borland C++, 21-24
C, comparación con el preprocesador de C, 392-393, 405
errores de compatibilidad, 280-281
Microsoft, 18-21
optimizadores, desactivado de, 235
selección, 17
condiciones, evaluación (instrucción if), 156
consolidar tipos de datos, 300
const
modificador (variables), 234-235
palabra reservada, 56
constantes, 42
con nombre, en comparación con numéricas, 419
cuerpo de macro, 392
de cadena, 209-212, 423
comparadas con constantes de carácter, 209-212

Materiał chroniony prawem autorskim

Aprendiendo® C en 24 Horas

Encuentre una respuesta para...

- Comprender los tipos de datos, operadores, ciclos y cadenas
- Aprender a manejar la entrada y salida (E/S) estándar del programa
- Trabajar con arreglos, estructuras y uniones para manipular datos
- Aprovechar los apuntadores para acceder y recuperar elementos de datos
- Usar funciones y variables de C para procesar ecuaciones matemáticas
- Explorar las técnicas de administración de la memoria
- Forjarse una base para aprender programación más avanzada en C

Nivel de usuario:

Principiante — Intermedio — Avanzado

Categoría: Programación

Cubre: C

Pearson Educación

¡Comience ahora mismo!

24 lecciones en las que invertirá una hora por lección

En tan sólo 24 lecciones de una hora o menos, estará listo para utilizar C. A través de un método directo y paso a paso, cada lección complementa a la anterior, permitiéndole aprender los fundamentos de la programación en C.

Tips ofrecen atajos y soluciones

Notas aclaran conceptos y procedimientos

Precauciones ayudan a evitar tropiezos comunes

Tony Zhang es ingeniero de software y cuenta con más de 15 años de experiencia en programación de computadoras, programación de aplicaciones cliente/servidor, bases de datos y redes. Tony maneja además la programación de sistemas en X86 y procesadores digitales avanzados de señal/imagen.

SAMS

ISBN 968-444-495-8

Visítenos en:
www.prenhall.com