

# Cómo usar este libro

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

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

## Características especiales de este libro

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

### La función *printf()*

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

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

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

#### Ejemplo 1

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

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

### Despliega

;Este es un ejemplo de algo impreso!

### Ejemplo 2

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

### Despliega

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

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

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

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

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

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

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

## Haciendo un mejor libro

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

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

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

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

## Convenciones usadas en este libro

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



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

# Aprendiendo C en 21 días

*Peter Aitken / Bradley Jones*

**TRADUCCIÓN**

Ing. Sergio Luis Ma. Ruiz Faudon  
Ingeniero Químico

**REVISIÓN TÉCNICA**

Gabriel Guerrero Reyes  
Doctor en Informática

**BIBLIOTECA**

FAC. ING. MECÁNICA ELECTRÓNICA Y ELECTRÓNICA  
UNIVERSIDAD DE GUANAJUATO

**Universidad de Guanajuato**  
Biblioteca de la F.I.M.E.E.



0007229



**Pearson  
Educación**

®

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

# Resumen del contenido

## La semana 1 de un vistazo

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

**Revisión de la semana 1** **159**

## La semana 2 de un vistazo

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

**Revisión de la semana 2** **379**

## La semana 3 de un vistazo

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

**Revisión de la semana 3** **585**

## Apéndices

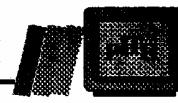
A	Tabla de caracteres ASCII .....	595
B	Palabras reservadas del C .....	599

<b>C</b>	Precedencia de operadores en C .....	603
<b>D</b>	Notación binaria y hexadecimal .....	605
<b>E</b>	Prototipos de función y archivos de encabezado .....	609
<b>F</b>	Funciones comunes en orden alfabético .....	619
<b>G</b>	Respuestas .....	627
<b>H</b>	Puntos específicos de los compiladores .....	679
<b>Indice</b>		<b>693</b>

# Contenido

<b>La semana de un vistazo</b>	<b>1</b>
<b>1 Comienzo .....</b>	<b>3</b>
Una breve historia del lenguaje C .....	4
¿Por qué usar C? .....	4
Preparación para la programación .....	5
El ciclo de desarrollo del programa .....	6
Creación del código fuente .....	7
Compilación del código fuente .....	8
Enlazar para crear un archivo ejecutable .....	9
Completando el ciclo de desarrollo .....	10
El primer programa en C .....	11
Tecleo y compilación de HELLO.C .....	12
Resumen .....	15
Preguntas y respuestas .....	16
Taller .....	17
Cuestionario .....	17
Ejercicios .....	18
<b>2 Los componentes de un programa C .....</b>	<b>21</b>
Un programa corto en C .....	22
Los componentes de un programa .....	23
La función <i>main()</i> (líneas 5-18) .....	23
La directiva <i>#include</i> (línea 2) .....	23
Definición de variables (línea 3) .....	24
Prototipo de función (línea 4) .....	24
Enunciados del programa (líneas 8, 9, 12, 13, 16, 17, 23) .....	24
Definición de función (líneas 21-24) .....	25
Comentarios del programa (líneas 1, 7, 11, 15, 20) .....	26
Llaves (líneas 6, 18, 22, 24) .....	27
Ejecución del programa .....	27
Una nota sobre la precisión .....	27
Revisión de las partes de un programa .....	28
Resumen .....	30
Preguntas y respuestas .....	30
Taller .....	31
Cuestionario .....	31
Ejercicios .....	32
<b>3 Variables y constantes numéricas .....</b>	<b>35</b>
Memoria de la computadora .....	36
Variables .....	37

Nombres de variable .....	37
Tipos de variables numéricas .....	39
Declaración de variables .....	42
La palabra clave <i>typedef</i> .....	43
Inicialización de variables numéricas .....	43
Constantes .....	44
Constantes literales .....	44
Constantes simbólicas .....	46
Resumen .....	49
Preguntas y respuestas .....	50
Taller .....	51
Cuestionario .....	51
Ejercicios .....	52
<b>4 Enunciados, expresiones y operadores .....</b>	<b>53</b>
Enunciados .....	54
Enunciados y el espacio en blanco .....	54
Enunciados compuestos .....	55
Expresiones .....	56
Expresiones simples .....	56
Expresiones complejas .....	56
Operadores .....	58
El operador de asignación .....	58
Operadores matemáticos .....	58
Precedencia de operadores y los paréntesis .....	63
Orden para la evaluación de subexpresiones .....	65
Operadores relacionales .....	65
El enunciado <i>if</i> .....	67
Evaluación de expresiones relacionales .....	72
Precedencia de los operadores relacionales .....	73
Operadores lógicos .....	75
Más sobre valores cierto/falso .....	76
Precedencia de los operadores lógicos .....	77
Operadores de asignación compuestos .....	79
El operador condicional .....	80
El operador coma .....	80
Resumen .....	81
Preguntas y respuestas .....	82
Taller .....	83
Cuestionario .....	83
Ejercicios .....	84
<b>5 Funciones: lo básico .....</b>	<b>87</b>
¿Qué es una función? .....	88
La definición de una función .....	88
La ilustración de una función .....	89



---

La manera en que trabaja una función .....	91
Las funciones y la programación estructurada .....	93
Las ventajas de la programación estructurada .....	93
La planeación de un programa estructurado .....	93
El enfoque descendente .....	95
Escritura de una función .....	96
El encabezado de la función .....	96
El cuerpo de la función .....	99
El prototipo de la función .....	104
Paso de argumentos a una función .....	105
Llamado de funciones .....	106
Recursión .....	107
¿Dónde se ponen las funciones? .....	109
Resumen .....	110
Preguntas y respuestas .....	110
Taller .....	111
Cuestionario .....	111
Ejercicios .....	112
<b>6 Control básico del programa .....</b>	<b>115</b>
Arreglos: lo básico .....	116
Control de la ejecución del programa .....	117
El enunciado <i>for</i> .....	117
Enunciados <i>for</i> anidados .....	123
El enunciado <i>while</i> .....	125
Enunciados <i>while</i> anidados .....	128
El ciclo <i>do...while</i> .....	130
Ciclos anidados .....	134
Resumen .....	135
Preguntas y respuestas .....	136
Taller .....	136
Cuestionario .....	137
Ejercicios .....	137
<b>7 Entrada/salida básica .....</b>	<b>139</b>
Desplegado de la información en la pantalla .....	140
La función <i>printf()</i> .....	140
Desplegado de mensajes con <i>puts()</i> .....	148
Entrada de datos numéricos con <i>scanf()</i> .....	149
Resumen .....	154
Preguntas y respuestas .....	154
Taller .....	155
Cuestionario .....	155
Ejercicios .....	156
<b>Revisión de la semana</b>	<b>159</b>

<b>La semana de un vistazo</b>	<b>167</b>
<b>8 Arreglos numéricos .....</b>	<b>169</b>
¿Qué es un arreglo? .....	170
Arreglos de una sola dimensión .....	170
Arreglos multidimensionales .....	175
Denominación y declaración de arreglos .....	176
Inicialización de arreglos .....	178
Tamaño máximo del arreglo .....	182
Resumen .....	184
Preguntas y respuestas .....	185
Taller .....	186
Cuestionario .....	186
Ejercicios .....	186
<b>9 Apuntadores .....</b>	<b>189</b>
¿Qué es un apuntador? .....	190
La memoria de la computadora .....	190
Creación de un apuntador .....	191
Los apuntadores y las variables simples .....	192
Declaración de apuntadores .....	192
Inicialización de apuntadores .....	192
Uso de apuntadores .....	193
Los apuntadores y los tipos de variables .....	195
Los apuntadores y los arreglos .....	197
El nombre del arreglo como un apuntador .....	197
Almacenamiento de elementos de arreglo .....	198
Aritmética de apuntadores .....	200
Precauciones con los apuntadores .....	204
Notación de subíndices de arreglo y apuntadores .....	205
Paso de arreglos a funciones .....	206
Resumen .....	210
Preguntas y respuestas .....	211
Taller .....	211
Cuestionario .....	212
Ejercicios .....	212
<b>10 Caracteres y cadenas .....</b>	<b>215</b>
El tipo de dato <i>char</i> .....	216
Uso de variables de carácter .....	217
Uso de cadenas .....	219
Arreglos de caracteres .....	219
Inicialización de arreglos de caracteres .....	220
Cadenas y apuntadores .....	221
Cadenas sin arreglos .....	221
Asignación de espacio para la cadena en la compilación .....	222



La función <i>malloc()</i> .....	222
Desplegado de cadenas y caracteres .....	227
La función <i>puts()</i> .....	227
La función <i>printf()</i> .....	228
Lectura de cadenas desde el teclado .....	229
Entrada de cadenas con la función <i>gets()</i> .....	229
Entrada de cadenas con la función <i>scanf()</i> .....	232
Resumen .....	234
Preguntas y respuestas .....	235
Taller .....	236
Cuestionario .....	236
Ejercicios .....	238
<b>11 Estructuras .....</b>	<b>241</b>
Estructuras simples .....	242
Definición y declaración de estructuras .....	242
Acceso de los miembros de la estructura .....	243
Estructuras más complejas .....	245
Estructuras que contienen estructuras .....	245
Estructuras que contienen arreglos .....	249
Arreglos de estructuras .....	251
Inicialización de estructuras .....	255
Estructuras y apuntadores .....	257
Apuntadores como miembros de estructuras .....	258
Apuntadores a estructuras .....	260
Apuntadores y arreglos de estructuras .....	262
Paso de estructuras como argumentos a funciones .....	265
Uniones .....	267
Definición, declaración e inicialización de uniones .....	267
Acceso de miembros de la unión .....	267
Listas encadenadas .....	272
La organización de una lista encadenada .....	273
La función <i>malloc()</i> .....	275
Implementación de una lista encadenada .....	275
<i>typedef</i> y las estructuras .....	275
Resumen .....	276
Preguntas y respuestas .....	277
Taller .....	277
Cuestionario .....	278
Ejercicios .....	278
<b>12 Alcance de las variables .....</b>	<b>281</b>
¿Qué es el alcance? .....	282
Una demostración del alcance .....	282
¿Por qué es importante el alcance? .....	284

Variables externas .....	284
Alcance de las variables externas .....	285
Cuándo usar variables externas .....	285
La palabra clave <i>extern</i> .....	286
Variables locales .....	287
Variables estáticas versus automáticas .....	287
El alcance de los parámetros de la función .....	290
Variables estáticas externas .....	291
Variables de registro .....	291
Variables locales y la función <i>main()</i> .....	292
¿Qué clase de almacenamiento se debe usar?.....	293
Variables locales y bloques .....	294
Resumen .....	295
Preguntas y respuestas .....	296
Taller .....	297
Cuestionario .....	297
Ejercicios .....	298
<b>13   Más sobre el control de programa .....</b>	<b>301</b>
Terminación anticipada de ciclos .....	302
El enunciado <i>break</i> .....	302
El enunciado <i>continue</i> .....	304
El enunciado <i>goto</i> .....	306
Ciclos infinitos .....	309
El enunciado <i>switch</i> .....	312
Terminación del programa .....	321
La función <i>exit()</i> .....	321
La función <i>atexit()</i> (sólo para el DOS) .....	322
Ejecución de comandos del sistema operativo en un programa .....	325
Resumen .....	327
Preguntas y respuestas .....	327
Taller .....	328
Cuestionario .....	328
Ejercicios .....	328
<b>14   Trabajando con la pantalla, la impresora y el teclado .....</b>	<b>331</b>
Los flujos y el C .....	332
¿Qué es exactamente la Entrada/Salida de un programa? .....	332
¿Qué es un flujo? .....	333
Flujos de texto contra flujos binarios .....	334
Los flujos predefinidos .....	334
Funciones de flujo del C.....	335
Un ejemplo .....	335
Aceptando entrada del teclado .....	336
Entrada de caracteres .....	336
Entrada formateada .....	351



---

Salida a pantalla .....	359
Salida de caracteres con <i>putchar()</i> , <i>putc()</i> y <i>fputc()</i> .....	359
Uso de <i>puts()</i> y <i>fputs()</i> para la salida de flujos .....	361
Uso de <i>printf()</i> y <i>fprintf()</i> para la salida formateada .....	362
Redirección de la entrada y la salida .....	369
Cuándo usar <i>fprintf()</i> .....	371
Uso de <i>stderr</i> .....	371
Resumen .....	373
Preguntas y respuestas .....	374
Taller .....	375
Cuestionario .....	375
Ejercicios .....	376
<b>Revisión de la semana</b>	<b>379</b>
<b>La semana de un vistazo</b>	<b>389</b>
<b>15 Más sobre apuntadores .....</b>	<b>391</b>
Apuntadores a apuntadores .....	392
Apuntadores y arreglos de varias dimensiones .....	393
Arreglos de apuntadores .....	402
Cadenas y apuntadores: una revisión .....	402
Arreglos de apuntadores a <i>char</i> .....	403
Un ejemplo .....	405
Apuntadores a funciones .....	411
Declaración de un apuntador a una función .....	411
Inicialización y uso de un apuntador a una función .....	412
Resumen .....	421
Preguntas y respuestas .....	421
Taller .....	422
Cuestionario .....	422
Ejercicios .....	423
<b>16 Uso de archivos de disco .....</b>	<b>425</b>
Flujos y archivos de disco .....	426
Tipos de archivos de disco .....	426
Nombres de archivo .....	427
Apertura de un archivo para usarlo .....	427
Escritura y lectura de datos de archivo .....	431
Entrada y salida de archivos formateados .....	431
Entrada y salida de caracteres .....	436
Entrada y salida directas de archivos .....	438
Bufer con archivos: cierre y vaciado de archivos .....	441
Acceso de archivos secuencial contra aleatorio .....	443
Las funciones <i>ftell()</i> y <i>rewind()</i> .....	444
La función <i>fseek()</i> .....	446
Detección del fin de archivo .....	449

Funciones para manejo de archivos .....	452
Borrado de un archivo .....	452
Renombrado de un archivo .....	453
Copiado de un archivo .....	454
Uso de archivos temporales .....	457
Resumen .....	459
Preguntas y respuestas .....	459
Taller .....	460
Cuestionario .....	460
Ejercicios .....	461
<b>17 Manipulación de cadenas .....</b>	<b>463</b>
Longitud y almacenamiento de cadenas .....	464
Copia de cadenas .....	465
La función <i>strcpy()</i> .....	465
La función <i>strncpy()</i> .....	467
La función <i>strdup()</i> .....	468
Concatenación de cadenas .....	469
La función <i>strcat()</i> .....	469
La función <i>strncat()</i> .....	471
Comparación de cadenas .....	472
Comparación de dos cadenas .....	473
Comparación de dos cadenas: ignorando mayúsculas y minúsculas .....	475
Comparación parcial de cadenas .....	475
Búsqueda en cadenas .....	476
La función <i>strchr()</i> .....	476
La función <i> strrchr()</i> .....	478
La función <i>strcspn()</i> .....	478
La función <i>strspn()</i> .....	479
La función <i>strpbrk()</i> .....	480
La función <i>strstr()</i> .....	481
Conversión de cadenas .....	482
Funciones diversas para cadenas .....	483
La función <i>strrev()</i> .....	483
Las funciones <i>strset()</i> y <i>strnset()</i> .....	484
Conversión de cadenas a números .....	485
La función <i>atoi()</i> .....	485
La función <i>atol()</i> .....	486
La función <i>atof()</i> .....	486
Funciones de prueba de caracteres .....	487
Resumen .....	492
Preguntas y respuestas .....	492
Taller .....	493
Cuestionario .....	493
Ejercicios .....	493



---

<b>18</b>	<b>Cómo obtener más de las funciones .....</b>	<b>495</b>
	Paso de apuntadores a funciones .....	496
	Apuntadores tipo <i>void</i> .....	500
	Funciones con número variable de argumentos .....	504
	Funciones que regresan un apuntador .....	507
	Resumen .....	509
	Preguntas y respuestas .....	510
	Taller .....	510
	Cuestionario .....	510
	Ejercicios .....	511
<b>19</b>	<b>Exploración de la biblioteca de funciones .....</b>	<b>513</b>
	Funciones matemáticas .....	514
	Funciones trigonométricas .....	514
	Funciones exponenciales y logarítmicas .....	515
	Funciones hiperbólicas .....	515
	Otras funciones matemáticas .....	516
	Manejo del tiempo .....	518
	Representación del tiempo .....	518
	Las funciones de tiempo .....	518
	Uso de las funciones de tiempo .....	522
	Funciones para el manejo de errores .....	524
	La función <i>assert()</i> .....	524
	El archivo de encabezado ERRNO.H .....	526
	La función <i>perror()</i> .....	527
	Búsqueda y ordenamiento .....	529
	Búsqueda con <i>bsearch()</i> .....	529
	Ordenamiento con <i>qsort()</i> .....	530
	Dos demostraciones de búsqueda y ordenamiento .....	531
	Resumen .....	537
	Preguntas y respuestas .....	537
	Taller .....	538
	Cuestionario .....	538
	Ejercicios .....	538
<b>20</b>	<b>Otras funciones .....</b>	<b>541</b>
	Conversiones de tipo .....	542
	Conversiones automáticas de tipo .....	542
	Conversiones explícitas con modificadores de tipo .....	543
	Asignación de espacio de almacenamiento en memoria .....	545
	La función <i>malloc()</i> .....	546
	La función <i>calloc()</i> .....	546
	La función <i>realloc()</i> .....	547
	La función <i>free()</i> .....	549
	Uso de argumentos de la línea de comandos .....	551

Operaciones sobre bits .....	554
Los operadores de desplazamiento .....	554
Los operadores lógicos a nivel de bit .....	555
El operador de complemento .....	556
Campos de bits en estructuras .....	556
Resumen .....	558
Preguntas y respuestas .....	559
Taller .....	560
Cuestionario .....	560
Ejercicios .....	561
<b>21 Cómo aprovechar las directivas del preprocesador y más .....</b>	<b>563</b>
Programación con varios archivos fuente .....	564
Ventajas de la programación modular .....	564
Técnicas de la programación modular .....	564
Componentes de los módulos .....	566
Variables externas y la programación modular .....	567
Uso de archivos .OBJ .....	568
El preprocesador de C .....	569
La directiva del preprocesador <code>#define</code> .....	569
La directiva <code>#include</code> .....	575
Uso de <code>#if</code> , <code>#elif</code> , <code>#else</code> y <code>#endif</code> .....	576
Uso de <code>#if...#endif</code> para ayudarse en la depuración .....	577
Cómo evitar la inclusión múltiple de archivos de encabezado .....	578
La directiva <code>#undef</code> .....	579
Macros predefinidas .....	579
Resumen .....	580
Preguntas y respuestas .....	581
Taller .....	581
Cuestionario .....	581
Ejercicios .....	582
<b>Revisión de la semana</b>	<b>585</b>
<b>Apéndices</b>	
<b>A Tabla de caracteres ASCII .....</b>	<b>595</b>
<b>B Palabras reservadas del C .....</b>	<b>599</b>
<b>C Precedencia de operadores en C .....</b>	<b>603</b>
<b>D Notación binaria y hexadecimal .....</b>	<b>605</b>
<b>E Prototipos de función y archivos de encabezado .....</b>	<b>609</b>
<b>F Funciones comunes en orden alfabético .....</b>	<b>619</b>
<b>G Respuestas .....</b>	<b>627</b>

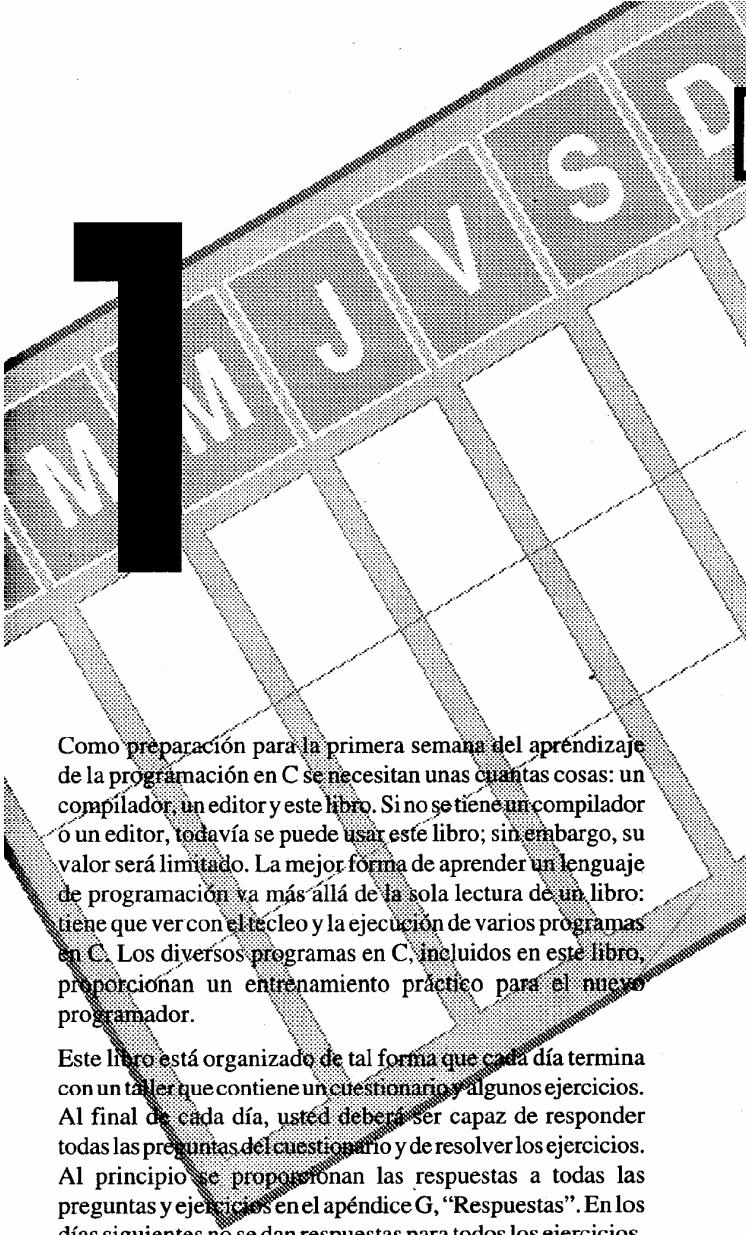


---

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

---

Respuestas para el Día 16 “Uso de archivos de disco” .....	671
Cuestionario .....	671
Ejercicios .....	672
Respuestas para el Día 17 “Manipulación de cadenas” .....	672
Cuestionario .....	672
Ejercicios .....	673
Respuestas para el Día 18 “Obteniendo más de las funciones” .....	674
Cuestionario .....	674
Ejercicios .....	674
Respuestas para el Día 19 “Exploración de la biblioteca de funciones” ....	675
Cuestionario .....	675
Ejercicios .....	676
Respuestas para el Día 20 “Otras funciones” .....	676
Cuestionario .....	676
Ejercicios .....	677
Respuestas para el Día 21 “Aprovechando las directivas del preprocesador y más” .....	678
Cuestionario .....	678
<b>H. Puntos específicos de los compiladores .....</b>	<b>679</b>
Instalación de la edición estándar del Visual C/C++ de Microsoft .....	682
Instalación de lo mínimo .....	683
Instalación del Turbo C/C++ para DOS de Borland .....	685
Instalación de lo mínimo para el Turbo C/C++ para DOS de Borland .....	686
¿Qué ofrecen los compiladores? .....	688
Borland C++ .....	688
Turbo C++ para DOS de Borland .....	689
Edición estándar del Visual C++ de Microsoft .....	690
Otros compiladores .....	691
<b>Indice .....</b>	<b>693</b>



**SEMANA**

# UN VISTAZO

- |   |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |

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

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

## Dónde andamos...

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

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



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



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

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

## Una breve historia del lenguaje C

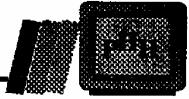
Tal vez se pregunte cuál ha sido el origen del lenguaje C y de dónde le vino su elegante nombre. El C fue creado por Dennis Ritchie en los laboratorios de la Bell Telephone, en 1972. El lenguaje no fue creado por el gusto de hacerlo, sino para un fin específico: el diseño del sistema operativo UNIX (el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

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

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

## Por qué usar C

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



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

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

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

## Preparación para la programación

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

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

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

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

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

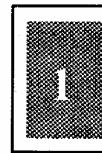
Como un ejemplo, supongamos que alguien nos pide escribir un programa para determinar el área de un círculo. El paso 1 está completo, ya que se sabe el objetivo: determinar el área de un círculo. El paso 2 consiste en determinar lo que se necesita saber para calcular el área. En este ejemplo, supongamos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

## El ciclo de desarrollo del programa

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



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



## Creación del código fuente

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

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

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

### Uso de un editor

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

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

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

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

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



**DEBE**

**NO DEBE**

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

## Compilación del código fuente

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

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

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

Para compilar RADIUS.C en una máquina UNIX, use

cc radius.c

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

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







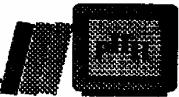












varios niveles intermedios. En los programas se debe ver cada mensaje y tomar una determinación. Siempre es mejor tratar de escribir todos los programas sin que aparezca ningún mensaje de advertencia o de error. (Con un mensaje de error el compilador no creará el archivo ejecutable.)

## Taller

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

## Cuestionario

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

## Ejercicios

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



## Comienzo

---

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

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

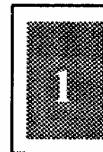
```
1: #include <stdio.h>
2:
3: int x,y;
4:
5: main()
6: {
7:     for ( x = 0; x < 10; x++ , printf( "\n" ) )
8:         for ( y = 0; y < 10; y++ )
9:             printf( "X" );
10:
11:    return 0;
12: }
```

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

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

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

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



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

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

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

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

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



## Comienzo

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



## Los componentes de un programa C

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

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

## Un pequeño programa en C

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

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

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



### Listado 2.1. MULTIPLY.C.

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

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

La salida del listado 2.1 es

```
Enter a number between 1 and 100: 35
Enter another number between 1 and 100: 23
35 times 23 = 805
```

## Los componentes de un programa

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

### La función `main()` (líneas 5-18)

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

### La directiva `#include` (línea 2)

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

## **Los componentes de un programa C**

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

## **Definición de variables (línea 3)**

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

## **Prototipo de función (línea 4)**

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

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

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

### ***printf()***

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

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

### ***scanf()***

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

### ***c = product (a,b);***

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

### ***return (x \* y);***

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

## **Definición de función (líneas 21-24)**

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

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

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

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

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

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

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

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

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

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

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

### DEBE

### NO DEBE

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

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

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

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

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

## Llaves (líneas 6, 18, 22, 24)

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

## Ejecución del programa

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

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

## Una nota sobre la precisión

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

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

## Revisión de las partes de un programa

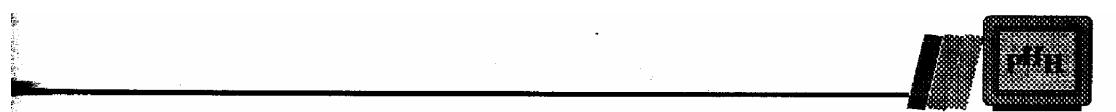
Ahora que han sido descritas todas las partes del programa, usted deberá ser capaz de ver cualquier programa y encontrar algunas similitudes. Examine el listado 2.2, LIST\_IT.C y vea si puede identificar las diferentes partes.



**Listado 2.2. LIST\_IT.C.**

```

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



```
1:     fclose(fp);
2:     return 0;
3: }
4:
5: void display_usage(void)
6: {
7:     fprintf(stderr, "\nProper Usage is: " );
8:     fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
9: }
```

continuación se presenta la salida del listado 2.2

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

2



## Los componentes de un programa C

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



LIST\_IT.C es muy similar a PRINT\_IT.C, que se tecleó en el ejercicio siete del Día 1, “Comienzo”. El listado 2.2 despliega en la pantalla listados de programas C guardados, en vez de enviarlos a la impresora.

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

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

## Resumen

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

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

## Preguntas y respuestas

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

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

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

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

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

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

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

2

## ler

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

## uestionario

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



## Los componentes de un programa C

---

10. ¿Qué es un archivo de inclusión?

### Ejercicios

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

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

3. Escriba un ejemplo de un comentario.



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

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



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

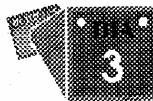
```
1: /* EX2-5.C */
2: #include <stdio.h>
3: #include <string.h>
4: main()
5: {
6:     char buffer[256];
7:
8:     printf( "Enter your name and press <Enter>:\n" );
9:     gets( buffer );
10:
11:    printf( "\nYour name has %d characters and spaces!",
12:            strlen( buffer ) );
13:
14:    return 0;
15: }
```

DIA

3

0

Variables  
y constantes  
numéricas



## Variables y constantes numéricas

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

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

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

## Memoria de la computadora

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

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

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

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



**Tabla 3.1. Espacio de memoria requerido para guardar datos.**

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

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

3

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

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

## VARIABLES

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

### NOMBRES DE VARIABLE

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

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

## Variables y constantes numéricas

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

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

```

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

```

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

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

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

## DEBE

## NO DEBE

**DEBE** Usar nombres de variable que sean descriptivos.

**DEBE** Adoptar un estilo para nombrar las variables y sígalo.

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

**NO DEBE** Usar nombres de variables en mayúsculas innecesariamente.

## Tipos de variables numéricas

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

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

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

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

## Variables y constantes numéricas

**Tabla 3.2. Tipos de datos numéricos del C.**

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

<sup>1</sup> Rango aproximado; precisión = 7 dígitos.

<sup>2</sup> Rango aproximado; precisión = 19 dígitos.

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

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

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

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

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

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

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

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



## Variables y constantes numéricas

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



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

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

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

## Declaración de variables

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

`nombre de tipo nombre de variable;`

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

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

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

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

## La palabra clave `typedef`

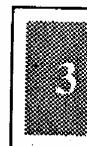
- La palabra clave `typedef` es usada para crear un nuevo nombre para un tipo de dato existente. De hecho, `typedef` crea un sinónimo. Por ejemplo, el enunciado
 

```
typedef int entero;
```

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

```
entero contador;
```

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



## Inicialización de variables numéricas

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

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

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

`x = 12`

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

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

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

## Variables y constantes numéricas

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

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

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

### DEBE

### NO DEBE

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

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

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

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

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

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

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

## Constantes

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

## Constantes literales

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

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

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

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

123.456  
0.019  
100.

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

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

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

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

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



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

## Constantes simbólicas

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

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

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

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

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

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

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

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



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

```
#define NOMBREDECONSTANTE literal
```

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

```
#define PI 3.14159
```

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

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

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

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

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

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

## Variables y constantes numéricas

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

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

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

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

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

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

El listado 3.2 produce la siguiente salida:



```
Enter your weight in pounds: 175
Enter your year of birth: 1960
Your weight in grams = 13914
In 2000 you will be 40 years old
```



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

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

3

## DEBE

## NO DEBE

**DEBE** Usar constantes para hacer que los programas sean más fáciles de leer.

**NO DEBE** Tratar de asignar un valor a una constante después de que ha sido iniciada.

# Resumen

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



## Variables y constantes numéricas

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

## Preguntas y respuestas

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

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

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

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

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

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

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

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

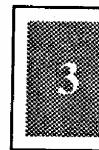
5. ¿Cuál es la diferencia práctica entre las constantes simbólicas creadas con la directiva `#define` y las creadas con la palabra clave `const`?



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

## Taller

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



### Cuestionario

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

### Ejercicios

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



## Variables y constantes numéricas

---

- e. El costo de una cosa.
  - f. La calificación máxima de un examen (suponga que es siempre 100).
  - g. La temperatura.
  - h. El valor neto de una persona.
  - i. La distancia a una estrella, en millas.
2. Determine nombres de variable adecuados para los valores del ejercicio 1.
  3. Escriba declaraciones para las variables del ejercicio 2.
  4. ¿Cuáles de los siguientes nombres de variable son válidos?
    - a. 123variable
    - b. x
    - c. anotación\_total
    - d. Peso\_en\_#s
    - e. uno
    - f. costo-bruto
    - g. RADIO
    - h. Radio
    - i. radio
    - j. ésta\_es\_una \_variable\_para\_guardar\_el\_ancho\_de\_una\_caja

**DIA**

**4**

**Enunciados,  
expresiones y  
operadores**



## Enunciados, expresiones y operadores

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

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

## Enunciados

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

`x = 2 + 3;`

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

## Enunciados y el espacio en blanco

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

`x = 2 + 3;`

y también es equivalente a

x =  
2  
+  
3;



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

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

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

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

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



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

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

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

## Enunciados compuestos

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

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

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

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

**BIBLIOTECA**  
FAC. ING. MECÁNICA, ELÉCTRICA Y ELECTRÓNICA  
UNIVERSIDAD DE GUANAJUATO

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

**DEBE****NO DEBE**

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

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

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

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

## Expresiones

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

### Expresiones simples

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

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

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

### Expresiones complejas

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

`2 + 8`

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

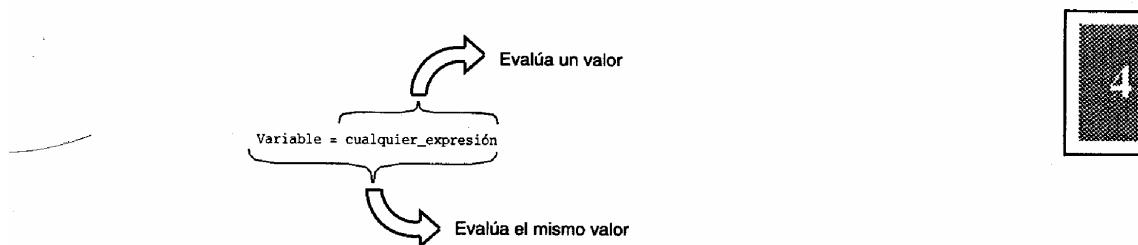
$1.25 / 8 + 5 * \text{tasa} + \text{tasa} * \text{tasa} / \text{costo}$

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

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

$x = a + 10;$

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



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

Por lo tanto, se pueden escribir enunciados como

$y = x = a + 10;$

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

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

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

## Operadores

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

### El operador de asignación

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

`x = y;`

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

variable = expresión;

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

## Operadores matemáticos

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

### Los operadores matemáticos unarios

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

Tabla 4.1. Operadores matemáticos unarios del C.

Operador	Símbolo	Acción	Ejemplo
<u>Incremento</u>	<code>++</code>	Incrementa al operando en 1	<code>++x, x++</code>
<u>Decremento</u>	<code>--</code>	Decrementa al operando en 1	<code>--x, x--</code>

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



`++x;  
-y;`

son equivalentes a

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

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

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

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

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

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

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

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

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

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



#### Listado 4.1. UNARY.C.

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

**Listado 4.1. continuación**

```

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

**Salir** La salida del programa es

```

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

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

**Los operadores matemáticos binarios**

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

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

```

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

**Tabla 4.2. Operadores matemáticos binarios del C.**

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

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

4

**Listado 4.2. SECONDS.C.**

```

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

```

**Listado 4.2. continuación**

```

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



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



El programa SECONDS.C sigue el mismo formato que han seguido los programas anteriores. Las líneas 1-3 proporcionan algunos comentarios para indicar lo que va a hacer el programa. La línea 4 es espacio en blanco para hacer más legible al programa.

De manera similar al espacio en blanco que se encuentra en enunciados y expresiones, las líneas en blanco son ignoradas por el compilador. La línea 5 incluye los archivos de encabezado necesarios para este programa. Las líneas 9 y 10 definen dos constantes, SECS\_PER\_MIN y SECS\_PER\_HOUR, que se usan para facilitar la lectura de los enunciados en el programa. La línea 12 declara todas las variables que serán usadas. A algunas personas les gusta declarar cada variable en su propia línea, en vez de declararlas todas en una sola, como se muestra anteriormente. De manera similar a muchos elementos del C, esto es, cuestión de estilo. Cualquier método es correcto.

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



## Precedencia de operadores y los paréntesis

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

$x = 4 + 5 * 3;$

Si primero se ejecuta la suma, se tiene

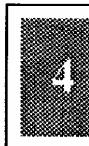
$x = 9 * 3;$

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

$x = 4 + 15;$

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

La precedencia de los operadores matemáticos del C se lista en la tabla 4.3. El número 1 es la mayor precedencia.



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

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

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

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

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

$12 \% 5 * 2$

## Enunciados, expresiones y operadores

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

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

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

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

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

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

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

Esta evaluación se ejecuta en los siguientes pasos:

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

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

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

$$25 - (2 * 14)$$

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

$$25 - 28$$

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

$$x = -3$$

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

## Orden para la evaluación de subexpresiones

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

$w * x / y * z$

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

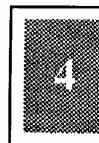
Sin embargo, entre los niveles de precedencia no hay garantía de que se siga el orden de izquierda a derecha. Vea esta expresión:

$w * x / y + z / y$

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

$w * x / ++y + z / y$

- Si la primera subexpresión es evaluada primero,  $y$  ha sido incrementada cuando es evaluada la segunda expresión. Si la segunda expresión es evaluada primero,  $y$  no ha sido incrementada y el resultado es diferente. Por lo tanto, se debe evitar este tipo de expresiones indeterminadas en la programación.
- El apéndice C, "Precedencia de operadores en C", lista la precedencia de todos los operadores de C.



### DEBE

### NO DEBE

**DEBE** Usar paréntesis para hacer claro el orden de evaluación de las expresiones.

**NO DEBE** Sobre cargar una expresión. Por lo general, es más claro partir una expresión en dos o más enunciados. Esto es especialmente cierto cuando se usan los operadores unarios (`++`) o (`--`).

## Operadores relacionales

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

## Enunciados, expresiones y operadores

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

**Tabla 4.4. Operadores relacionales del C.**

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

**Tabla 4.5. Operadores relacionales en uso.**

Expresión	Evalúa a
<code>5 == 1</code>	0 (falso)
<code>5 &gt; 1</code>	1 (cierto)
<code>5 != 1</code>	1 (cierto)
<code>(5 + 10) == (3 * 5)</code>	1 (cierto)

### DEBE

### NO DEBE

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

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

## enunciado if

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

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

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

```
if (expresión)
    enunciado;
```

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

### DEBE

### NO DEBE

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

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

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

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

## 4 Enunciados, expresiones y operadores

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

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

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

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

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

### Captura

#### Listado 4.3. IF.C.

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



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



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



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



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

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

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

**Captura****Listado 4.4. El enunciado if con una cláusula else.**

```

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

**Salida**

E:\>list0404  
Input an integer value for x: 99  
Input an integer value for y: 8  
x is greater than y  
E:\>list0404  
Input an integer value for x: 8  
Input an integer value for y: 99  
x is smaller than y  
E:\>list0404  
Input an integer value for x: 99  
Input an integer value for y: 99  
x is equal to y

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

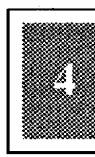
Observe que el programa del listado 4.4 usa enunciados if anidados. El anidado significa poner (anidar) uno o más enunciados de C dentro de otro enunciado de C. En el caso del listado 4.4, un enunciado *if* es parte de la cláusula *else* del primer enunciado *if*.

## **El enunciado if**

### **Forma 1**

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

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

4

### **Forma 2**

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

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

### **Forma 3**

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

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

## Enunciados, expresiones y operadores

### Ejemplo 1

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

### Ejemplo 2

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

## Evaluación de expresiones relacionales

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

### Listado 4.5. Demostración de la evaluación de expresiones relacionales.

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

A continuación se presenta la salida del listado 4.5:

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

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

#### La expresión

`x = 5`

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

`x == 5`

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

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

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

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

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

4

## Precedencia de los operadores relacionales

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

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

`if (x + 2 > y)`

## 4

## Enunciados, expresiones y operadores

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

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

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

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

Por lo tanto, si se escribe

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

es lo mismo que escribir

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

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

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

### DEBE

### NO DEBE

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

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

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

#### ■ 4.6. Orden de precedencia de los operadores relacionales del C.

Operador	Precedencia relativa
<code>&lt;&lt;= &gt;=</code>	1
<code>!= ==</code>	2

## Operadores lógicos

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

#### ■ 4.7. Operadores lógicos del C.

Operador	Símbolo	Ejemplo
y	<code>&amp;&amp;</code>	<code>exp1 &amp;&amp; exp2</code>
o	<code>  </code>	<code>exp1    exp2</code>
no	<code>!</code>	<code>!exp1</code>

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

#### ■ 4.8. Operadores lógicos del C en uso.

Expresión	Evaluá a
<code>(exp1 &amp;&amp; exp2)</code>	Cierto (1) solamente si ambos <code>exp1</code> y <code>exp2</code> son ciertos. En caso contrario, falso (0).
<code>(exp1    exp2)</code>	Cierto (1) si cualquiera de <code>exp1</code> y <code>exp2</code> es cierto. En caso contrario, falso (0).
<code>(!exp1)</code>	Falso (0) si <code>exp1</code> es cierto, y cierto (1) si <code>exp1</code> es falso.

## Enunciados, expresiones y operadores

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

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

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

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

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

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

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

o

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

## Más sobre valores cierto/falso

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

Un valor de 0 representa falso.

Cualquier valor diferente de 0 representa cierto.

Esto es ilustrado por el siguiente ejemplo:

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

En este caso, el valor de  $x$  es impreso.

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

(expresión)

es equivalente a escribir

(expresión != 0)

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

(!expresión)

que es equivalente a

(expresión == 0)

## Precedencia de los operadores lógicos

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

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

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

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

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

$a < b \&& (a < c \&& c < d)$

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

$a < b \&& (a < c \&& c < d)$



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

$(a < b \text{ || } a < c) \text{ && } c < d$

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

**Captura****Listado 4.6. Precedencia de los operadores lógicos.**

```

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

**Salida**

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

**Análisis**

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

## Operadores de asignación compuestos

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

$x = x + 5;$

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

$x += 5;$

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

$exp1 \ op= exp2$

que es equivalente a escribir:

$exp1 = exp1 \ op \ exp2;$

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



**Tabla 4.10. Ejemplos de operadores de asignación compuestos.**

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

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

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

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

## El operador condicional

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

```
exp1 ? exp2 : exp3
```

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

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

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

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

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

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

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

## El operador coma

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

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

Por ejemplo, el enunciado

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

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

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

### DEBE

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

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

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

### NO DEBE



## Resumen

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

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

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

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

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

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

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

## Preguntas y respuestas

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

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

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

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

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

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

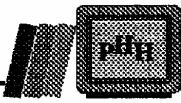
- ✓ 4. ¿Es el operador de resta (-) unario o binario?

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

$x = -y;$

contra el uso binario que se muestra:

$x = a - b; //$



5. Los números negativos son considerados ciertos o falsos?

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

## aller

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

### Cuestionario

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

$$x = 5 + 8;$$

2. ¿Qué es una expresión?

3. En una expresión que contiene varios operadores, ¿qué es lo que determina el orden en el que se ejecutan las operaciones?

4. Si la variable x tiene el valor de 10, ¿cuáles son los valores de x y a después de que cada uno de los siguientes enunciados se ejecuta por separado?

a = x++;

a = ++x;

5. ¿Cuál es el resultado de la expresión  $10 \% 3$ ?

6. ¿Cuál es el resultado de la expresión  $5 + 3 * 8 / 2$ ?

7. Reescriba la expresión de la pregunta 6, añadiendo paréntesis de tal forma que dé como resultado 16.

8. Si una expresión evalúa a falso, ¿qué valor tiene la expresión?

9. ¿Cuál tiene mayor precedencia?

a. == o <

b. \* o +

c. != o ==

d. >= o >

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





## Enunciados, expresiones y operadores

### Ejercicios

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

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

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

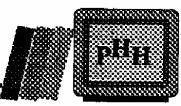
```
if (x < 1)
    if (x > 10)
        enunciado;
```

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

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

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

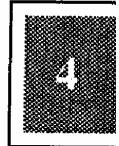
- a. `if (x == 4)`
- b. `if (x != y - z)`
- c. `if (z = 1)`
- d. `if (y)`



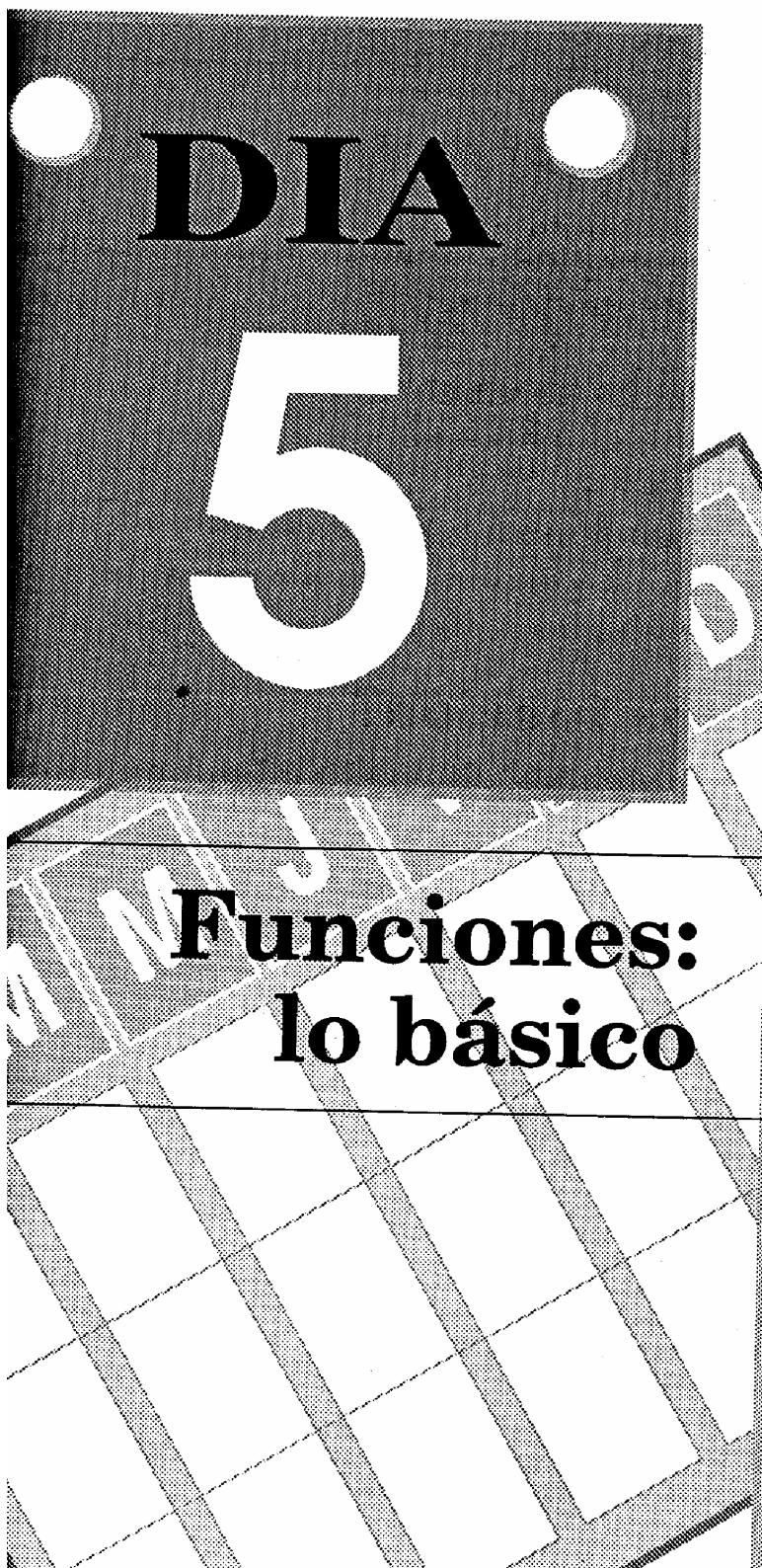
9. Escriba un enunciado `if` que determine si alguien es legalmente un adulto (edad 21 años), pero no un anciano (edad 65 años).

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

```
/* Un programa con problemas...*/  
#include <stdio.h>  
int x= 1:  
main()  
{  
    if( x = 1);  
    printf(" x equals 1" );  
otherwise  
    printf(" x does not equal 1");  
  
    return  
}
```



4





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

### Hoy aprenderá

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

## ¿Qué es una función?

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

### La definición de una función

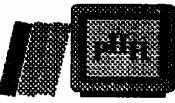
Primero la definición: una función es una sección de código de C que tiene un nombre, independiente, que ejecuta una tarea específica y que opcionalmente regresa un valor al programa que la llama. Ahora veamos las partes de esta definición.

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

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

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

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



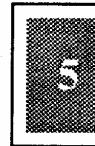
Esto es todo por lo que se refiere a la definición. Téngalo presente mientras ve la siguiente sección.

## La ilustración de una función

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

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

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



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

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

## Funciones: lo básico

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

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

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

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

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

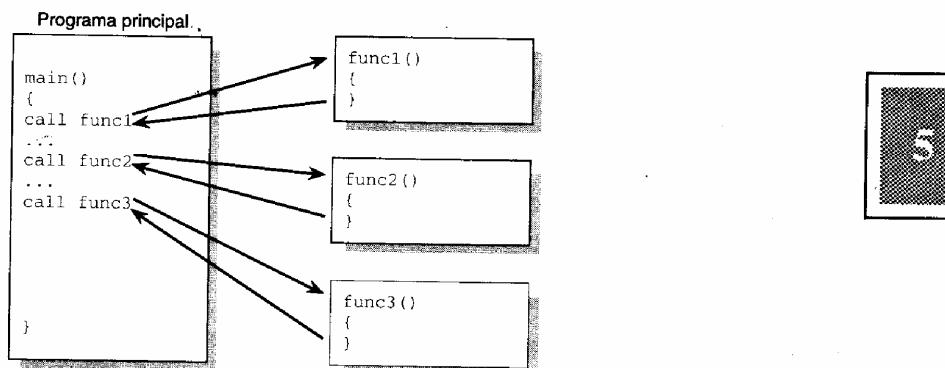
Si se compara la estructura de la función `cube()` con la de la función `main()` se verá que son la misma. `main()` es también una función. Otras funciones que ya se han usado son `printf()` y `scanf()`. Aunque `printf()` y `scanf()` son funciones de biblioteca (en vez de ser funciones definidas por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



## a manera en que abaja una función

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

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



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

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

## Funciones

### Prototipo de función

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

## Funciones: lo básico

### Definición de función

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

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

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

### Ejemplos de prototipo de función

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

### Ejemplos de definición de función

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

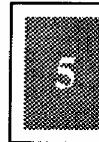
# Las funciones y la programación estructurada

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

## Las ventajas de la programación estructurada

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

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



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

## La planeación de un programa estructurado

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



## Funciones: lo básico

---

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

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

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

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

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

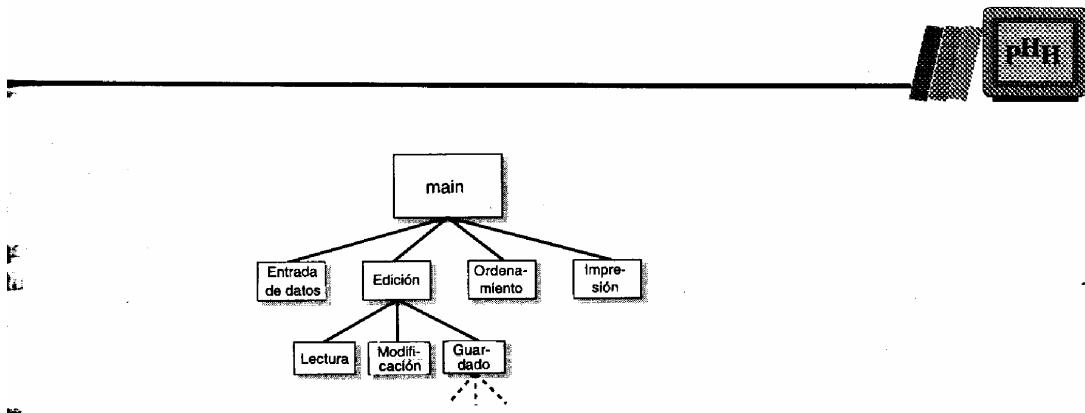
- Leer del disco la lista de direcciones existente.
- Modificar una o más entradas.
- Guardar la lista actualizada en el disco.

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

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

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

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



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

## El enfoque descendente

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

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

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

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



### DEBE

### NO DEBE

**DEBE** Planificar antes de comenzar a escribir el código. Comenzando con la determinación de la estructura del programa, se puede ahorrar tiempo en la escritura y depuración del código.

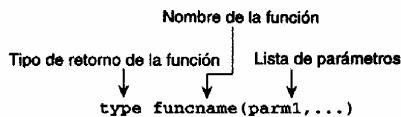
**NO DEBE** Tratar de hacer todo en una sola función. Una sola función debe hacer una sola tarea, como la lectura de información de un archivo.

## Escritura de una función

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

### El encabezado de la función

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



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

### El tipo de retorno de la función

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

```

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

### El nombre de la función

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

### La lista de parámetros

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

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

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

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

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

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

```
void func2(void)
```

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

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

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

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

#### Listado 5.2. La diferencia entre argumentos y parámetros.

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

## Funciones: lo básico

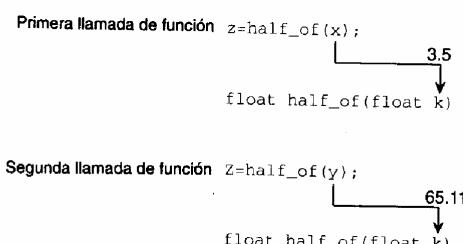
### Listado 5.2. continuación

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



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

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



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



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

**DEBE****NO DEBE**

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

**NO DEBE** Pasar valores innecesarios a una función.

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

## El cuerpo de la función

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

### Variables locales

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

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

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

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

## Funciones: lo básico

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



### Listado 5.3. Demostración de las variables locales.

```

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



```

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



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

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

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

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

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

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



## Enunciados de función

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

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

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

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

## Regreso de un valor

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

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

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

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

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

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

```

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

```

E:\>list0504
Enter two different integer values:
200 300
The larger value is 300.

E:\>list0504
Enter two different integer values:
300
200
The larger value is 300.
```

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

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

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

## El prototipo de la función

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

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

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

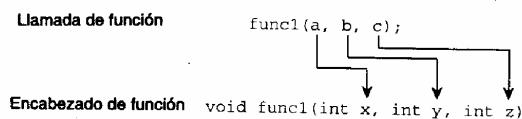
¿Dónde deben ponerse los prototipos de función en el código fuente? Deben ser puestos antes del inicio de `main()` o antes de que la función sea definida por primera vez. Para mejorar la legibilidad, lo mejor es agrupar todos los prototipos en una sola posición.

DEBE	NO DEBE
<b>NO DEBE</b> Tratar de regresar un valor que tiene un tipo diferente al tipo de la función.	
<b>DEBE</b> Usar variables locales siempre que sea posible.	
<b>NO DEBE</b> Dejar que las funciones se hagan muy largas. Si una función comienza a hacerse muy larga, trate de partirla en tareas más pequeñas.	
<b>DEBE</b> Limitar cada función a una sola tarea.	
<b>NO DEBE</b> Poner varios enunciados <code>return</code> si no son necesarios. Se debe tratar de <code>return</code> solo <code>return</code> cuando sea posible. Sin embargo, algunas veces el tener más de un <code>return</code> es más fácil y claro.	

## Asignación de argumentos a una función

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

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



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

5

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

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

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

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

## Llamado de funciones

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

```
wait(12);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

## DEBE

**DEBE** Pasar parámetros a las funciones en orden para hacer a las funciones genéricas, y por lo tanto, reutilizables!

**DEBE** Aprovechar la capacidad de poner funciones en expresiones.

**NO DEBE** Hacer confuso un enunciado individual poniendo muchas funciones en él. Sólo ponga funciones en los enunciados cuando no hagan el código más confuso.

## NO DEBE

## Recursión

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

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

$$x! = x * (x - 1) * (x - 2) * (x - 3) \dots * (2) * 1$$

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

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

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

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

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

## Funciones: lo básico



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

```

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



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



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



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

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

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

### DEBE

### NO DEBE

**DEBE** Comprender y trabajar con la recursión antes de usarla.

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



## ¿Dónde se ponen las funciones

- Tal vez se pregunte en qué parte del código fuente debe poner las definiciones de función.
- Por ahora deben ir en el mismo archivo de código fuente que `main()` y después del final de `main()`. La estructura básica de un programa que usa funciones se muestra en la figura 5.6.
- Se pueden guardar las funciones definidas por el usuario en un archivo del código fuente separado, separado de `main()`. Esta técnica es útil con programas grandes y cuando se quiere usar el mismo juego de funciones en más de un programa. Esta técnica se trata en el Día 21, “Aprovechando las directivas del preprocesador y más”.



## Funciones: lo básico

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

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

## Resumen

Este capítulo le presentó las funciones, que son una parte importante de la programación en C. Las funciones son secciones independientes de código que ejecutan tareas específicas. Cuando el programa necesita que se ejecute una tarea llama a la función que ejecuta esa tarea. El uso de funciones es esencial para la programación estructurada, un método de diseño de programa que enfatiza el enfoque modular descendente. La programación estructurada crea programas más eficientes y también más fáciles de usarse por uno, el programador.

También se aprendió que una función consiste en encabezado y cuerpo. Aquél incluye información acerca del tipo de retorno, nombre y parámetros de la función; éste, declaraciones de variables locales y los enunciados del C que se ejecutan cuando es llamada la función. Por último, se vio que las variables locales, aquellas declaradas dentro de una función, son completamente independientes de cualquier otra variable de programa declarada en cualquier otro lado.

## Preguntas y respuestas

1. ¿Qué pasa si necesito regresar más de un valor de una función?

Muchas veces necesitará regresar más de un valor de una función, o lo que es más común, deseará cambiar un valor que le es enviado a la función y guardar el



cambio después de que termine la función. Esto se trata en el Día 18, "Obteniendo más de las funciones".

2. ¿Cómo sé qué tan bueno es el nombre de una función?

Un buen nombre de función describe lo más específicamente posible lo que hace la función.

3. Cuando se declaran variables al principio del listado, antes de `main()`, pueden usarse en cualquier lugar, pero las variables locales sólo pueden usarse en la función específica. ¿Por qué no declarar todo antes de `main()`?

En el Día 12, "Alcance de las variables" se trata a mayor detalle el alcance de las variables.

4. ¿Qué otras formas hay de usar la recursión?

La función factorial es un primer ejemplo sobre el uso de la recursión. En muchos cálculos estadísticos se necesita el número del factorial. La recursión es simplemente un ciclo. Sin embargo, tiene una diferencia con respecto a otros ciclos. Con la recursión cada vez que es llamada una función recursiva se crea un nuevo juego de variables. Esto no es cierto en los otros ciclos que verá en el siguiente capítulo.

5. ¿Tiene que ser `main()` la primera función en un programa?

No. Es un estándar en C que la función `main()` sea la primera función que ejecute. Sin embargo, puede ser puesta en cualquier lugar del archivo fuente. La mayoría de la gente la pone primero para que sea fácil de localizar.



## Taller

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

## Cuestionario

1. ¿Va a usar programación estructurada cuando escriba sus programas en C?
2. ¿Cómo funciona la programación estructurada?
3. ¿Dónde entran las funciones del C en la programación estructurada?
4. ¿Cuál debe ser la primera línea de una definición de función y qué información contiene?
5. ¿Qué tantos valores puede regresar una función?

6. Si una función no regresa un valor, ¿con qué tipo debe ser declarada?
7. ¿Cuál es la diferencia entre una definición de función y un prototipo de función?
8. ¿Qué es una variable local?
9. ¿En qué son especiales las variables locales?

## Ejercicios

1. Escriba un encabezado para una función llamada `hazlo()`, que tome tres argumentos de tipo `char` y regrese un tipo `float` al programa que la llama.
2. Escriba un encabezado para una función llamada `imprime_un_número()`, que tome un solo argumento de tipo `int` y no regrese nada al programa que la llama.
3. ¿Qué tipo de valor regresan las siguientes funciones?
  - a. `int imprime_error( float num_error );`
  - b. `long lee_registro( int num_reg, int longitud );`
4. **BUSQUEDA DE ERRORES:** ¿Cuál es el error en el siguiente listado?

```
#include <stdio.h>
void print_msg( void );
main()
{
    print_msg( "This is a message to print" );
}

void print_msg( void )
{
    puts( "This is a message to print" );
    return 0;
}
```

5. **BUSQUEDA DE ERRORES:** ¿Cuál es el error en la siguiente definición de función?
 

```
int twice(int y);
{
    return (2 * y);
}
```
6. **Vuelva a escribir** el listado 5.4 de tal forma que sólo necesite un enunciado `return`.



7. Escriba una función que reciba dos números como argumentos y regrese el valor de su producto.
8. Escriba una función que reciba dos números como argumentos. La función debe dividir el primer número entre el segundo. No divida cuando el segundo número sea cero. (Consejo: Use un enunciado `if`.)
9. Escriba una función que llame a las funciones de los ejercicios 7 y 8.
10. Escriba un programa que use una función para encontrar el promedio de cinco valores tipo `float` tecleados por el usuario.
11. Escriba una función recursiva que calcule el valor de 3 a la potencia de otro número. Por ejemplo, si se le pasa 4, la función regresará 81.



DIA

6

Control básico  
del programa

## Control básico del programa

En el Día 4, “Enunciados, expresiones y operadores”, se trató al enunciado `if`, que da algo de control sobre el flujo de los programas. Sin embargo, muchas veces se necesita algo más que la simple habilidad de tomar decisiones sobre cierto o falso. Este capítulo presenta tres nuevas maneras de controlar el flujo del programa. Hoy aprenderá

- La manera de usar arreglos simples.
- La manera de usar ciclos `for`, `while` y `do...while` para ejecutar enunciados varias veces.
- Cómo se pueden anidar enunciados de control de programa.

Este capítulo no pretende dar un tratamiento completo de estos temas, pero sí suficiente información para que usted sea capaz de comenzar a escribir programas reales. Estos temas se tratan a mayor detalle en el Día 13, “Más sobre el control de programa”.

## Arreglos: lo básico

Antes de que tratemos al enunciado `for`, hagamos una pausa y aprendamos lo básico de los arreglos. (Véase el Día 8, “Arreglos numéricos” para una explicación a fondo de los arreglos.) El enunciado `for` y los arreglos están íntimamente relacionados en C, por lo que es difícil definir uno sin explicar el otro. Para ayudarle a comprender los arreglos, que se usan en los ejemplos del enunciado `for` que se presentan a continuación, se da una rápida explicación de los arreglos.

Un *arreglo* es un grupo indexado de ubicaciones de almacenamiento de datos que tienen el mismo nombre y se distinguen entre ellas por un *subíndice* o *índice*, un número que se pone a continuación del nombre de la variable encerrado entre corchetes. (Esto le quedará más claro conforme avance.) De manera similar a otras variables del C, los arreglos deben ser declarados. Una declaración de arreglo incluye tanto el tipo de dato como el tamaño de arreglo (la cantidad de elementos en el arreglo). Por ejemplo, el enunciado

```
int datos [1000];
```

declara a un arreglo llamado `datos` que tiene el tipo `int` y contiene 1,000 elementos. A los **elementos individuales** se hace referencia mediante subíndices, como `datos[0]` hasta `datos[999]`. El primer elemento es `datos[0]`, y no `datos[1]`. En otros lenguajes, como el **BASIC**, el primer elemento de un arreglo es 1, pero esto no es cierto en C.

Cada **elemento** de este arreglo es equivalente a una variable entera normal y puede ser usado **en la misma forma**. El subíndice de un arreglo puede ser otra variable del C, como en este **ejemplo**:

```
int datos[1000];
int contador;
contador = 100;
datos[contador] = 12;      /* Que es igual a datos[100] = 12 */
```



Esta ha sido una rápida introducción a los arreglos. Sin embargo, con esto debe ser capaz de comprender la manera en que se usan los arreglos en los ejemplos de programa que se encuentran posteriormente en este capítulo. Si todos los detalles de los arreglos no le han quedado claros, no se preocupe. Ya verá más acerca de los arreglos en el Día 8, "Arreglos numéricos".

### DEBE

### NO DEBE

**NO DEBE** Declarar arreglos con subíndices mayores de lo que necesita, ya que se desperdicia memoria.

**NO DEBE** Olvidar que en el C a los arreglos se les hace referencia comenzando con el subíndice 0 y no con 1.

## Control de la ejecución del programa

El orden por omisión de ejecución en un programa de C es descendente. La ejecución comienza al principio de la función `main()`, y avanza enunciado por enunciado hasta que se llega al final de `main()`. Sin embargo, este orden rara vez se encuentra en los programas de C reales. El lenguaje C incluye una variedad de enunciados para el control de programa, que le permiten controlar el orden de la ejecución del programa. Ya ha aprendido la manera de usar el operador fundamental de decisiones del C, el enunciado `if`, por lo que exploraremos tres enunciados de control adicionales que encontrará útiles.

### El enunciado `for`

El enunciado `for` es una construcción de programación del C que ejecuta un bloque de uno o más enunciados una determinada cantidad de veces. A veces es llamado el *ciclo for*, debido a que la ejecución del programa por lo general hace ciclos por los enunciados más de una vez. Ya ha visto unos cuantos enunciados `for`, que han sido usados en los ejemplos de programación anteriormente en este libro. Ahora se encuentra listo para ver la manera en que funciona el enunciado `for`.

Un enunciado `for` tiene la siguiente estructura:

```
for(inicial; condición; incremento)  
    enunciado
```

*inicial*, *condición* e *incremento* son expresiones del C, y *enunciado* es un enunciado simple o compuesto del C. Cuando se encuentra un enunciado `for` durante la ejecución del programa, suceden los siguientes eventos:



## Control básico del programa

1. La expresión *inicial* es evaluada. Lo *inicial* es por lo general un enunciado de asignación que pone una variable a un valor determinado.
2. La expresión de *condición* es evaluada. La *condición* es típicamente una expresión relacional.
3. Si la *condición* evalúa a falso (esto es, a cero), el enunciado *for* termina, y la ejecución pasa al primer enunciado que se encuentra a continuación del enunciado *del for*.
4. Si la *condición* evalúa a cierto (esto es, a diferente de cero), se ejecutan los enunciados que se encuentran dentro del *for*.
5. La expresión de *incremento* es evaluada y la ejecución regresa al paso dos.

La operación de un enunciado *for* se muestra esquemáticamente en la figura 6.1. Observe que el enunciado nunca ejecuta si la condición es falsa la primera vez que es evaluada.

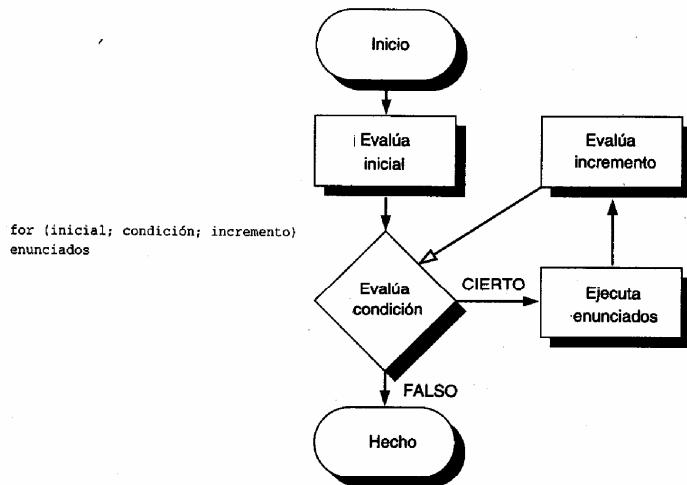


Figura 6.1. Representación esquemática de un enunciado *for*.

Aquí tenemos un ejemplo simple. El programa en el listado 6.1 usa un enunciado *for* para imprimir los números del 1 al 20. Puede ver que el código resultante es más compacto que como lo sería si fuera usado un enunciado *printf()* para cada uno de los 20 valores.

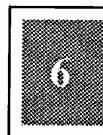


### Listado 6.1. Demostración de un enunciado for simple.

```
1: /* Demuestra un enunciado for simple */
2:
3: #include <stdio.h>
4:
5: int count;
6:
7: main()
8: {
9:     /* Imprime los números del 1 al 20 */
10:
11:    for (count = 1; count <= 20; count++)
12:        printf("\n%d", count);
13: }
```



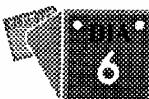
```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```



El diagrama en la figura 6.2 ilustra la operación del ciclo for en el listado 6.1.



La línea 3 incluye el archivo de encabezado de entrada/salida estándar. La línea 5 declara una variable de tipo int llamada count, que será usada en el ciclo for. Las líneas 11 y 12 son el ciclo for. Cuando se llega al enunciado for, se ejecuta primero el enunciado inicial. En este listado el enunciado inicial es count=1. Esto inicializa a count para que de esta forma pueda ser usado en el resto del ciclo. El segundo paso en la ejecución de este enunciado for es la evaluación de la condición count <= 20. Debido a que count acaba de ser inicializado a 1, se sabe que es menor que 20, por lo que el enunciado del comando for, printf(), se ejecuta. Después de ejecutar la función de impresión es evaluada la expresión de incremento, count++. Esto añade 1 a count haciendo que sea 2.



## Control básico del programa

Ahora el programa regresa y revisa nuevamente la condición. Si es cierta, vuelve a ejecutar el `printf()`, y el incremento suma a `count` (haciendo que sea 3) y la condición es revisada. Este ciclo continúa hasta que la condición evalúa a falso, y en este punto el programa sale del ciclo y continúa en la siguiente línea (línea 13), que en este listado da por terminado al programa.

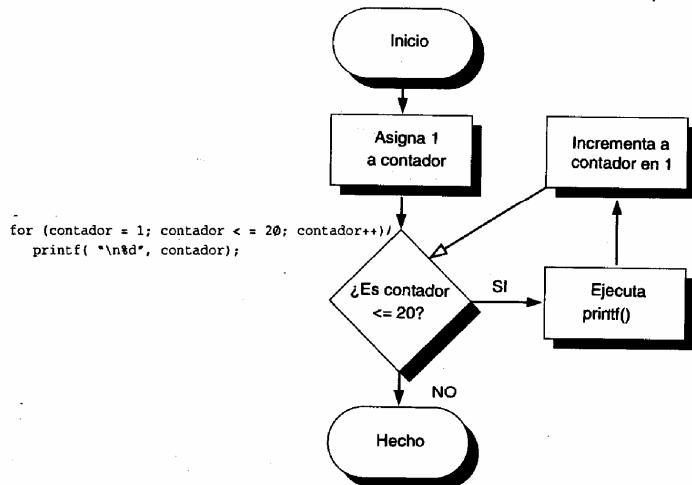


Figura 6.2. La manera en que funciona el ciclo for del listado 6.1.

El enunciado `for` es usado frecuentemente, como en el ejemplo anterior, para contar, incrementando un contador de un valor a otro. También se le puede usar para “contar al revés”, disminuyendo en vez de incrementar la variable del contador.

```
for (contador = 100; contador > 0; contador--)
```

También se puede incrementar en un valor diferente de 1.

```
for (contador = 0; contador < 1000; contador += 5)
```

El enunciado `for` es bastante flexible. Por ejemplo, se puede omitir la expresión de inicialización si la variable que se ha de probar ha sido inicializada anteriormente en el programa. (Sin embargo, todavía se debe usar el separador de punto y coma, como se muestra.)

```
contador = 1;  
for ( ; contador < 1000; contador++)
```

La expresión de inicialización no necesita ser de hecho una inicialización, sino que puede ser cualquier expresión válida del C. Sin importar lo que sea, se ejecuta una sola vez, cuando el enunciado `for` se ejecuta por primera vez. Por ejemplo, lo siguiente imprime “Ahora se ordena el arreglo...”



```
contador = 1;
for ( printf("Ahora se ordena el arreglo...") ; contador < 1000; contador++)
/* Aquí van los enunciados para el ordenamiento */
```

También se puede omitir la expresión de incremento, ejecutando la actualización en el cuerpo del enunciado `for`. Nuevamente debe ser incluido el punto y coma. Por ejemplo, para imprimir los números del 0 al 99, se puede escribir

```
for (contador = 0; contador < 100; )
printf("%d", contador++);
```

La expresión de prueba que hace que termine el ciclo puede ser cualquier expresión de C. Mientras evalúe a cierto (diferente de cero) el enunciado `for` continúa ejecutando. Se pueden usar los operadores lógicos del C para construir expresiones de prueba complejas. Por ejemplo, el siguiente enunciado `for` imprime los elementos de un arreglo llamado `arreglo[]`, deteniéndose cuando todos los elementos han sido impresos o se ha encontrado un elemento con un valor de 0.

```
for (contador = 0; contador < 1000 && arreglo[contador] != 0; contador++)
printf("%d", arreglo[contador]);
```

Se podría simplificar todavía más el ciclo `for` anterior, escribiéndolo de la manera siguiente. (Si no entiende los cambios hechos a las expresiones de prueba, necesita revisar el Día 4, "Enunciados, expresiones y operadores".)

```
for (contador = 0; contador < 1000 && arreglo[contador]; )
printf("%d", arreglo[contador++]);
```

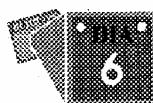
Se puede poner a continuación del enunciado `for` un enunciado nulo, haciendo que todo el trabajo se ejecute en el mismo enunciado `for`. Recuerde que el enunciado nulo es un punto y coma sólo en una línea. Por ejemplo, para inicializar todos los elementos de un arreglo de 1,000 elementos al valor 50, se podría escribir

```
for (contador = 0; contador < 1000; arreglo[contador++] = 50)
;
```

En este enunciado `for` el valor de 50 es asignado a cada miembro del arreglo por la parte de incremento del enunciado.

En el Día 4, "Enunciados, expresiones y operadores", se mencionó que el operador de coma del C es usado a veces en los enunciados `for`. Se puede crear una expresión separando dos subexpresiones con el operador de coma. Las dos subexpresiones son evaluadas (en orden de izquierda a derecha) y la expresión completa evalúa al valor de la subexpresión que se encuentra a la derecha. Usando al operador de coma se puede hacer que cada parte de un enunciado `for` ejecute varias tareas.

Imagine que tiene dos arreglos de 1,000 elementos, `a[]` y `b[]`. Se quiere copiar el contenido de `a[]` a `b[]` en orden inverso, de forma tal que después de la operación de copia `b[0] = a[999], b[1] = a[998]` y así sucesivamente. El siguiente enunciado `for` hace el truco:



## Control básico del programa

```
for ( i = 0, j = 999; i < 1000; i++, j-- )
    b[j] = a[i];
```

El operador de coma es usado para inicializar dos variables, *i* y *j*. También es usado en la parte de incremento para modificar las dos variables dentro de cada ciclo.

### Sintaxis

#### El enunciado *for*

```
for(inicial; condición; incremento)      enunciado(s)
```

Lo *inicial* es cualquier expresión válida del C. Por lo general, es un enunciado de asignación, que pone una variable a un valor determinado.

La *condición* es cualquier expresión válida del C. Por lo general, es una expresión relacional. Cuando la *condición* evalúa a falso (0) termina el enunciado *for*, y la ejecución pasa al primer enunciado que se encuentra después del enunciado del *for*. En caso contrario se ejecutan los enunciados del *for*.

El *incremento* es cualquier expresión válida del C. Por lo general es una expresión que incrementa una variable que ha sido inicializada por la expresión *inicial*.

Los enunciados son los enunciados que se ejecutan mientras la condición permanezca cierta.

Un enunciado *for* es un enunciado de ciclo. Puede tener una inicialización, una prueba de condición y un incremento como parte del comando. El enunciado *for* ejecuta primero la expresión *inicial*. Luego revisa la condición, y en caso de que sea cierta se ejecutan los enunciados. Una vez que los enunciados se terminan, es evaluada la expresión de *incremento*. El enunciado *for* vuelve a revisar entonces la condición, y continúa haciendo ciclo hasta que la condición es falsa.

#### Ejemplo 1

```
/* Imprime el valor de x al tiempo en que cuenta de 0 a 9 */
int x;
for( x = 0; x < 10; x++ )
    printf( "\nEl valor de x es %d", x );
```

#### Ejemplo 2

```
/* Pide cifras al usuario hasta que se teclea 99 */
int num = 0;
for( ; num != 99; )
    scanf( "%d", &num );
```

#### Ejemplo 3

```
/* Permite que el usuario teclee hasta 10 valores enteros.      */
/* Los valores son guardados en un arreglo llamado valor.      */
/* Si se teclea 99 el ciclo se detiene                         */
/*                                                               */
/*                                                               */
```

```
int valor[10];
int contador, numero=0;
for ( contador = 0; contador < 10 && numero != 99; contador++)
{
    puts ( "Teclee un número, 99 para terminar ");
    scanf ( "%d", &numero );
    valor[contador] = numero;}
```

## Enunciados for anidados

Un enunciado `for` puede ser ejecutado dentro de otro enunciado `for`. A esto se le llama *anidado*. (Ya se vio esto en el Día 4, “Enunciados, expresiones y operadores”, dentro del enunciado `if`.) Anidando enunciados `for` se puede hacer programación compleja. El listado 6.2 no es un programa complejo, pero ilustra el anidado de dos enunciados `for`.

Captura

### Listado 6.2. Demostración de enunciados for anidados.

```
1: /* Demuestra el anidado de dos enunciados for */
2:
3: #include <stdio.h>
4:
5: void draw_box( int row, int column);
6:
7: main()
8: {
9:     draw_box( 8, 35 );
10: }
11:
12: void draw_box( int row, int column )
13: {
14:     int col;
15:     for( ; row > 0; row-- )
16:     {
17:         for(col = column; col > 0; col-- )
18:             printf( "X" );
19:
20:         printf( "\n" );
21:     }
22: }
```

Salida

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

## Control básico del programa

### ANÁLISIS

El trabajo principal de este programa se realiza en la línea 18. Cuando se ejecuta este programa se imprimen 280 X en la pantalla, formando un cuadro de 8 por 35. El programa tiene solamente un comando para imprimir una X, pero se encuentra anidado en dos ciclos.

En este listado se declara en la línea 5 el prototipo de función para `draw_box()`. Esta función toma dos variables de tipo `int`, `row` y `column`, que contienen las dimensiones del cuadro de X que será trazado. En la línea 9 `main()` llama a `draw_box()`, y le pasa 8 como el valor de `row` y 35 como el valor de `column`.

Viendo detalladamente la función `draw_box()` se pueden ver unas cuantas cosas que no se entienden fácilmente. La primera es por qué se declara la variable local `col`. La segunda es por qué se usa el segundo `printf()` en la línea 20. Ambas cosas se verán más claras después de observar los dos ciclos `for`.

En la línea 15 comienza el primer ciclo `for`. No se hace la inicialización, debido a que el valor inicial de `row` fue pasado a la función. Viendo la condición se ve que este ciclo `for` se ejecuta hasta que `row` es igual a 0. Al ejecutar por primera vez la línea 15 `row` vale 8, y por lo tanto el programa continúa a la línea 17.

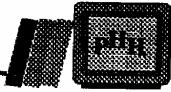
La línea 17 contiene el segundo enunciado `for`. Aquí el parámetro que se ha pasado, `column`, es copiado a una variable local, `col`, de tipo `int`. El valor de `col` es inicialmente 35 (el valor pasado por medio de `column`) y `column` conserva su valor original. Debido a que `col` es mayor que cero, se ejecuta la línea 18, imprimiendo una X. Luego `col` es decrementado y el ciclo continúa. Cuando `col` es 0 el ciclo `for` termina y el control pasa a la línea 20. La línea 20 hace que se imprima en la pantalla el comienzo de una nueva línea. (En el Día 7, "Entrada/salida básica", se trata a detalle la impresión). Despues de moverse a una nueva línea en la pantalla el control llega al final de los enunciados del primer ciclo `for`, y por lo tanto se ejecuta la expresión de incremento que resta 1 de `row`, haciendo que sea 7. Esto regresa el control a la línea 17. Observe que el valor de `col` fue 0 la última vez que se usó. Si se hubiera usado `column` en vez de `col`, hubiera fallado la prueba de la condición, debido a que nunca sería mayor que 0. Solamente la primera línea sería impresa. Quite la inicialización de la línea 17 y cambie las dos variables `col` a `column` para ver lo que pasa realmente.

### DEBE

### NO DEBE

**NO DEBE** Poner mucho procesamiento en el enunciado `for`. Aunque se puede usar el separador coma, por lo general es más claro poner algo del funcionamiento en el cuerpo del ciclo.

**DEBE** Recordar usar el punto y coma si se usa un `for` con un enunciado nulo. Ponga el punto y coma en una linea aparte, o ponga un espacio entre él y el final del enunciado `for`.



```
for( contador = 0; contador < 1000; arreglo[contador] = 50) ;  
/* observe el espacio! */
```

## El enunciado *while*

El enunciado *while*, también llamado el *ciclo while*, ejecuta un bloque de enunciados en tanto una condición específica sea cierta. El enunciado *while* tiene la siguiente forma:

```
while (condición)  
    enunciado
```

Esta *condición* es cualquier expresión de C y el *enunciado* es un enunciado del C, simple o compuesto. Cuando la ejecución del programa llega al enunciado *while* suceden los siguientes eventos:

1. Es evaluada la expresión de la *condición*.
2. Si la *condición* evalúa a falso (esto es, a cero), el enunciado *while* termina, y la ejecución pasa al primer enunciado que se encuentre a continuación de los *enunciados del while*.
3. Si la *condición* evalúa a cierto (esto es, diferente de cero), se ejecutan los *enunciados C del while*.
4. La ejecución regresa al paso uno.

La operación de un enunciado *while* se encuentra diagramada en la figura 6.3.

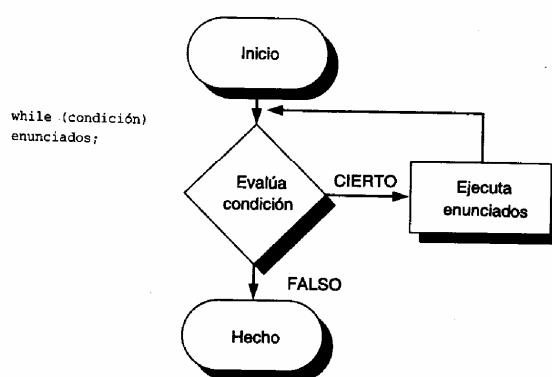


Figura 6.3. Representación esquemática de la operación de un enunciado *while*.

## Control básico del programa

El listado 6.3 es un programa simple que usa un enunciado `while` para imprimir los números del 1 al 20. (Esta es la misma tarea que se ejecuta por un enunciado `for` en el listado 6.1.)



### Listado 6.3. Demostración de un enunciado `while` simple.

```

1: /* Demuestra un enunciado while simple */
2:
3: #include <stdio.h>
4:
5: int count;
6:
7: main()
8: {
9:     /* Imprime los números del 1 al 20 */
10:    count = 1;
11:
12:    while (count <= 20)
13:    {
14:        printf("\n%d", count);
15:        count++;
16:    }
17: }
```



```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```



Examine el listado 6.3 y compárelo con el listado 6.1, que usa un enunciado `for` para ejecutar la misma tarea. En la línea 11 `count` es inicializado a 1. Debido a que el enunciado `while` no contiene una sección de inicialización, uno debe hacerse cargo de la inicialización de cualquier variable antes de comenzar el `while`. La línea 13 es el enunciado



while actual, y contiene el mismo enunciado condicional que el listado 6.1, count  $\leq 20$ . En el ciclo while la línea 16 se ocupa de incrementar a count. ¿Qué cree que pasaría si se le olvida poner la línea 16 en el programa? El programa no sabría cuándo parar, debido a que count siempre sería 1, que es siempre menor que 20.

Tal vez se haya dado cuenta de que un enunciado while es esencialmente un enunciado for sin los componentes de inicialización e incremento. Por lo tanto,

```
for ( ; condición ; )  
es equivalente a  
while (condición)
```

Debido a esta equivalencia, cualquier cosa que pueda ser hecha con un enunciado for también puede ser hecha con un enunciado while. Cuando se usa un enunciado while se debe hacer primero cualquier inicialización que se necesite en un enunciado separado, y la actualización debe ser realizada por un enunciado que sea parte del ciclo while.

Cuando son requeridas la inicialización y la actualización, la mayoría de los programadores de C con experiencia prefieren usar un enunciado for en vez de un enunciado while. Esta preferencia se basa, en primer lugar, sobre la legibilidad del código fuente. Cuando se usa un enunciado for las expresiones de inicialización e incremento se ubican juntas, y son fáciles de encontrar y modificar. En un enunciado while las expresiones de inicialización y de actualización se encuentran ubicadas por separado y pueden ser menos obvias.

## Sintaxis

### El enunciado while

```
while( condición )  
    enunciado(s)
```

La condición es cualquier expresión válida del C y por lo general es una expresión relacional. Cuando la condición evalúa a falso (cero) el enunciado while termina, y la ejecución pasa al primer enunciado que se encuentra a continuación de los enunciados del while. En caso contrario se ejecutan los enunciados C que se encuentran en el while.

Los enunciados son los enunciados C que se ejecutan siempre y cuando la condición sea cierta.

Un enunciado while es un enunciado de ciclo del C. El permite la ejecución repetida de un enunciado o de un bloque de enunciados en tanto la condición permanezca cierta (diferente de cero). Si la condición no es cierta cuando el comando while se ejecuta por primera vez, los enunciados nunca se ejecutan.

### Ejemplo 1

```
int x = 0;  
while( x < 10 )
```

## Control básico del programa

```
{  
    printf( "\nEl valor de x es %d", x );  
    x++;  
}
```

### Ejemplo 2

```
/* Pide cifras al usuario hasta que se teclea 99 */  
int num = 0;  
while( num <= 99 )  
    scanf( "%d", &num );  
    num++
```

### Ejemplo 3

```
/* Permite que el usuario teclee hasta 10 valores enteros. */  
/* Los valores son guardados en un arreglo llamado valor. */  
/* Si se teclea 99 el ciclo se detiene. */  
int valor[10];  
int contador = 0;  
int numero;  
while ( contador < 10 && numero != 99 )  
{  
    puts ( "Teclee un número, 99 para terminar" );  
    scanf ( "%d", &numero );  
    valor[contador] = numero;  
    contador++;  
}
```

## Enunciados while anidados

De manera similar a los enunciados `for` e `if`, los enunciados `while` también pueden anidarse. El listado 6.4 muestra un ejemplo de enunciados `while` anidados. Aunque éste no es el mejor uso de un enunciado `while`, el ejemplo presenta algunas ideas nuevas.

### Captura

#### Listado 6.4. Demostración de enunciados while anidados.

```
1:  /* Demuestra enunciados while anidados */  
2:  
3:  #include <stdio.h>  
4:  
5:  int array[5];  
6:  
7:  main()  
8:  {  
9:      int ctr = 0,  
10:         nbr = 0;  
11:  
12:     printf( "This program prompts you to enter 5 numbers\n" );  
13:     printf( "Each number should be from 1 to 10\n" );  
14:  
15:     while ( ctr < 5 )  
          0 L5
```

```
16:      {
17:          nbr = 0;
18:          while ( nbr < 1 || nbr > 10 ) ,
19:          {
20:              printf( "\nEnter number %d of 5: ", ctr + 1 );
21:              scanf( "%d", &nbr );
22:          }
23:          array[ctr] = nbr;
24:          ctr++;
25:      }
26:  }
27:
28: for( ctr = 0; ctr < 5; ctr++ )
29:     printf( "\nValue %d is %d", ctr + 1, array[ctr] );
30: }
```



This program prompts you to enter 5 numbers  
Each number should be from 1 to 10  
Enter number 1 of 5: 3  
Enter number 2 of 5: 6  
Enter number 3 of 5: 3  
Enter number 4 of 5: 9  
Enter number 5 of 5: 2  
Value 1 is 3  
Value 2 is 6  
Value 3 is 3  
Value 4 is 9  
Value 5 is 2



De manera similar a los listados anteriores, la línea 1 contiene un comentario con una descripción del programa, y la línea 3 contiene un enunciado `#include` para el archivo de encabezado de entrada/salida estándar. La línea 5 contiene una declaración para un arreglo (llamado `array`) que puede guardar cinco valores enteros. La función `main()` contiene dos variables locales adicionales, `ctr` y `nbr` (líneas 9 y 10). Observe que estas variables son inicializadas a cero al mismo tiempo que son declaradas. También observe que es usado el operador coma como separador al final de la línea 9, permitiendo que `nbr` sea declarado como `int` sin volver a poner el comando de tipo `int`. Dar declaraciones de esta forma es una práctica común de muchos programadores de C. Las líneas 11 y 12 imprimen mensajes, diciendo lo que el programa hace y lo que se espera del usuario. Las líneas 15 a 26 contienen el primer comando `while` y sus enunciados. Las líneas 18 a 22 también contienen un ciclo `while` anidado con sus propios enunciados, que son todos parte del `while` exterior.

Este ciclo exterior continúa ejecutando mientras `ctr` sea menor a 5 (línea 15). En tanto `ctr` sea menor que 5, la línea 17 pone a `nbr` a 0, con las líneas 18 a 22 (el enunciado `while` anidado) se obtiene un número en la variable `nbr`, la línea 24 pone el número en el arreglo y la línea 25 incrementa a `ctr`. Luego el ciclo vuelve a comenzar. Por lo tanto, el ciclo exterior obtiene cinco números, y pone a cada uno en `array` indexado por `ctr`.



## Control básico del programa

El ciclo interno es un buen uso del enunciado `while`. Sólo son válidos los números del 1 al 10, por lo que mientras que el usuario teclee un número válido no tiene caso continuar con el programa. Las líneas 18 a 22 previenen la continuación. Este enunciado `while` establece que mientras el número sea menor que 1 o mientras sea mayor que 10, el programa debe imprimir un mensaje para pedir el número y luego obtenerlo.

Las líneas 28 y 29 imprimen los valores que se encuentran guardados en `array`. Observe que debido a que los enunciados `while` han terminado de usar la variable `ctr`, el comando `for` puede reutilizarla. Comenzando en cero e incrementándola de uno en uno, el `for` hace ciclo cinco veces, imprimiendo el valor de `ctr` más uno (debido a que el contador comenzó en cero), e imprimiendo el valor correspondiente de `array`.

Como práctica adicional hay dos cosas que se pueden cambiar en este programa. La primera son los valores que son aceptados por el programa. En vez de 1 a 10 trate de hacer que acepte de 1 a 100. También puede cambiar la cantidad de valores que acepta. Actualmente acepta 5 números. Trate de que acepte 10.

### DEBE

### NO DEBE

**NO DEBE** Usar la siguiente convención si no es necesario.

```
while( x )
```

En vez de ello, use:

```
while( x != 0 )
```

Aunque ambas funcionan, la segunda es más clara cuando se está depurando el código. Cuando se compila producen virtualmente el mismo código.

**DEBE** Usar el enunciado `for` en vez del enunciado `while` si necesita inicializar e incrementar dentro del ciclo. El enunciado `for` mantiene juntos los enunciados de inicialización, condición e incremento. El enunciado `while` no lo hace.

## El ciclo `do...while`

La tercera construcción de ciclo del C es el ciclo `do...while`, que ejecuta un bloque de enunciados mientras una condición específica sea cierta. El ciclo `do...while` prueba la condición al final del ciclo en vez de hacerlo al principio, como es hecho por el ciclo `for` y por el ciclo `while`.

La estructura del ciclo `do...while` es la siguiente:

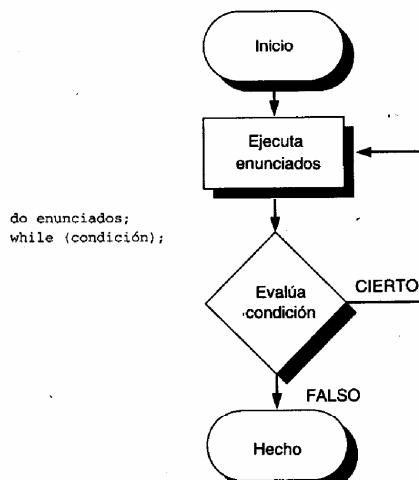
```
do  
    enunciado  
    while (condición);
```

La *condición* es cualquier expresión del C, y el *enunciado* es un enunciado simple o compuesto del C.

Cuando la ejecución del programa llega a un enunciado `do...while` suceden los siguientes eventos:

1. Se ejecutan los enunciados que se encuentran en *enunciado*.
2. La *condición* es evaluada. Si es cierta, la ejecución regresa al paso 1. Si es falsa el ciclo termina.

La operación de un ciclo `do...while` se muestra esquemáticamente en la figura 6.4.



**Figura 6.4.** La operación de un ciclo `do...while`.

Los enunciados asociados con un ciclo `do...while` son siempre ejecutados por lo menos una vez. Esto se debe a que la condición es evaluada al final, en vez de al principio del ciclo. Por el contrario, los ciclos `for` y `while` evalúan la condición al inicio del ciclo y, por lo tanto, los enunciados asociados no se ejecutan si la condición es falsa al inicio.

El ciclo `do...while` es usado menos frecuentemente que los ciclos `while` y `for`. Es el más adecuado cuando los enunciados asociados con el ciclo deben ser ejecutados por lo menos una vez. Por supuesto que es posible lograr lo mismo con un ciclo `while`, asegurándose de que la condición sea cierta cuando la ejecución llegue al ciclo por primera vez. Sin embargo, probablemente sea más directo un ciclo `do...while`.



## Control básico del programa

El listado 6.5 muestra un ejemplo de un ciclo do...while.



### Listado 6.5. Demostración de un enunciado do...while simple.

```
1: /* Demuestra un enunciado do...while simple */
2:
3: #include <stdio.h>
4:
5: int get_menu_choice( void );
6:
7: main()
8: {
9:     int choice;
10:
11:    choice = get_menu_choice();
12:
13:    printf( "You chose Menu Option %d", choice );
14: }
15:
16: int get_menu_choice( void )
17: {
18:     int selection = 0;
19:
20:     do
21:     {
22:         printf( "\n" );
23:         printf( "\n1 - Add a Record" );
24:         printf( "\n2 - Change a record" );
25:         printf( "\n3 - Delete a record" );
26:         printf( "\n4 - Quit" );
27:         printf( "\n" );
28:         printf( "\nEnter a selection:" );
29:
30:         scanf( "%d", &selection );
31:
32:     }while ( selection < 1 || selection > 4 );
33:
34:     return selection;
35: }
```



- 1 - Add a Record
- 2 - Change a record
- 3 - Delete a record
- 4 - Quit

Enter a selection:8

```

1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit

Enter a selection:4
You chose Menu Option 4

```

**Analisis**

Este programa proporciona un menú con cuatro alternativas. El usuario selecciona una de ellas, y luego el programa imprime el número seleccionado. Algunos programas que se encuentran posteriormente en este libro usan y amplían este concepto. Por ahora deberá ser capaz de seguir la mayor parte del listado. La función `main()` (líneas 7-14) no añaden nada a lo que ya sabe. Todo lo de `main()` pudiera haber sido escrito en una sola línea.

```
printf( "You chose Menu Option %d", get_menu_option() );
```

Si se fuera a ampliar este programa y actuar sobre la selección, tal vez necesitaría el valor regresado por `get_menu_choice()`, por lo que conviene asignar el valor a una variable (como `choice`).

Las líneas 16 a 35 contienen a `get_menu_choice()`. Esta función despliega un menú en la pantalla (líneas 22 a 28) y luego obtiene una selección. Debido a que se tiene que desplegar un menú por lo menos una vez para obtener una respuesta, es adecuado usar un ciclo `do...while`. En el caso de este programa, el menú es desplegado hasta que se da una selección válida. La línea 32 contiene la parte `while` del enunciado `do...while` y valida el valor de la selección que, adecuadamente, es llamado `selection`. Si el valor dado no se encuentra entre 1 y 4, el menú se vuelve a desplegar y se le pide al usuario un nuevo valor. Cuando se da una selección válida el programa continúa a la línea 34, la cual regresa el valor en la variable `selection`.

**Sintaxis****El enunciado `do...while`**

```

do
{
    enunciado(s)
}while( condición );

```

La `condición` es cualquier expresión de C válida y, por lo general, es una expresión relacional. Cuando la `condición` evalúa a falso (cero) el enunciado `while` termina, y la ejecución pasa al primer enunciado que se encuentra a continuación del enunciado `while`. En caso contrario el programa regresa al `do` para repetir el ciclo, y se ejecutan los enunciados del C que se encuentran en `enunciado(s)`.

Los `enunciado(s)` son un enunciado simple del C o un bloque de enunciados, que se ejecutan la primera vez que se pasa por el ciclo y luego mientras la `condición` permanece cierta.



## Control básico del programa

Un enunciado `do...while` es un enunciado de ciclo del C. El permite la ejecución repetida de un enunciado o de un bloque de enunciados mientras la condición se mantenga cierta (diferente a cero). A diferencia del enunciado `while`, un ciclo `do...while` ejecuta sus enunciados por lo menos una vez.

### Ejemplo 1

```
/* imprime aunque la condición falle */
int x = 10;
do
{
    printf( "\nEl valor de x es %d", x );
}while( x != 10 );
```

### Ejemplo 2

```
/* obtiene números hasta que el número es mayor que 99 */
int num;
do
{
    scanf( "%d", &num );
}while( num <= 99 );
```

### Ejemplo 3

```
/* Le permite al usuario dar hasta 10 valores enteros.      */
/* Los valores son guardados en un arreglo llamado valor.  */
/* Si se da 99 el ciclo termina                           */
int valor[10];
int contador = 0;
int num;
do
{
    puts( "Teclee un número, 99 para terminar" );
    scanf( "%d", &num );
    valor[contador] = num;
    contador++;
}while( contador < 10 && num != 99 );
```

## Ciclos anidados

El término *ciclo anidado* se refiere a un ciclo que se encuentra dentro de otro ciclo. Ya ha visto ejemplos de algunos enunciados anidados. El C no pone limitación sobre el anidado de los ciclos, a excepción de que cada ciclo interno debe estar encerrado completamente en el ciclo externo. No se pueden tener ciclos traslapados. Por lo tanto, lo siguiente no es permitido:

```
for ( contador = 1; contador < 100; contador++)
{
    do
```

```

{
    /* el ciclo do...while */
} /* fin del ciclo for */
} while (x != 0);
Si el ciclo do...while es puesto completamente dentro del ciclo for, no hay
problemas.
for ( contador = 1; contador < 100; contador++)
{
    do
    {
        /* el ciclo do...while */
    }while (x != 0);
} /* fin del ciclo for */

```

Cuando use ciclos anidados recuerde que los cambios que se hacen en el ciclo interior también pueden afectar al ciclo exterior. En el ejemplo anterior, si el ciclo interior do...while modifica el valor de contador, la cantidad de veces que ejecuta el ciclo for exterior es afectada. Sin embargo, observe que el ciclo interior puede ser independiente de cualquier variable del ciclo exterior. En este ejemplo no lo son.

El buen estilo de sangrado hace que el código con ciclos anidados sea fácil de leer. Cada nivel de ciclo debe ser indentado un paso más que el nivel anterior. Esto etiqueta claramente al código asociado con cada ciclo.

## DEBE

## NO DEBE

**NO DEBE** Tratar de traslapar ciclos. Los puede anidar pero deben estar completamente dentro del otro.

**DEBE** Usar el ciclo do...while cuando sepa que un ciclo debe ejecutar por lo menos una vez.

6

## Resumen

Ahora está casi listo para comenzar a escribir programas en C reales por su cuenta.

El C tiene tres enunciados de ciclo que controlan la ejecución del programa: `for`, `while` y `do...while`. Cada una de estas construcciones le permiten al programa ejecutar un bloque de enunciados cero, una o más veces, basado en la condición de determinada variable de programa. Muchas tareas de programación se adaptan a la ejecución repetitiva que permiten estos enunciados de ciclo.

Aunque los tres pueden usarse para realizar la misma tarea, cada uno es diferente. El enunciado `for` le permite inicializar, evaluar e incrementar en un solo comando. El enunciado



## Control básico del programa

`while` funciona mientras una condición es cierta. El enunciado `do...while` siempre ejecuta sus instrucciones por lo menos una vez, y continúa ejecutándolas hasta que una condición es falsa.

El anidado es poner un comando dentro de otro. El C le permite el anidado de cualquiera de sus comandos. El anidado de enunciados `if` fue demostrado en el Día 4, "Enunciados, expresiones y operadores". En este capítulo fueron anidados los enunciados `for`, `while` y `do...while`.

## Preguntas y respuestas

1. ¿Cómo sé qué enunciado de control de programa debo usar, el `for`, el `while` o el `do...while`?

Si observa los cuadros de sintaxis que se proporcionan, podrá ver que cualquiera de los tres puede ser usado para resolver un problema de ciclo. Sin embargo, cada uno de ellos tiene una forma particular de hacerlo. El enunciado `for` es el mejor cuando se sabe que se necesita inicializar e incrementar en el ciclo. Si solamente se tiene una condición que se quiere satisfacer, y no se está manejando una cantidad específica de ciclos, el `while` es una buena alternativa. Si se sabe que un juego de enunciados necesita ser ejecutado por lo menos una vez, un `do...while` puede ser lo mejor. Debido a que los tres pueden usarse para la mayoría de los problemas, lo mejor es aprenderlos todos y luego evaluar cada situación de programación para determinar cuál es el más adecuado.

2. ¿A qué tanta profundidad puedo anidar mis ciclos?

Puede anidar tantos ciclos como desee. Sin embargo, si el programa requiere más de dos ciclos, considere el uso de una función en vez de ellos. Tal vez encuentre difícil seguir la pista por todas estas llaves, y una función es más fácil de seguir en el código.

3. ¿Puedo anidar diferentes comandos de ciclo?

Se pueden anidar comandos `if`, `for`, `while`, `do...while` o cualquier otro. Encontrará que muchos de los programas que trate de escribir requerirán que anide por lo menos unos cuantos de éstos.

## Taller

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

## Cuestionario

1. ¿Cuál es el valor de índice del primer elemento de un arreglo?
2. ¿Cuál es la diferencia entre un enunciado `for` y un enunciado `while`?
3. ¿Cuál es la diferencia entre un enunciado `while` y un enunciado `do...while`?
4. ¿Es cierto que un enunciado `while` puede ser usado y obtener todavía los mismos resultados que al codificar un enunciado `for`?
5. ¿Qué debe recordarse cuando se anidan enunciados?
6. ¿Puede un enunciado `while` ser anidado en un enunciado `do...while`?

## Ejercicios

1. Escriba una declaración para un arreglo que guarde 50 valores de tipo `long`.
2. Muestre un enunciado que asigne el valor de 123.456 al quincuagésimo elemento del arreglo del ejercicio 1.

3. ¿Cuál es el valor de `x` cuando se termina el siguiente enunciado?

```
for( x = 0; x < 100, x++ );
```

4. ¿Cuál es el valor de contador cuando se termina el siguiente enunciado?

```
for( contador = 2; contador < 10; contador +=3 ) ;
```

5. ¿Qué tantas X imprime lo siguiente?

```
for( x = 0; x < 10; x++ )  
    for( y = 5; y > 0; y-- )  
        puts( "X" );
```

6. Escriba un enunciado `for` para contar de 1 a 100 en incrementos de 3.

7. Escriba un enunciado `while` para contar de 1 a 100 en incrementos de 3.

8. Escriba un enunciado `do...while` para contar de 1 a 100 en incrementos de 3.

9. BUSQUEDA DE ERRORES: ¿Qué está equivocado en el siguiente fragmento de código?

```
record = 0;  
while( record < 100 )  
{
```

6

```
    printf( "\nRecord %d ", record );
    printf( "\nGetting next number..." );
}
```

10. BUSQUEDA DE ERRORES: ¿Qué está equivocado en el siguiente fragmento de código? (¡MAXVALUES no es el problema!)

```
for ( counter = 1; counter < MAXVALUES; counter++ );
    printf( "\nCounter = %d", counter );
```



**Entrada/salida  
básica**



## Entrada/salida básica

En la mayoría de los programas que se crean se necesita desplegar información en la pantalla o leer información del teclado. Muchos de los programas que se han presentado en los capítulos anteriores han realizado estas tareas, pero tal vez no lo haya comprendido exactamente. Hoy aprenderá

- Algo acerca de los enunciados de entrada y salida del C.
- La manera de desplegar información en la pantalla con las funciones de biblioteca `printf()` y `puts()`.
- La manera de formatear la información que es desplegada en la pantalla.
- Cómo leer datos del teclado con la función de biblioteca `scanf()`.

Este capítulo no pretende dar un tratamiento completo a estos temas, sino solamente proporcionar la información suficiente para que de esta forma se pueda comenzar a escribir programas reales. Estos temas se tratan a mayor detalle posteriormente en el libro.

# Desplegado de la información en la pantalla

En la mayoría de los programas usted necesitará desplegar información en la pantalla. La manera más común de hacer esto es usando las funciones de biblioteca del C, `printf()` y `puts()`.

## La función `printf()`

La función `printf()`, que es parte de la biblioteca estándar del C, es probablemente la manera más versátil de que dispone un programa para desplegar datos en la pantalla. Ya ha visto que se ha usado `printf()` en muchos de los ejemplos de este libro. Ahora necesita saber la manera en que trabaja la función `printf()`.

Imprimir un mensaje de texto en la pantalla es simple. Llame a la función `printf()`, pasándole el mensaje deseado entre comillas dobles. Por ejemplo, para desplegar ¡Ha ocurrido un error! se escribe

```
printf("¡Ha ocurrido un error!");
```

Sin embargo, además de mensajes de texto frecuentemente se necesita desplegar el valor de las variables del programa. Esto es un poco más complicado que desplegar solamente un mensaje. Por ejemplo, supongamos que se quiere desplegar el valor de la variable numérica `x` en la pantalla, junto con algún texto de identificación. Es más, se quiere que la información comience al principio de una nueva línea. Se puede usar la función `printf()` de la manera siguiente:



```
printf("\nEl valor de x es %d", x);  
y el desplegado en la pantalla, suponiendo que el valor de x sea 12, sería  
El valor de x es 12
```

En este ejemplo se le pasan dos argumentos a `printf()`. El primer argumento se encuentra encerrado entre comillas dobles y es llamado el *formato*. El segundo argumento es el nombre de la variable (`x`) que contiene el valor que va a ser impreso.

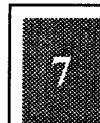
Un formato de `printf()` especifica la manera en que se formatea la salida. Los tres posibles componentes de un formato son los siguientes:

- El texto literal es desplegado exactamente igual a como se dio en el formato. En el ejemplo, los caracteres que comienzan con la E (de El) y hasta, pero sin incluir el %, comprenden el texto literal.
- Una *secuencia de escape* proporciona control especial del formateo. Una secuencia de escape consiste en la diagonal inversa (\) seguida de un solo carácter. En el ejemplo anterior \n es la secuencia de escape. Este es llamado el carácter de *nueva línea* y significa "moverse al inicio de la nueva línea". Las secuencias de escape también se usan para imprimir determinados caracteres. En la tabla 7.1 se listan más secuencias de escape.

46 7

Tabla 7.1. Las secuencias de escape más frecuentemente usadas.

Secuencia	Significado
\a	Campana (alerta)
\b	Retroceso
\n	Nueva línea
\t	Tabulador horizontal
\\\	Diagonal inversa
\?	Signo de interrogación
\'	Comilla simple
\"	Comilla doble



- Un *especificador de conversión* consiste en el signo de % seguido de un solo carácter. En el ejemplo, el especificador de conversión es %d. Un especificador de conversión le dice a `printf()` la manera en que debe interpretar a la(s) variable(s) que ha(n) de ser impresa(s). El %d le dice a `printf()` que interprete la variable x como un entero decimal con signo.

### Las secuencias de escape de *printf()*

Veamos ahora los componentes del formato a mayor detalle. Las secuencias de escape se usan para controlar la posición de la salida moviendo el cursor de la pantalla, así como para imprimir caracteres que, de no hacerlo así, podrían tener un significado especial para *printf()*. Por ejemplo, para imprimir un solo carácter de diagonal inversa se incluye una doble diagonal inversa (\) en el formato. La primera diagonal inversa le dice a *printf()* que la segunda diagonal inversa debe ser interpretada como un carácter literal y no como el comienzo de una secuencia de escape. En general, la diagonal inversa le dice a *printf()* que interprete el siguiente carácter de una manera especial. A continuación se presentan algunos ejemplos:

Carácter	Descripción
n	El carácter n
\n	Nueva línea
\"	El carácter comillas dobles
S"	El comienzo o final de un texto

La tabla 7.1 lista las secuencias de escape del C más comúnmente usadas. Una lista completa puede encontrarla en el Día 15, “Más sobre apuntadores”.

El listado 7.1 es un programa que muestra algunas de las secuencias de escape frecuentemente usadas.



**Listado 7.1. Uso de las secuencias de escape de *printf()*.**

```

1: /* Demostración de las secuencias de escape usadas frecuentemente */
2:
3: #include <stdio.h>
4:
5: #define QUIT 3
6:
7: int get_menu_choice( void );
8: void print_report( void );
9:
10: main()
11: {
12:     int choice = 0;
13:
14:     while( choice != QUIT )
15:     {
16:         choice = get_menu_choice();
17:
18:         if( choice == 1 )
19:             printf( "\nBeeping the computer\b\b\b" );
20:         else

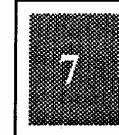
```

---



```
21:     {
22:         if( choice == 2 )
23:             print_report();
24:     }
25: }
26: printf( "You chose to quit!" );
27: }
28:
29: int get_menu_choice( void )
30: {
31:     int selection = 0;
32:
33:     do
34:     {
35:         printf( "\n" );
36:         printf( "\n1 - Beep Computer" );
37:         printf( "\n2 - Display Report");
38:         printf( "\n3 - Quit");
39:         printf( "\n" );
40:         printf( "\nEnter a selection:" );
41:
42:         scanf( "%d", &selection );
43:
44:     }while ( selection < 1 || selection > 3 );
45:
46:     return selection;
47: }
48:
49: void print_report( void )
50: {
51:     printf( "\nSAMPLE REPORT" );
52:     printf( "\n\nSequence\tMeaning" );
53:     printf( "\n=====\\t=====" );
54:     printf( "\n\\a\t\tbell (alert)" );
55:     printf( "\n\\b\t\tbackspace" );
56:
57: }
```

---



```
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence      Meaning
=====        =====
\\a            bell (alert)
\\b            backspace
```

```
...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

 El listado 7.1 parece largo en comparación con los ejemplos anteriores, pero proporciona algunas adiciones que vale la pena mencionar. El archivo de encabezado, STDIO.H, fue incluido en la línea 3 debido a que se usa `printf()` en este listado. En la línea 5 se define una constante llamada QUIT. En el Día 3, "Variables y constantes numéricas", se aprendió que `#define` hace que el uso de la constante QUIT sea equivalente a usar el valor 3. Las líneas 7 y 8 son prototipos de función. Este programa tiene dos funciones, `get_menu_choice()` y `print_report()`. `get_menu_choice()` está definida en las líneas 29 a 47. Esto es similar a la función de menú que se encuentra en el listado 6.5. Las líneas 35 y 39 contienen llamadas a `printf()` que imprimen la secuencia de escape de nueva línea. Las líneas 36, 37, 38 y 40 también usan el carácter de escape de nueva línea e imprimen texto. La línea 35 pudiera haber sido eliminada, cambiando la línea 36 para que fuera de la manera siguiente:

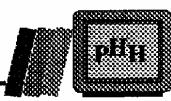
```
printf( "\n\n1 - Beep Computer" );
```

Sin embargo, dejar la línea 35 hace que el programa sea más fácil de leer.

Observando a la función `main()` se ve el comienzo de un ciclo `while` en la línea 14. Los enunciados del `while` se mantendrán haciendo ciclo mientras la selección no sea igual a QUIT. Debido a que QUIT es una constante se le podría haber reemplazado con el número 3. Mas sin embargo, de haber hecho esto el programa no sería tan claro. La línea 16 obtiene la variable choice, que luego es analizada en las líneas 18 a 25 en un enunciado `if`. Si el usuario escoge 1, la línea 19 imprime el carácter de nueva línea, un mensaje y luego da tres pitidos. Si el usuario selecciona 2 en el menú, la línea 23 llama a la función `print_report()`.

`print_report()` está definida en las líneas 49 a 57. Esta función simple muestra lo fácil que es usar a `printf()` y a las secuencias de escape para imprimir información formateada en la pantalla. Ya ha visto el carácter de nueva línea. Las líneas 52 a 56 también usan el carácter de escape tabulador, \t. Con él se alinean verticalmente las columnas del reporte. Las líneas 54 y 55 pueden parecer confusas al principio, pero si se comienza a la izquierda y se empieza a analizar yendo hacia la derecha, toman sentido. La línea 54 imprime una nueva línea (\n), luego una diagonal inversa (\) y luego la letra a, seguida de dos tabuladores (\t\t). La línea termina con un texto descriptivo (bell (alert)). La línea 55 sigue el mismo formato.

Este programa imprime las primeras dos líneas de la tabla 7.1, junto con un título de reporte y encabezados de columnas. En el ejercicio nueve se completará este programa, haciéndolo que imprima el resto de la tabla.



## Los especificadores de conversión de `printf()`

El formato debe contener un especificador de conversión para cada variable a ser impresa. Luego `printf()` despliega cada variable, como lo indica su especificador de conversión correspondiente. Se aprenderá más acerca de este proceso en el Día 15, "Más sobre apuntadores". Por ahora, asegúrese de usar el especificador de conversión que corresponda al tipo de la variable que vaya a imprimirse.

¿Qué significa esto exactamente? Si va a imprimir una variable que es un *entero decimal con signo* (tipos `int` y `long`), use el especificador de conversión `%d`. Para un *entero decimal sin signo* (tipos `unsigned int` y `unsigned long`), use `%u`. Para una *variable de punto flotante* (tipos `float` y `double`), use el especificador `%f`. Los especificadores de conversión que se necesitan más frecuentemente se encuentran listados en la tabla 7.2.

Tabla 7.2. Los especificadores de conversión más comúnmente necesitados.

Especificador	Significado
<code>%c</code>	Un solo carácter
<code>%d</code>	Entero decimal con signo
<code>%f</code>	Número decimal de punto flotante
<code>%s</code>	Cadena de caracteres
<code>%u</code>	Entero decimal sin signo

El texto literal de un especificador de formato es cualquier cosa que no califica como secuencia de escape o especificador de conversión. El texto literal es impreso simplemente como es, incluyendo todos los espacios.

¿Qué hay acerca de la impresión de valores de más de una variable? Un solo enunciado `printf()` puede imprimir una cantidad ilimitada de variables, pero el formato debe contener un especificador de conversión para cada variable. Los especificadores de conversión son apareados con las variables en orden de izquierda a derecha. Si se escribe

```
printf("Tasa = %f, cantidad = %d", tasa, cantidad);
```

la variable `tasa` es apareada con el especificador `%f` y la variable `cantidad` es apareada con el especificador `%d`. Las posiciones de los especificadores de conversión en el formato determinan la posición de la salida. Si hay más variables pasadas a la función `printf()` que especificadores de conversión, las variables que no hacen par no se imprimen. Si hay más especificadores de conversión que variables, los especificadores sin aparear imprimen "basura".





## Entrada/salida básica

No se está limitado a imprimir los valores de variables con `printf()`. Los argumentos pueden ser cualquier expresión del C válida. Por ejemplo, para imprimir la suma de `x` y `y`, se podría escribir:

```
z = x + y;  
printf("%d", z);
```

También se podría escribir:

```
printf("%d", x + y);
```

Cualquier programa que use a `printf()` debe incluir el archivo de encabezado `STDIO.H`. El listado 7.2 muestra el uso de `printf()`. El Día 15, "Más sobre apuntadores", da mayores detalles sobre `printf()`.



### Listado 7.2. Uso de `printf()` para desplegar valores numéricos.

```
1:  /* Demostración del uso de printf() para desplegar valores numéricos. */  
2:  
3:  #include <stdio.h>  
4:  
5:  int a = 2, b = 10, c = 50;  
6:  float f = 1.05, g = 25.5, h = -0.1;  
7:  
8:  main()  
9:  {  
10:    printf("\nDecimal values without tabs: %d %d %d", a, b, c);  
11:    printf("\nDecimal values with tabs: \t%d \t%d \t%d", a, b, c);  
12:    printf("\nThree floats on 1 line: \t%f\t%f\t%f", f, g, h);  
13:    printf("\nThree floats on 3 lines: \n\t%f\n\t%f\n\t%f", f,  
14:           g,h);  
15:    printf("\nThe rate is %f%%", f);  
16:    printf("\nThe result of %f/%f = %f", g, f, g / f);  
17:  
18: }
```



```
Decimal values without tabs: 2 10 50  
Decimal values with tabs:      2      10      50  
Three floats on 1 line: 1.050000 25.500000 -0.100000  
Three floats on 3 lines:  
1.050000  
25.500000  
-0.100000  
The rate is 1.050000%  
The result of 25.500000/1.050000 = 24.285715
```



El listado 7.2 imprime seis líneas de información. Las líneas 10 y 11 imprimen cada una tres valores decimales, `a`, `b` y `c`. La línea 10 los imprime sin tabuladores y la línea 11 los imprime con tabuladores. Las líneas 13 y 14 imprimen cada una tres variables

`float, f, g y h.` La línea 14 las imprime en una línea y la línea 13 las imprime en tres líneas. La línea 16 imprime una variable `float, f,` seguida por un signo de porcentaje. Debido a que el signo de porcentaje normalmente es un mensaje para imprimir una variable, se debe poner dos de ellos juntos para imprimir un solo signo de porcentaje. Esto es exactamente similar al carácter de escape de diagonal inversa. La línea 17 muestra un concepto final. Cuando se imprimen valores en los especificadores de conversión no se tienen que usar variables. También se pueden usar expresiones, como `g / f,` o hasta constantes.

## DEBE

## NO DEBE

**NO DEBE** Tratar de poner varias líneas de texto en un solo enunciado `printf()`. La mayoría de las veces es más claro imprimir varias líneas con varios enunciados de impresión que hacerlo con uno solo y con varios caracteres de escape de nueva linea (`\n`).

**NO DEBE** Olvidar usar el carácter de escape de nueva línea cuando imprima varias líneas de información en enunciados `printf()` separados.

### La función `printf()`

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

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

El formato es requerido. Sin embargo, los argumentos son opcionales. Para cada argumento debe haber un especificador de conversión. La tabla 7.2 lista los especificadores de conversión necesarios más comunes.

El formato también puede contener secuencias de escape. La tabla 7.1 lista las secuencias de escape más frecuentemente usadas.

Los siguientes son ejemplos de llamadas a `printf()` y su salida:

#### Ejemplo 1

```
#include <stdio.h>
main()
{
    printf( "Este es un ejemplo de algo impreso!");
}
```

## Entrada/salida básica

### Despliega

¡Esté es un ejemplo de algo impreso!

### Ejemplo 2

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

### Despliega

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

## Desplegado de mensajes con *puts()*

La función *puts()* también puede ser usada para desplegar mensajes de texto en la pantalla, pero no puede desplegar variables numéricas. *puts()* toma un sola cadena como su argumento y la despliega, añadiendo automáticamente una nueva línea al final. Por ejemplo, el enunciado

```
puts("Hola.");
ejecuta la misma acción que
printf("Hola.\n");
```

Se pueden incluir secuencias de escape (incluyendo `\n`) en una cadena que se le pasa a *puts()*. Tienen el mismo efecto que cuando se usan con *printf()* (véase la tabla 7.1).

Cualquier programa que use *puts()* debe incluir al archivo de encabezado *STDIO.H*. Tome en cuenta de que *STDIO.H* debe ser incluido solamente una vez en cualquier programa

### DEBE

### NO DEBE

**DEBE** Usar la función *puts()* en vez de la función *printf()* cada vez que se quiera imprimir texto, pero que no se necesite imprimir variables.

**NO DEBE** Tratar de usar especificadores de conversión con los enunciados *puts()*.

### Sintaxis

#### La función *puts()*

```
#include <stdio.h>
puts( cadena );
```

*puts()* es una función que copia una cadena al dispositivo de salida estándar, que por lo general es la pantalla. Cuando use *puts()* incluya el archivo de encabezado de entrada/salida estándar (*STDIO.H*). *puts()* también añade un carácter de nueva línea al final de

la cadena que es impresa. La cadena puede contener secuencias de escape. La tabla 7.1, que se mostró anteriormente, lista las secuencias de escape más frecuentemente usadas.

Los siguientes son ejemplos de llamadas a `puts()` y su salida:

**Ejemplo 1**

```
puts( "¡Esto es impreso con la función puts()!" );
```

**Despliega**

```
¡Esto es impreso con la función puts()!
```

**Ejemplo 2**

```
puts( "Esto se imprime en la primera línea. \nEsto se imprime en la  
segunda línea." );  
puts( "Esto se imprime en la tercera línea." );  
puts( "Si se usara printf(), las cuatro líneas estarían en dos  
líneas!" );
```

**Despliega**

```
Esto se imprime en la primera línea.  
Esto se imprime en la segunda línea.  
Esto se imprime en la tercera línea.  
Si se usara printf(), las cuatro líneas estarían en dos líneas!
```

## Entrada de datos numéricos con `scanf()`

Así como muchos programas necesitan la salida de datos a pantalla, también necesitan la entrada de datos del teclado. La manera más flexible para que el programa pueda leer datos numéricos del teclado es el uso de la función de biblioteca `scanf()`.

La función `scanf()` lee datos del teclado de acuerdo con un formato especificado, y asigna los datos de entrada a una o más variables del programa. De manera similar a `printf()`, `scanf()` usa un formato para describir el formato de la entrada. El formato utiliza los mismos especificadores de conversión que la función `printf()`. Por ejemplo, el enunciado

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

lee un entero decimal del teclado y lo asigna a la variable entera `x`. De manera similar el enunciado

```
scanf( "%f", &tasa );
```

lee un valor de punto flotante del teclado y lo asigna a la variable `tasa`.



## Entrada/salida básica

¿Qué hace el & antes del nombre de la variable? El símbolo & es el operador de dirección de C, que es explicado a detalle en el Día 9, "Apuntadores". Por ahora, todo lo que necesita recordar es que `scanf()` requiere el símbolo & antes de cada nombre de variable numérica en su lista de argumentos (a menos que la variable sea un *apuntador*, lo que también se explica en el Día 9).

Un solo `scanf()` puede aceptar la entrada de más de un valor si se incluyen varios especificadores de conversión en el formato y varias variables (nuevamente cada una de ellas precedida por un & en la lista de argumentos). El enunciado

```
scanf("%d %f", &x, &tasa);
```

da entrada a un valor entero y a un valor de punto flotante, y los asigna a las variables `x` y `tasa`, respectivamente. Cuando se da entrada a diversas variables, `scanf()` usa al espacio en blanco para separar la entrada en campos. El espacio en blanco pueden ser blancos, tabuladores o nuevas líneas. Cada especificador de conversión en el formato del `scanf()` es apareado con un campo de entrada. El final de cada campo de entrada es identificado por el espacio en blanco.

Esto le da bastante flexibilidad. En respuesta al `scanf()` anterior se podría teclear

```
10 12.45
```

También se podría teclear

```
10          12.45
```

```
o
```

```
10  
12.45
```

Mientras haya algún espacio en blanco entre los valores, `scanf()` puede asignar cada valor a su variable.

De manera similar a las otras funciones tratadas en este capítulo, los programas que usan a `scanf()` deben incluir al archivo de encabezado `STDIO.H`. Aunque el listado 7.3 le da un ejemplo sobre el uso de `scanf()`, una descripción más completa se presenta en el Día 15, "Más sobre apuntadores".



### Listado 7.3. Uso de `scanf()` para obtener valores numéricos.

```
1:  /* Demostración del uso de scanf() */  
2:  
3:  #include <stdio.h>  
4:  
5:  #define QUIT 4  
6:  
7:  int get_menu_choice( void );  
8:
```

```
9: main()
10:{ 
11:     int choice = 0;
12:     int int_var = 0;
13:     float float_var = 0.0;
14:     unsigned unsigned_var = 0;
15:
16:     while( choice != QUIT )
17:     {
18:         choice = get_menu_choice();
19:
20:         if( choice == 1 )
21:         {
22:             puts( "\nEnter a signed decimal integer (i.e. -123)" );
23:             scanf( "%d", &int_var );
24:         }
25:         if ( choice == 2 )
26:         {
27:             puts( "\nEnter a decimal floating-point number
28:                   (i.e. 1.23)" );
29:             scanf( "%f", &float_var );
30:         }
31:         if ( choice == 3 )
32:         {
33:             puts( "\nEnter an unsigned decimal integer \
34:                   (i.e. 123)" );
35:             scanf( "%u", &unsigned_var );
36:         }
37:     printf( "\nYour values are: int: %d  float: %f  unsigned: %u ",
38:             int_var, float_var, unsigned_var );
39:
40:     int get_menu_choice( void )
41:     {
42:         int selection = 0;
43:
44:         do
45:         {
46:             puts( "1 - Get a signed decimal integer" );
47:             puts( "2 - Get a decimal floating-point number" );
48:             puts( "3 - Get an unsigned decimal integer" );
49:             puts( "4 - Quit" );
50:             puts( "\nEnter a selection:" );
51:
52:             scanf( "%d", &selection );
53:         }while ( selection < 1 || selection > 4 );
54:
55:
56:         return selection;
57:     }
```



## Entrada/salida básica

```
7
Salida
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123  float: 1231.123047 unsigned: 321
```



El listado 7.3 usa los mismos conceptos de menú que se usaron en el listado 7.1. Las diferencias en `get_menu_choice()` (líneas 40 a 57) son menores, pero deben ser mencionadas. En primer lugar se usa `puts()` en vez de `printf()`. Debido a que no se imprimen variables no hay necesidad de usar `printf()`. Debido a que se usa `puts()` se han quitado los caracteres de escape de nueva línea de las líneas 47 a 49. La línea 54 también fue cambiada para permitir valores del 1 al 4, debido a que ahora hay cuatro opciones de menú. Observe que la línea 52 no ha cambiado; sin embargo, ahora debe entenderla mejor. `scanf()` obtiene un valor decimal y lo pone en la variable de selección. En la línea 56 la función regresa `selection` al programa que la llama.

Los listados 7.1 y 7.3 usan la misma estructura de `main()`. Un enunciado `if` evalúa a `choice`, que es el valor de retorno de `get_menu_choice()`. Basándose en el valor de `choice`, el programa imprime un mensaje, pide que se teclee un número y lee el valor con `scanf()`. Observe la diferencia entre las líneas 22, 28 y 32. Cada una está puesta para obtener un tipo diferente de variable. Las líneas 12 a 14 declaran variables para los tipos apropiados.

Cuando el usuario selecciona `quit`, el programa imprime el último número tecleado para cada uno de los tres tipos. Si el usuario no teclea un valor se imprime 0, debido a que las líneas 12 y 13 inicializan a los tres tipos. Una nota final sobre las líneas 20 a 34: Los enunciados `if` que se usan aquí no están bien estructurados. Si usted piensa que una estructura `if...else` hubiera sido mejor, está en lo correcto. En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se presenta un nuevo enunciado de control, `switch`. Este enunciado hubiera sido la mejor opción.

## DEBE

## NO DEBE

**NO DEBE** Olvidar incluir al operador *de dirección de* (&) cuando use variables en `scanf()`.

**DEBE** Usar `printf()` o `puts()` junto con `scanf()`. Use las funciones de impresión para desplegar un mensaje donde pida los datos que quiere obtener con `scanf()`.

### La función `scanf()`

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

`scanf()` es una función que usa un *especificador de conversión* en un *formato* dado para poner valores en las variables de argumento. Los argumentos deben ser las direcciones de las variables, en vez de las variables actuales. Para las variables numéricas se puede pasar la dirección, añadiendo el operador *de dirección de* (&) al inicio del nombre de variable. Cuando se use `scanf()` se debe incluir el archivo de encabezado `STDIO.H`.

`scanf()` lee campos de entrada del flujo de entrada estándar que, por lo general, es el teclado. Pone cada uno de estos campos leídos en un argumento. Cuando pone la información la convierte al formato del especificador de conversión correspondiente que se encuentra en el formato. Para cada argumento debe haber un especificador de conversión. La tabla 7.2, mostrada anteriormente, lista los especificadores de conversión necesarios más comunes.

Los siguientes son ejemplos de llamadas a `scanf()`:

#### Ejemplo 1

```
int x, y, z;
scanf( "%d %d %d", &x, &y, &z);
```

#### Ejemplo 2

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



```

    {
        float y;
        int x;

        puts( "Teclee un punto flotante, luego un entero" );
        scanf( "%f %d", &y, &x );
        printf( "\nSe tecleó %f y %d ", y, x );
    }

```

## Resumen

Al terminar este capítulo ya está listo para escribir sus propios programas en C. Combinando las funciones `printf()`, `puts()` y `scanf()` y el control de programación que se aprendió en los capítulos anteriores, se tienen las herramientas necesarias para escribir programas simples.

El desplegado en pantalla se ejecuta con las funciones `printf()` y `puts()`. La función `puts()` puede desplegar solamente mensajes de texto, mientras que `printf()` puede desplegar mensajes de texto y variables. Ambas funciones usan secuencias de escape para los caracteres especiales y control de la impresión.

La función `scanf()` lee uno o más valores numéricos del teclado e interpreta a cada uno de acuerdo con un especificador de conversión. Cada valor es asignado a una variable de programa.

## Preguntas y respuestas

1. ¿Por qué debo usar `puts()` si `printf()` hace todo lo que hace `puts()` y más?

Debido a que `printf()` hace más, tiene una sobrecarga adicional. Cuando se esté tratando de escribir un programa pequeño eficiente o cuando los programas se hacen grandes y los recursos son valiosos, se querrá tomar ventaja de la pequeña sobrecarga de `puts()`. Por lo general, use el recurso disponible más sencillo.

2. ¿Por qué necesito incluir `STDIO.H` cuando uso `printf()`, `puts()` o `scanf()`?

`STDIO.H` contiene los prototipos para las funciones de entrada/salida estándares. `printf()`, `puts()` y `scanf()` son tres de estas funciones estándares. Trate de ejecutar un programa sin el archivo de encabezado `STDIO.H` y vea los errores y avisos que obtiene.

3. ¿Qué pasa si no pongo el operador (`&`) en una variable de `scanf()`?

Este es un error fácil de cometer. Pueden producirse resultados impredecibles si olvida la dirección del operador. Cuando lea acerca de los apuntadores en los Días 9 y 13, "Apuntadores" y "Más sobre el control de programa", comprenderá mejor esto. Por ahora, simplemente sepa que si omite la dirección del operador, scanf() no pone la información tecleada en la variable sino en algún otro lugar de la memoria. Esto puede no producir efecto visible alguno, o bien, por lo contrario, dar lugar a cualquier efecto, incluyendo el de que su computadora se trabe y tenga que rearrancarla.

## Taller

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

### Cuestionario

1. ¿Cuál es la diferencia entre puts() y printf()?
2. ¿Qué archivo de encabezado debe ser incluido cuando se usa a printf()?
3. ¿Qué hacen las siguientes secuencias de escape?
  - a. \\
  - b. \\b
  - c. \\n
  - d. \\t
  - e. \\a
4. ¿Qué especificadores de conversión deben ser usados para imprimir lo siguiente?
  - a. una cadena.
  - b. un entero decimal con signo.
  - c. un número decimal de punto flotante.
5. ¿Cuál es la diferencia entre usar cada uno de los siguientes en el texto literal de puts()?
  - a. b
  - b. \\b
  - c. \\
  - d. \\W



## Ejercicios



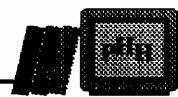
**Nota:** Comenzando con este capítulo, algunos de estos ejercicios le piden que escriba programas completos que ejecuten una tarea particular. Siempre hay más de una manera de hacer las cosas en C, por lo que las respuestas que se proporcionan al final del libro no deben ser interpretadas como las únicas correctas. Si usted puede escribir su propio código que ejecute lo que se desea, ¡está muy bien! Si tiene problemas, vea el ejemplo de respuesta como ayuda. Estas respuestas son presentadas con un mínimo de comentarios. ¡Es buena práctica para usted imaginar la manera en que trabajan!

1. Escriba un enunciado con `printf()` y otro con `puts()` que inicien una nueva línea.
2. Escriba un enunciado `scanf()` que pueda ser usado para obtener un carácter, un entero decimal sin signo y otro carácter solo.
3. Escriba los enunciados para obtener un valor entero e imprimirla.
4. Modifique el ejercicio 3 para que acepte solamente valores pares (2, 4, 6, etcétera).
5. Modifique el ejercicio 4 para que regrese valores hasta que se teclee el número 99 o hasta que se hayan tecleado seis valores pares. Guarde los números en un arreglo. (Consejo: ¡Se necesita un ciclo!)
6. Cambie el ejercicio 5 en un programa ejecutable. Añada una función que imprima los valores del arreglo, separados con tabuladores, en una sola línea. (Sólo imprima los valores que fueron guardados en el arreglo.)
7. **BUSQUEDA DE ERRORES:** Encuentre el error en el siguiente fragmento de código:

```
printf( "Jack said, \"Peter Piper picked a peck of pickled  
peppers.\");
```

8. **BUSQUEDA DE ERRORES:** Encuentre los errores en el siguiente programa:

```
int get_1_or_2( void )  
{  
    int answer = 0;  
  
    while( answer < 1 || answer > 2 )  
    {
```



```
    printf(Enter 1 for Yes, 2 for No);
    scanf( "%f", answer );
}
return answer;
}
```

9. Usando el listado 7.1 complete la función print\_report() para que imprima el resto de la tabla 7.1.
10. Escriba un programa que reciba del teclado dos valores de punto flotante y luego despliegue su producto.
11. Escriba un programa que reciba del teclado 10 valores enteros y luego despliegue su suma.
12. Escriba un programa que reciba del teclado enteros, guardándolos en un arreglo. La entrada debe parar cuando se teclee un cero o cuando se llegue al fin del arreglo. Luego, encuentre y despliegue los valores mayor y menor del arreglo. (Nota: Este es un problema difícil debido a que todavía no han sido tratados los arreglos completamente. Si tiene dificultades, trate de resolverlos nuevamente después de haber leído el Día 8, "Arreglos numéricos".)



## SEMANA

1

### Revisión de la semana 1

Después de que haya acabado su primera semana de aprendizaje sobre la manera de programar en C, deberá sentirse a gusto tecleando programas y usando el compilador y el editor. El siguiente programa reúne muchos de los temas de la semana anterior.



**Nota:** Los números a la izquierda de los números de renglón indican el capítulo donde se trata el concepto presentado en esa línea. Si no entiende bien la linea, vea el capítulo a que se hace referencia para mayor información.

### REVISIÓN

- 1
- 2
- 3
- 4
- 5
- 6
- 7



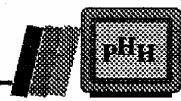
## Revisión de la semana 1

---

### Listado R1.1. Listado de revisión de la semana 1.

---

```
CH02    1: /* Nombre del programa: week1.c */  
        2: /* programa para teclear las edades e ingresos */  
        3: /* de hasta 100 gentes. El programa imprime un */  
        4: /* reporte basado en las cantidades tecleadas. */  
        5: /*—————*/  
        6: /*—————*/  
        7: /* Archivos de inclusión */  
        8: /*—————*/  
CH02    9: #include <stdio.h>  
10:  
CH02   11: /*—————*/  
12: /* Constantes definidas */  
13: /*—————*/  
14:  
CH03   15: #define MAX 100  
16: #define YES 1  
17: #define NO 0  
18:  
CH02   19: /*—————*/  
20: /* Variables */  
21: /*—————*/  
22:  
CH03   23: long income[MAX]; /* Para guardar los ingresos */  
24: int month[MAX], day[MAX], year[MAX]; /* Para guardar fechas de  
nacimiento */  
25: int x, y, ctr; /* Para contadores */  
26: int cont; /* Para control de programa */  
27: long month_total, grand_total; /* Para totales */  
28:  
CH02   29: /*—————*/  
30: /* Prototipos de función */  
31: /*—————*/  
32:  
CH05   33: void main(void)  
34: int display_instructions(void);  
35: void get_data(void);  
36: void display_report();  
37: int continue_function(void);  
38:  
39: /*—————*/  
40: /* Inicio del programa */
```



```
41: /*-----*/
42:
CH02 43: void main()
44: {
CH05 45:     cont = display_instructions();
46:
CH04 47:     if ( cont == YES )
48:     {
CH05 49:         get_data();
CH05 50:         display_report();
51:     }
CH04 52:     else
CH07 53:         printf( "\nProgram Aborted by User!\n\n");
54: }
CH02 55: /*-----*
56: * Función: display_instructions()
57: * Objetivo: esta función despliega información sobre la manera *
58: *           de usar este programa y pide al usuario teclear 0 *
59: *           para terminar o 1 para continuar. *
60: * Regresa: NO - si el usuario teclea 0 *
61: * YES - si el usuario teclea cualquier número diferente de 0 *
62: *-----*/
63:
CH05 64: int display_instructions( void )
65: {
CH07 66:     printf("\n\n");
67:     printf("\nThis program enables you to enter up to 99 \
       people's ");
68:     printf("\nincome and birthdays. It then prints the \
       incomes by");
69:     printf("\nmonth along with the overall income and \
       overall average.");
70:     printf("\n");
71:
CH05 72:     cont = continue_function();
73:
CH05 74:     return( cont );
75: }
CH02 76: /*-----*
77: * Función: get_data() *
```



## Revisión de la semana 1

### Listado R1.1. continuación

```
78: * Objetivo: esta función obtiene datos del usuario. *
79: *           continúa pidiendo datos hasta que se hayan tecleado   *
80: *           100 gentes o hasta que el usuario teclee 0 en el mes   *
81: * Regresa: nada   *
82: * Nota: Se permite que se teclee 0/0/0 para la fecha de nacimiento *
83: *        en caso de que el usuario no esté seguro. También permite   *
84: *        que todos los meses sean de 31 días.                      *
85: *-----*/  
86:  
CH05 87: void get_data(void)  
88: {  
CH06 89:     for ( cont = YES, ctr = 0; ctr < MAX && cont == YES; ctr++ )  
90:     {  
CH07 91:         printf("\nEnter information for Person %d.", ctr+1 );  
92:         printf("\n\tEnter Birthday:");  
93:  
CH06 94:         do  
95:         {  
CH07 96:             printf("\n\tMonth (0 - 12): ");  
CH07 97:             scanf("%d", &month[ctr]);  
CH06 98:         }while (month[ctr] < 0 || month[ctr] > 12 );  
99:  
CH06 100:        do  
101:        {  
CH07 102:            printf("\n\tDay (0 - 31): ");  
CH07 103:            scanf("%d", &day[ctr]);  
CH06 104:        }while ( day[ctr] < 0 || day[ctr] > 31 );  
105:  
CH06 106:        do  
107:        {  
CH07 108:            printf("\n\tYear (0 - 1994): ");  
CH07 109:            scanf("%d", &year[ctr]);  
CH06 110:        }while ( year[ctr] < 0 || year[ctr] > 1994 );  
111:  
CH07 112:        printf("\nEnter Yearly Income (whole dollars): ");  
CH07 113:        scanf("%ld", &income[ctr]);  
114:  
CH05 115:        cont = continue_function();  
116:    }  
CH07 117: /* ctr es igual a la cantidad de gente que se ha tecleado. */
```

```
118: }
CH02 119: /*—————*
120: * Función: display_report() *
121: * Objetivo: esta función despliega un reporte en la pantalla *
122: * Regresa: nada *
123: * Notas: Se podría haber impreso más información. *
124: *—————*/
125:
CH05 126: void display_report()
127: {
CH04 128:     grand_total = 0;
CH07 129:     printf("\n\n\n");           /* se salta unas cuantas líneas */
130:     printf("\n          SALARY SUMMARY");
131:     printf("\n          ======");
132:
CH06 133:     for( x = 0; x <= 12; x++ ) /* para cada mes, incluyendo 0 */
134:     {
135:         month_total = 0;
CH04 136:         for( y = 0; y < ctr; y++ )
CH06 137:         {
138:             if( month[y] == x )
CH04 139:                 month_total += income[y];
CH04 140:         }
141:         printf("\nTotal for month %d is %ld", x, month_total);
CH07 142:         grand_total += month_total;
CH04 143:     }
144:     printf("\n\nReport totals:");
CH07 145:     printf("\nTotal Income is %ld", grand_total);
146:     printf("\nAverage Income is %ld", grand_total/ctr );
147:
148:     printf("\n\n* * * End of Report * * *");
149: }
CH02 150: /*—————*
151: * Función: continue_function() *
152: * Objetivo: esta función le pregunta al usuario si quiere continuar. *
153: * Regresa: YES - si el usuario desea continuar *
154: *           NO - si el usuario desea terminar *
155: *—————*/
156:
CH05 157: int continue_function( void )
158: {
```



## Revisión de la semana 1

### Listado R1.1. continuación

```
CH07    159:     printf("\n\nDo you wish to continue? (0=NO/1=YES): ");
160:     scanf( "%d", &x );
161:
CH06    162:     while( x < 0 || x > 1 )
163:     {
CH07    164:         printf("\n%d is invalid!", x);
165:         printf("\nPlease enter 0 to Quit or 1 to Continue: ");
166:         scanf("%d", &x);
167:     }
CH04    168:     if(x == 0)
CH05    169:         return(NO);
CH04    170:     else
CH05    171:         return(YES);
172: }
```

Después de haber completado los cuestionarios y los ejercicios del Día 1, “Comienzo”, y del 2, “Los componentes de un programa C”, deberá ser capaz de teclear y compilar este programa. Este programa contiene más comentarios que otros listados en este libro. Estos comentarios son típicos de un programa C real. En particular, observe los comentarios que se encuentran al principio del programa y antes de cada función principal. Los comentarios de las líneas 1 a la 5 contienen una descripción del programa completo, incluyendo el nombre del programa. Algunos programadores también incluyen información como el nombre del autor del programa, el compilador usado, su número de versión, las bibliotecas enlazadas en el programa y la fecha en que el programa fue creado. Los comentarios que se encuentran antes de cada función describen el objetivo de la función, los valores de retorno posibles, las convenciones de llamado de la función y cualquier otra cosa que se relaciona específicamente con esa función.

Los comentarios de las líneas 1 a 5 especifican que se puede dar información en este programa para un máximo de 100 gentes. Antes de que pueda teclear los datos el programa llama a la función `display_instructions()` (línea 45). Esta función despliega instrucciones sobre el uso del programa y le pregunta si quiere continuar o terminar. En las líneas 66 a 70 se puede ver que esta función usa la función `printf()`, que se vio en el Día 7, “Entrada/salida básica”, para desplegar las instrucciones.

La función `continue_function()`, que se encuentra en las líneas 157 a 172, usa algunas de las características tratadas al final de la semana. La función pregunta si se quiere continuar (línea 159). Usando el enunciado de control `while` del Día 6, “Control básico del programa”, la función verifica que la respuesta tecleada sea 0 o 1. Mientras



---

la respuesta no sea alguno de estos dos valores la función se mantiene pidiendo una respuesta. Una vez que el programa recibe la respuesta adecuada, un enunciado `if...else` (Día 4, “Enunciados, expresiones y operadores”) regresa una constante de YES o NO.

La parte modular de este programa se encuentra en dos funciones: `get_data()` y `display_report()`. La función `get_data()` le pide que teclee datos, poniendo la información en los arreglos declarados cerca del principio del programa. Usando un enunciado `for`, en la línea 89, se pide que se tecleen datos hasta que `cont` no sea igual a la constante definida YES (regresada de la función `continue_function()`) o el contador, `ctr`, sea mayor o igual al número máximo de elementos del arreglo, `MAX`. Este programa revisa la información tecleada, para asegurarse de que es la adecuada. Por ejemplo, las líneas 94 a 98 le piden que teclee un mes. Los únicos valores que acepta el programa son del 0 al 12. Si se teclea un número mayor de 12 el programa vuelve a pedir el mes. La línea 115 llama a la función `continue_function()` para revisar si se quiere continuar añadiendo datos.

Cuando se responde a la función de continuar con un 0 o cuando el número máximo de juegos de información es tecleado (de acuerdo al ajuste de `MAX`), el programa regresa a la línea 50 de `main()`, donde llama a `display_report()`. La función `display_report()`, que se encuentra en las líneas 119 a 149, imprime un reporte en la pantalla. Este reporte usa un ciclo `for` anidado, para dar el total de ingresos para cada mes y un gran total para todos los meses. Este reporte puede parecer complicado. Si es así, revise el Día 6, “Control básico del programa”, sobre la explicación de los enunciados anidados. Muchos de los reportes que se crean cuando uno es programador son más complicados que éste.

Este programa usa lo que se ha aprendido en la primera semana de aprendizaje del C. Ha sido una gran cantidad de material en sólo una semana, ¡pero lo ha logrado! Si usa todo lo que ha aprendido esta semana podrá escribir sus propios programas en C. Sin embargo, todavía hay límites a lo que puede hacer.

# 2

Ya ha terminado su primera semana de aprendizaje sobre cómo programar en C. Por ahora debe sentirse a gusto tecleando programas y usando el editor y el compilador.

## Adónde vamos...

La segunda semana trata una gran cantidad de material. Aprenderá muchas de las características que forman la parte metular del lenguaje C. Aprenderá cómo usar arreglos numéricos y de carácter, expandir variables de tipo carácter en arreglos y cadenas, y agrupar tipos diferentes de variables usando estructuras.

La segunda semana complementa algunos de los temas aprendidos en la primera semana, presenta enunciados adicionales para el control de programa, proporciona explicaciones detalladas de funciones y presenta funciones alternativas.

Los Días 9, "Apuntadores" y 12, "Alcance de las variables", se enfocan en conceptos del C que son extremadamente importantes. Pasará tiempo extra trabajando con apuntadores y sus funciones básicas.

## SEMANA

# UN VISTAZO

8

9

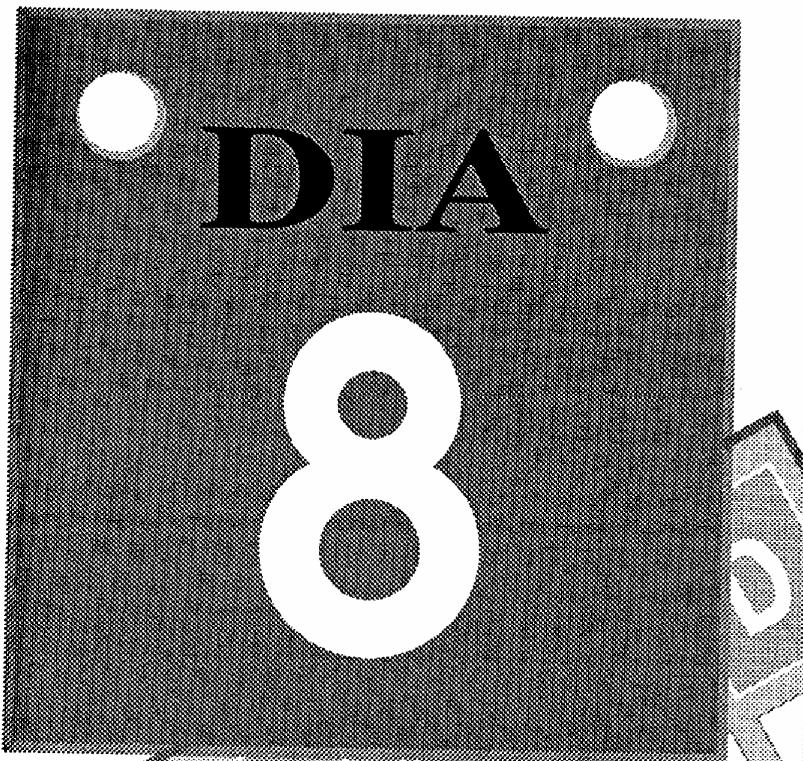
10

11

12

13

14



Los arreglos son un tipo de almacenamiento de datos que se usan frecuentemente en los programas en C. Ya se tuvo una breve introducción en el Día 6, "Control básico del programa". Hoy aprenderá

- Lo que es un arreglo.
- La definición de arreglos numéricos de una sola y de varias dimensiones.
- Cómo declarar e inicializar arreglos.

## ¿Qué es un arreglo?

Un *arreglo* es una colección de posiciones de almacenamiento de datos, donde cada una tiene el mismo tipo de dato y el mismo nombre. Cada posición de almacenamiento en un arreglo es llamada un *elemento del arreglo*. ¿Por qué necesitamos arreglos en los programas? Esta pregunta puede ser respondida con un ejemplo. Si se está llevando cuenta de los gastos de un negocio en 1994 y se están archivando los recibos por mes, se puede tener una carpeta separada para los recibos de cada mes, pero sería más conveniente tener una sola carpeta con 12 compartimientos.

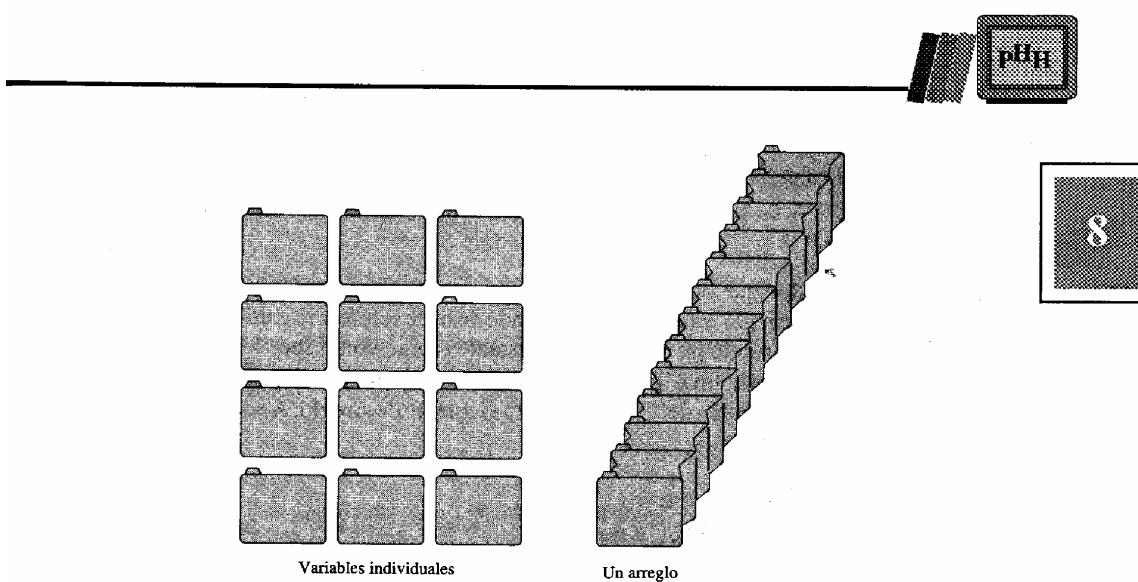
Extienda este ejemplo a la programación de computadora. Imagínese que está diseñando un programa para llevar la cuenta de los totales de gastos de un negocio. El programa podría declarar 12 variables separadas, cada una para el total de gastos del mes. Este enfoque es similar a tener 12 carpetas separadas para los recibos. Sin embargo, la buena práctica de programación utilizaría un arreglo con 12 elementos, guardando el total de cada mes en el elemento de arreglo correspondiente. Este enfoque es comparable al archivado de los recibos en una sola carpeta con 12 compartimientos. La figura 8.1 ilustra la diferencia entre el uso de variables individuales y de un arreglo.

## Arreglos de una sola dimensión

Un arreglo de una *sola dimensión* es un arreglo que tiene solamente un subíndice. Un *subíndice* es un número encerrado entre corchetes a continuación del nombre del arreglo. Este número puede identificar la cantidad de elementos individuales en el arreglo. Un ejemplo hará esto más claro. Para el programa de gastos de un negocio, se podría usar esta línea del programa

```
float gastos[12];
```

para declarar un arreglo de tipo `float`. El arreglo es llamado `gastos` y contiene 12 elementos. Cada uno de los 12 elementos es el equivalente exacto de una sola variable `float`. En los arreglos se pueden usar todos los tipos de datos del C. Los elementos de arreglos del C son numerados siempre comenzando en 0, por lo que los 12 elementos de gastos son numerados del 0 al 11. En el ejemplo anterior, el total de gastos de enero sería guardado en `gastos[0]`, los de febrero en `gastos[1]` y así sucesivamente.



**Figura 8.1.** Las variables son como carpetas individuales, y un arreglo es como una sola carpeta con muchos compartimientos.

Cuando se declara un arreglo, el compilador reserva un bloque de memoria lo suficientemente grande como para guardar el arreglo completo. Los elementos individuales del arreglo son guardados en posiciones consecutivas de memoria, como se ilustra en la figura 8.2.

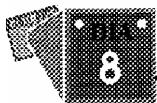


**Figura 8.2.** Los elementos del arreglo son guardados en posiciones secuenciales en memoria.

La ubicación de las declaraciones de arreglos en el código fuente es importante. De manera similar a las variables que no son de arreglo, la posición de la declaración afecta la manera en que el programa puede usar el arreglo. El efecto de la ubicación de la declaración se trata a mayor detalle en el Día 12, “Alcance de las variables”. Por ahora ponga las declaraciones de arreglo junto con las otras declaraciones de variable, inmediatamente antes del inicio de `main()`.

Un elemento de arreglo puede ser usado en cualquier parte del programa en donde pueda ser usada una variable del mismo tipo que no sea de arreglo. Los elementos individuales de los arreglos son accesados usando el nombre del arreglo, seguido del subíndice del elemento encerrado entre corchetes. Por ejemplo, el enunciado

```
gastos[1] = 89.95;
```



## Arreglos numéricos

guarda el valor 89.95 en el segundo elemento del arreglo. (Recuerde que el primer elemento del arreglo es `gastos[0]` y no `gastos[1]`.) De manera similar, el enunciado  
`gastos[10] = gastos[11];`

asigna el valor que se encuentra guardado en el elemento de arreglo `gastos[11]` al elemento de arreglo `gastos[10]`. Cuando se hace referencia a un elemento de arreglo, el subíndice del arreglo puede ser una constante literal, como sucede en estos ejemplos. Sin embargo, los programas pueden frecuentemente usar un subíndice que es una variable entera, o una expresión del C o incluso otro elemento de arreglo. A continuación se presentan algunos ejemplos:

```
float gastos[100];
int a[10];
/* aquí van otros enunciados */
gastos[i] = 100;      /* i es una variable entera      */
gastos[2+3] = 100;    /* es equivalente a gastos[5]   */
gastos[a[2]] = 100;  /* a[] es un arreglo de enteros */
```

El último ejemplo tal vez necesite una explicación. Digamos, por ejemplo, que se tiene un arreglo de enteros llamado `a[]` y que el valor 8 se encuentra guardado en el elemento `a[2]`. Entonces, al escribir

```
gastos[a[2]]
```

$a[2] = 8$

se tiene el mismo efecto que al escribir

```
gastos[8];
```

Cuando use arreglos no olvide el esquema de numeración de los elementos: en un arreglo de  $n$  elementos los subíndices permitidos van de 0 a  $n - 1$ . Si se usa un valor de subíndice  $n$  se pueden tener errores de programa. El compilador de C no se da cuenta si el programa usa un subíndice de arreglo que está fuera de los límites. El programa compila y enlaza, pero, por lo general, los subíndices fuera de rango producen resultados erróneos.

Algunas veces tal vez quiera tratar un arreglo de  $n$  elementos como si sus elementos estuvieran numerados del 1 al  $n$ . Por ejemplo, en el ejemplo anterior sería un método más natural guardar el total de gastos de enero en `gastos[1]`, los de febrero en `gastos[2]` y así sucesivamente. La manera más simple de hacer esto es declarar el arreglo con un elemento más de los necesarios e ignorar al elemento 0. En este caso se declararía al arreglo

```
float gastos[13];
```

También se podría guardar algún dato relacionado en el elemento 0 (tal vez el total de gastos anuales).

El programa EXPENSES.C, que se encuentra en el listado 8.1, muestra el uso de un arreglo. Este es un programa simple que no tiene un uso práctico, sino que es solamente para objetos de demostración.



### Listado 8.1. EXPENSES.C.

8

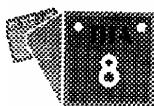
```

1:  /* EXPENSES.C - Demostración del uso de un arreglo */
2:
3:  #include <stdio.h>
4:
5:  /* Declara un arreglo donde guarda gastos y una variable de contador */
6:
7:  float expenses[13];
8:  int count;
9:
10: main()
11: {
12:     /* Recibe los datos del teclado y los guarda en el arreglo */
13:
14:     for (count = 1; count < 13; count++)
15:     {
16:         printf("Enter expenses for month %d: ", count);
17:         scanf("%f", &expenses[count]);
18:     }
19:
20:     /* Imprime el contenido del arreglo */
21:
22:     for (count = 1; count < 13; count++)
23:     {
24:         printf("\nMonth %d = $%.2f", count, expenses[count]);
25:     }
26: }
```



Enter expenses for month 1: 100  
 Enter expenses for month 2: 200.12  
 Enter expenses for month 3: 150.50  
 Enter expenses for month 4: 300  
 Enter expenses for month 5: 100.50  
 Enter expenses for month 6: 34.25  
 Enter expenses for month 7: 45.75  
 Enter expenses for month 8: 195.00  
 Enter expenses for month 9: 123.45  
 Enter expenses for month 10: 111.11  
 Enter expenses for month 11: 222.20  
 Enter expenses for month 12: 120.00

Month 1 = \$100.00  
 Month 2 = \$200.12  
 Month 3 = \$150.50  
 Month 4 = \$300.00  
 Month 5 = \$100.50  
 Month 6 = \$34.25  
 Month 7 = \$45.75



## Arreglos numéricos

```
Month 8 = $195.00  
Month 9 = $123.45  
Month 10 = $111.11  
Month 11 = $222.20  
Month 12 = $120.00
```

### ANÁLISIS

Cuando se ejecuta EXPENSES.C, el programa le pide que teclee los gastos para los meses del 1 al 12. Los valores que se teclean son guardados en un arreglo. Se debe dar algún valor para cada mes. Después de que se da el duodécimo valor se despliega en la pantalla el contenido del arreglo.

El flujo del programa es similar a los listados que se han visto anteriormente. La línea 1 comienza con un comentario que describe lo que el programa va a hacer. Observe que está incluido el nombre del programa, EXPENSES.C. Incluyendo el nombre del programa en un comentario se sabe qué programa se está viendo. Esto es útil cuando se imprimen los listados y luego se quiere hacer algún cambio.

La línea 5 contiene un comentario adicional, con una explicación sobre las variables que se están declarando. En la línea 7 es declarado un arreglo de 13 elementos. En este programa sólo se necesitan 12 elementos, uno para cada mes, pero han sido declarados 13. El ciclo `for` que se encuentra en las líneas 14 a 18 ignora al elemento 0. Esto permite que el programa use los elementos del 1 al 12, que están relacionados directamente con los 12 meses. Regresando, en la línea 8 es declarada una variable, `count`, y usada a lo largo del programa como contador e índice de arreglo.

La función `main()` del programa comienza en la línea 10. Como se dijo anteriormente, el programa usa un ciclo `for` para imprimir un mensaje y aceptar un valor para cada uno de los 12 meses. Observe que en la línea 17 la función `scanf()` usa un elemento de arreglo. En la línea 7 el arreglo `expenses` fue declarado como `float`, por lo que se usa `%f`. También es puesto el operador de *dirección de (&)* antes del elemento del arreglo, como si fuera una variable regular tipo `float` y no un elemento de arreglo.

Las líneas 22 a 25 contienen un segundo ciclo `for`, que imprime los valores que se acaban de teclear. Se ha añadido un comando de formateo adicional a la función `printf()`, para que de esta forma los valores de gastos se impriman en forma mejor ordenada. Por ahora, simplemente sepa que `% .2f` imprime un número de punto flotante con dos decimales. En el Día 14, “Trabajando con la pantalla, la impresora y el teclado”, se tratan comandos adicionales para el formateo.

### DEBE

### NO DEBE

**NO DEBE** Olvidar que los subíndices de arreglo comienzan con el elemento 0.

**DEBE** Usar arreglos en vez de crear varias variables que guarden la misma cosa.

(Por ejemplo, si se quieren guardar las ventas totales para cada mes del año, cree un arreglo con 12 elementos que guarden las ventas, en vez de crear una variable de ventas para cada mes.)

## Arreglos multidimensionales

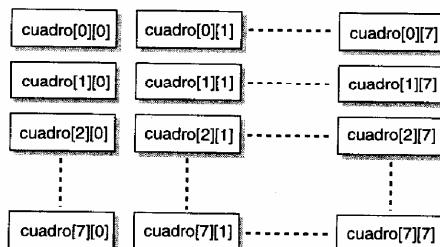
Un arreglo multidimensional tiene más de un subíndice. Un *arreglo bidimensional* tiene dos subíndices, un *arreglo tridimensional* tiene tres subíndices y así sucesivamente. No hay límite a la cantidad de dimensiones que pueda tener un arreglo en C. (Hay un límite sobre el tamaño total del arreglo, que se mencionará posteriormente en este capítulo.)

Por ejemplo, tal vez quiera escribir un programa que juegue damas. El tablero de damas contiene 64 cuadros, acomodados en 8 renglones y 8 columnas. El programa podría representar al tablero como un arreglo bidimensional de la manera siguiente:

```
int cuadro[8][8];
```

El arreglo resultante tiene 64 elementos: `cuadro[0][0]`, `cuadro[0][1]`, `cuadro[0][2]`... `cuadro[7][6]`, `cuadro[7][7]`. La estructura de este arreglo bidimensional se ilustra en la figura 8.3.

```
int cuadro[8][8];
```



**Figura 8.3.** Un arreglo bidimensional tiene una estructura de columnas y renglones.

De manera similar, se podría pensar en un arreglo tridimensional para un cubo. Dejamos a su imaginación a los arreglos de cuatro dimensiones (y superiores). Todos los arreglos, sin importar qué tantas dimensiones tengan, son guardados secuencialmente en memoria. En el Día 15, “Más sobre apuntadores”, se dan más detalles sobre el almacenamiento de arreglos.

## Denominación y declaración de arreglos

Las reglas para asignar nombres a los arreglos son las mismas que para los nombres de variables, que fueron tratadas en el Día 3, “Variables y constantes numéricas”. Un nombre de arreglo debe ser único. No puede ser usado para otro arreglo o para cualquier otro identificador (variable, constante, etc.). Como probablemente se ha dado cuenta, las declaraciones de arreglo siguen la misma forma que las variables que no son de arreglo, a excepción de que la cantidad de elementos del arreglo debe ser encerrada entre corchetes y puesta inmediatamente después del nombre del arreglo.

Cuando se declara un arreglo se puede especificar la cantidad de elementos con una constante literal (como se ha hecho en los ejemplos anteriores) o con una constante simbólica creada con la directiva `#define`. Por lo tanto,

```
#define MESES 12
int arreglo[MESES];
```

es equivalente a

```
int arreglo[12];
```

Sin embargo, en la mayoría de los compiladores no se pueden declarar los elementos del arreglo con una constante simbólica creada con la palabra clave `const`.

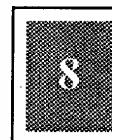
```
const int MESES = 12;
int arreglo[MESES];           /* ;Erróneo! */
```

El listado 8.2, GRADES.C, es otro programa que muestra el uso de un arreglo unidimensional. GRADES.C usa un arreglo para guardar 10 calificaciones escolares.



### Listado 8.2. GRADES.C.

```
1:  /* GRADES.C - Programa de ejemplo con un arreglo */
2:  /* Pide 10 grados escolares y luego calcula el promedio */
3:
4:  #include <stdio.h>
5:
6:  #define MAX_GRADE 100
7:  #define STUDENTS 10
8:
9:  int grades[STUDENTS];
10:
11: int idx;
12: int total = 0;           /* usado para promedio */
13:
14: main()
15: {
16:     for( idx=0;idx<1-STUDENTS;idx++)
```



```
17:      {
18:          printf( "Enter Person %d's grade: ", idx +1);
19:          scanf( "%d", &grades[idx] );
20:
21:          while ( grades[idx] > MAX_GRADE )
22:          {
23:              printf( "\nThe highest grade possible is %d",
24:                      MAX_GRADE );
25:              printf( "\nEnter correct grade: " );
26:              scanf( "%d", &grades[idx] );
27:
28:              total += grades[idx];
29:          }
30:
31:      printf( "\n\nThe average score is %d", ( total / STUDENTS ) );
32:
33:      return (0);
34: }
```

```
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

De manera similar a EXPENSES.C, este listado le pide datos al usuario. Le pide las calificaciones grados de 10 gentes. En vez de imprimir cada calificación, imprime el promedio.

Como ha visto anteriormente, los arreglos son nombrados de manera similar a las variables regulares. En la línea 9, el arreglo para este programa es llamado `grades`. Es correcto suponer que este arreglo guarda calificaciones escolares. En las líneas 6 y 7, se definen dos constantes, `MAX_GRADE` y `STUDENTS` (grado máximo y estudiantes). Estas constantes pueden ser cambiadas fácilmente. Sabiendo que `STUDENTS` está definido como 10, se sabe que el arreglo `grades` tiene 10 elementos. En el listado están declaradas otras dos variables, `idx` y `total`. Se usa `idx`, una abreviatura para índice, como contador y subíndice del arreglo. Un gran total de todos los grados se guarda en `total`.

La parte modular de este programa es el ciclo `for`, que se encuentra en las líneas 16 a 29. El enunciado `for` inicializa `idx` a 0, el primer subíndice del arreglo. Luego se mantiene

haciendo ciclos mientras `idx` sea menor que la cantidad de estudiantes. Cada vez que inicia el ciclo, incrementa a `idx` en 1. En cada ciclo el programa pide la calificación de la persona (líneas 18 y 19). Observe que en la línea 18 se añade 1 a `idx`, para contar a la gente de 1 a 10 en vez de 0 a 9. Debido a que cualquier arreglo comienza con el subíndice 0, la primera calificación es puesta en `grade[0]`. En vez de confundir al usuario pidiéndole la calificación de la persona 0, se le pide la calificación de la persona 1.

Las líneas 21 a 26 contienen un ciclo `while` anidado dentro del ciclo `for`. Esta es una revisión de edición, que asegura que el grado no sea mayor que el grado máximo, `MAX_GRADE`. Se le pide al usuario teclear una calificación correcta, en caso de que haya dado una calificación que sea demasiado alta. Se deben revisar los datos del programa cada vez que sea posible.

La línea 28 suma el grado tecleado a un contador total. En la línea 31 es usado este total para imprimir el grado promedio (`total/STUDENTS`).

### DEBE

### NO DEBE

**DEBE** Usar enunciados `#define` para crear constantes que puedan usarse cuando se declaran arreglos. De esta manera podrá cambiar fácilmente la cantidad de elementos del arreglo. En `GRADES.C` se podría cambiar la cantidad de estudiantes en `#define` y no se tendría que hacer ningún otro cambio en el programa.

**DEBE** Evitar los arreglos multidimensionales con más de tres dimensiones. Recuerde que los arreglos multidimensionales pueden hacerse muy grandes rápidamente.

## Inicialización de arreglos

Se puede inicializar a todo o parte del arreglo cuando se le declara. Ponga a continuación de la declaración de arreglo un signo de igual y una lista de valores, encerrados entre llaves y separados por comas. Los valores listados son asignados en orden a los elementos del arreglo, comenzando con el número 0. Por ejemplo,

```
int arreglo[4] = { 100, 200, 300, 400 };
```

asigna el valor 100 a `arreglo[0]`, 200 a `arreglo[1]`, 300 a `arreglo[2]` y 400 a `arreglo[3]`. Si se omite el tamaño del arreglo, el compilador crea un arreglo lo suficientemente grande como para guardar los valores de inicialización. Por lo tanto, el enunciado

```
int arreglo[] = { 100, 200, 300, 400 };
```



tendría exactamente el mismo efecto que el enunciado de declaración de arreglo anterior. Sin embargo, se pueden incluir menos valores de inicialización, como se ve en este ejemplo:

```
int arreglo[10] = {1, 2, 3};
```

Si no se inicializa explícitamente a los elementos del arreglo, no se puede estar seguro del valor que contienen cuando ejecuta el programa. Si se incluyen demasiados valores inicializadores (más inicializadores que elementos de arreglo), el compilador detecta un error.

Los arreglos multidimensionales también pueden ser inicializados. La lista de valores de inicialización es asignada en orden a los elementos del arreglo, cambiando primero el último subíndice del arreglo. Por ejemplo,

```
int arreglo[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

da como resultado las siguientes asignaciones:

```
arreglo[0][0] es igual a 1  
arreglo[0][1] es igual a 2  
arreglo[0][2] es igual a 3  
arreglo[1][0] es igual a 4  
arreglo[1][1] es igual a 5  
arreglo[1][2] es igual a 6  
...  
arreglo[3][1] es igual a 11  
arreglo[3][2] es igual a 12
```

Cuando se inicializan arreglos multidimensionales, se puede hacer más claro el código fuente usando llaves adicionales para agrupar los valores de inicialización, y también distribuyéndolos en varias líneas. La siguiente inicialización es equivalente a la anterior:

```
int arreglo[4][3] = {{ 1, 2, 3 }, { 4, 5, 6 },  
                     { 7, 8, 9 }, { 10, 11, 12 } };
```

Recuerde, los valores de inicialización deben estar separados por comas, incluso aunque haya llaves entre ellos. También asegúrese de usar las llaves en pares, una llave derecha para cada llave izquierda, ya que si no lo hace el compilador se confunde.

Ahora veamos un ejemplo que muestra las ventajas de los arreglos. El programa del listado 8.3, RANDOM.C, crea un arreglo tridimensional de 1000 elementos y lo llena con números al azar. Luego el programa despliega en la pantalla los elementos del arreglo. Imagine qué tantas líneas de código fuente necesitaría para ejecutar la misma tarea con variables que no sean de arreglo.

En este programa verá una nueva función de biblioteca, getch(). La función getch() lee un solo carácter del teclado. En el listado 8.3 getch() hace pausa en el programa hasta que el usuario oprima una tecla. getch() se trata a detalle en el Día 14, "Trabajando con la pantalla, la impresora y el teclado".

## Arreglos numéricos

---



### Listado 8.3. RANDOM.C.

```

1:  /* RANDOM.C - Demostración del uso de un arreglo multidimensional */
2:
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:  /* Declara un arreglo tridimensional con 1000 elementos */
6:
7:  int random[10][10][10];
8:  int a, b, c;
9:
10: main()
11: {
12:     /* Llena el arreglo con números al azar. La función de */
13:     /* biblioteca del C, rand(), regresa un número al azar. */
14:     /* Usa un ciclo for para cada subíndice del arreglo. */
15:
16:     for (a = 0; a < 10; a++)
17:     {
18:         for (b = 0; b < 10; b++)
19:         {
20:             for (c = 0; c < 10; c++)
21:             {
22:                 random[a][b][c] = rand();
23:             }
24:         }
25:     }
26:
27:     /* Ahora despliega los elementos del arreglo de 10 en 10 */
28:
29:     for (a = 0; a < 10; a++)
30:     {
31:         for (b = 0; b < 10; b++)
32:         {
33:             for (c = 0; c < 10; c++)
34:             {
35:                 printf("\nrandom[%d][%d][%d] = ", a, b, c);
36:                 printf("%d", random[a][b][c]);
37:             }
38:             printf("\nPress a key to continue, CTRL-C to \
39:                   quit.");
40:             getch();
41:         }
42:     }      /* fin de main() */

```

---



random[0][0][0] = 346  
random[0][0][1] = 130  
random[0][0][2] = 10982



```
random[0][0][3] = 1090
random[0][0][4] = 11656
random[0][0][5] = 7117
random[0][0][6] = 17595
random[0][0][7] = 6415
random[0][0][8] = 22948
random[0][0][9] = 31126
Press a key to continue, CTRL-C to quit.
random[0][1][0] = 9004
random[0][1][1] = 14558
random[0][1][2] = 3571
random[0][1][3] = 22879
random[0][1][4] = 18492
random[0][1][5] = 1360
random[0][1][6] = 5412
random[0][1][7] = 26721
random[0][1][8] = 22463
random[0][1][9] = 25047
Press a key to continue, CTRL-C to quit
...
random[9][8][0] = 6287
random[9][8][1] = 26957
random[9][8][2] = 1530
random[9][8][3] = 14171
random[9][8][4] = 6951
random[9][8][5] = 213
random[9][8][6] = 14003
random[9][8][7] = 29736
random[9][8][8] = 15028
random[9][8][9] = 18968
Press a key to continue, CTRL-C to quit.
random[9][9][0] = 28559
random[9][9][1] = 5268
random[9][9][2] = 20182
random[9][9][3] = 3633
random[9][9][4] = 24779
random[9][9][5] = 3024
random[9][9][6] = 10853
random[9][9][7] = 28205
random[9][9][8] = 8930
random[9][9][9] = 2873
Press a key to continue, CTRL-C to quit.
```



En el Día 6, "Control básico del programa", se vio un programa que usaba un enunciado `for` anidado. Este programa tiene dos ciclos `for` anidados. Antes de que vea a detalle los enunciados `for`, observe que las líneas 7 y 8 declaran cuatro variables. La primera es un arreglo llamado `random`, usado para guardar los números al azar. `random` es un arreglo tipo `int` de tres dimensiones de 10 por 10 por 10, dando un total de 1,000 elementos tipo `int` ( $10 \times 10 \times 10$ ). Imagínese tener que tratar con 1,000 nombres de variable únicos si no fuera posible usar arreglos. Luego la línea 8 declara tres variables, `a`, `b` y `c`, usadas para control de los ciclos `for`.



## 8 Arreglos numéricos

Este programa también incluye en la línea 4 un nuevo archivo de encabezado, **STDLIB.H** (que significa standard library: biblioteca estándar). Este es incluido para proporcionar el prototipo para la función **rand()**, que es usada en la línea 22.

El grueso del programa está contenido en dos enunciados **for** anidados. El primero se encuentra en las líneas 16 a 25 y el segundo en las líneas 29 a 41. Ambos **for** anidados tienen la misma estructura. Trabajan de manera similar a los ciclos del listado 6.2 pero van a un nivel más profundo. En el primer juego de enunciados **for**, la línea 22 se ejecuta repetidamente. La línea 22 asigna el valor de retorno de una función, **rand()**, a un elemento del arreglo **random**. **rand()** es una función de biblioteca que regresa un número al azar.

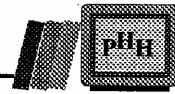
Regresando en el listado se puede ver que la línea 20 cambia a la variable **c** de 0 a 9. Esto hace ciclo por el subíndice de la extrema derecha del arreglo **random**. La línea 18 hace ciclos sobre **b**, el subíndice medio del arreglo **random**. Cada vez que cambia a **b**, hace ciclo por todos los elementos **c**. La línea 16 incrementa la variable **a**, que hace ciclo por el subíndice de la extrema izquierda. Cada vez que cambia este subíndice hace ciclo por todos los 10 valores del subíndice **b**, que a su vez hace ciclo por todos los 10 valores de **c**. Este ciclo inicializa a cada uno de los valores del arreglo **random** con un número al azar.

Las líneas 29 a 41 contienen el segundo anidamiento de enunciados **for**. Funcionan de manera similar a los enunciados **for** anteriores, pero este ciclo imprime cada uno de los valores asignados anteriormente. Despues de que son desplegados 10, la línea 38 imprime un mensaje y espera a que se oprima una tecla. La línea 39 se ocupa de la opresión de tecla. **getch()** regresa el valor de la tecla que ha sido oprimida. Si no se oprime una tecla, **getch()** espera hasta que suceda. Ejecute este programa y observe los valores desplegados.

## Tamaño máximo del arreglo

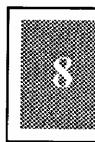
Debido a la manera en que funcionan los modelos de memoria, se debe tratar, por ahora, de no crear más de 64 K de datos en variables. La explicación de este límite se encuentra más allá del alcance de este libro, pero no hay por qué preocuparse, ya que ninguno de los programas de este libro excede esta limitación. Para aprender más, o para resolver esta limitación, consulte los manuales del compilador. Por lo general 64 K es suficiente espacio de datos para los programas, en particular para los programas relativamente simples que se escribirán mientras estudie con este libro. Un solo arreglo puede ocupar los 64 K del almacenamiento de datos, si el programa no usa otras variables. De otra forma se necesita repartir el espacio de datos disponible en la manera adecuada.

El tamaño de un arreglo en bytes depende de la cantidad de elementos que tiene y del tamaño de cada elemento. El tamaño del elemento depende del tipo de dato del arreglo. Los tamaños para cada tipo de dato numérico, dados en la tabla 3.2, son repetidos aquí en la tabla 8.1 para su comodidad.



**Tabla 8.1. Requerimientos de espacio de almacenamiento para los tipos de datos numéricos de la mayoría de las PC.**

Elemento	
Tipo de dato	Tamaño (Bytes)
int	2
short	2
long	4
float	4
double	8



Para calcular el espacio de almacenamiento requerido para un arreglo, se multiplica la cantidad de elementos del arreglo por el tamaño del elemento. Por ejemplo, un arreglo de 500 elementos de tipo float requiere  $(500) * (4) = 2000$  bytes de espacio de almacenamiento.

El espacio de almacenamiento puede ser determinado dentro de un programa usando el operador `sizeof()` del C. `sizeof()` es un operador unario y no una función. Toma como argumento el nombre de una variable o el nombre de un tipo de dato y regresa el tamaño en bytes de su argumento. El uso de `sizeof()` es ilustrado en el listado 8.4.



**Listado 8.4. Uso del operador `sizeof()` para determinar los requerimientos de espacio de almacenamiento para un arreglo.**

```
1: /* Demostración del operador sizeof() */
2:
3: #include <stdio.h>
4:
5: /* Declara varios arreglos de 100 elementos */
6:
7: int intarray[100];
8: float floatarray[100];
9: double doublearray[100];
10:
11: main()
12: {
13:     /* Despliega el tamaño de los tipos de datos numéricos */
14:
15:     printf("\n\nSize of int = %d bytes", sizeof(int));
16:     printf("\nSize of short = %d bytes", sizeof(short));
17:     printf("\nSize of long = %d bytes", sizeof(long));
```

**Listado 8.4. continuación**

```

18:     printf("\nSize of float = %d bytes", sizeof(float));
19:     printf("\nSize of double = %d bytes", sizeof(double));
20:
21:     /* Despliega el tamaño de los tres arreglos */
22:
23:     printf("\nSize of intarray = %d bytes", sizeof(intarray));
24:     printf("\nSize of floatarray = %d bytes",
25:            sizeof(floatarray));
26:     printf("\nSize of doublearray = %d bytes",
27:            sizeof(doublearray));
    }

```



Size of int = 2 bytes  
 Size of short = 2 bytes  
 Size of long = 4 bytes  
 Size of float = 4 bytes  
 Size of double = 8 bytes  
 Size of intarray = 200 bytes  
 Size of floatarray = 400 bytes  
 Size of doublearray = 800 bytes



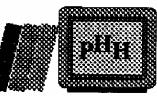
Teclee y compile el programa de este listado usando los procedimientos que aprendió en el Día 1, “Comienzo”. Cuando el programa ejecuta, despliega el tamaño en bytes de los tres arreglos y de los cinco tipos de datos numéricos.

En el Día 3, “Variables y constantes numéricas”, se ejecutó un programa similar. Sin embargo, este listado usa `sizeof()` para determinar el tamaño de almacenamiento de arreglos. Las líneas 7, 8 y 9 declaran tres arreglos, cada uno de diferente tipo. Las líneas 23, 24 y 25 imprimen el tamaño de cada arreglo. El tamaño debe ser igual al tamaño del tipo de variable del arreglo multiplicado por la cantidad de elementos. Por ejemplo, si un `int` es de dos bytes, `intarray` debe ser 2 por 100, o sea 200 bytes. Ejecute el programa y revise los valores.

## Resumen

Este capítulo presenta los arreglos numéricos, un método poderoso de almacenamiento de datos que le permite agrupar una cantidad de conceptos de datos del mismo tipo bajo el mismo nombre de grupo. Los conceptos individuales, o elementos del arreglo, son identificados usando un subíndice después del nombre del arreglo. Las tareas de programación de computadora que involucran el procesamiento repetitivo de datos lo llevan por sí mismas al almacenamiento en arreglos.

De manera similar a las variables que no son de arreglos, los arreglos deben ser declarados antes de que puedan usarse. En forma opcional, los elementos del arreglo pueden ser inicializados cuando es declarado el arreglo.



## Preguntas y respuestas

1. ¿Qué pasa si uso en un arreglo un subíndice que es mayor a la cantidad de elementos del arreglo?

Si se usa un subíndice que está fuera de los límites de la declaración del arreglo, es probable que el programa compile y hasta corra. Sin embargo, los resultados de este error son impredecibles. Este puede ser un error difícil de encontrar cuando empieza a causar problemas, por lo que debe asegurarse de ser cuidadoso cuando inicialice y accese los elementos del arreglo.

2. ¿Qué pasa si uso un arreglo sin inicializarlo?

Este error no produce errores de compilación. Si no se inicializa un arreglo habrá cualquier valor en los elementos del arreglo. Se pueden obtener resultados impredecibles. Se debe siempre inicializar las variables y los arreglos, para que de esta forma se sepa exactamente lo que hay en ellos. En el Día 12, "Alcance de las variables", se presenta una excepción a la necesidad de inicialización. Por el momento asegúrese de hacerlo.

3. ¿Qué tantas dimensiones puede tener un arreglo?

Como se dijo en el capítulo, se pueden tener tantas dimensiones como quiera. Conforme añade más dimensiones se usa más espacio de almacenamiento de datos. Se debe declarar al arreglo tan grande como se necesite para evitar el desperdicio de espacio de almacenamiento.

4. ¿Hay alguna manera fácil de inicializar todo un arreglo de un jalón?

Cada elemento de un arreglo debe ser inicializado. La manera más segura para que un programador de C novato inicialice un arreglo es con una declaración, como se mostró en este capítulo, o con un enunciado `for`. Hay otras formas de inicializar un arreglo, pero están más allá del alcance de este capítulo y del libro en este momento.

5. ¿Puedo sumar dos arreglos (o multiplicarlos, dividirlos o restarlos)?

Si se declaran dos arreglos no se les puede sumar. Cada elemento debe ser sumado individualmente. El ejercicio 10 ilustra este punto.

6. ¿Por qué es mejor usar un arreglo en vez de variables individuales?

Con los arreglos se pueden agrupar valores con un solo nombre. En el listado 8.3 fueron guardados 1,000 valores. La creación de 1,000 nombres de variable, y la inicialización de cada una de ellas a un número al azar, habría necesitado una tremenda cantidad de tecleo. Usando un arreglo se facilita la tarea.



## Arreglos numéricos

# Taller

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

## Cuestionario

1. ¿Cuáles de los tipos de datos del C pueden usarse en un arreglo?
2. Si se declara un arreglo con 10 elementos, ¿cuál es el subíndice del primer elemento?
3. En un arreglo de una sola dimensión declarado con n elementos, ¿cuál es el subíndice del último elemento?
4. ¿Qué pasa si el programa trata de accesar un elemento con un subíndice fuera de rango?
5. ¿Cómo se declara un arreglo multidimensional?
6. Un arreglo es declarado con el enunciado

```
int arreglo[2][3][5][8];
```

¿Qué tantos elementos tiene el arreglo?

## Ejercicios

1. Escriba una línea de programa en C que declare tres arreglos enteros de una dimensión, llamados uno, dos y tres, con 1,000 elementos cada uno.
2. Escriba los enunciados necesarios para declarar un arreglo entero de 10 elementos e inicializar todos sus elementos a 1.
3. Dado el arreglo

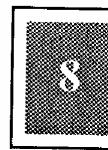
```
int ochentayocho[88];
```

escriba el código para inicializar a todos los elementos del arreglo a 88.

4. Dado el arreglo

```
int cosa[12][10];
```

escriba el código para inicializar todos los elementos del arreglo a 0.



5. BUSQUEDA DE ERRORES: ¿Qué hay de erróneo en el siguiente fragmento de código?

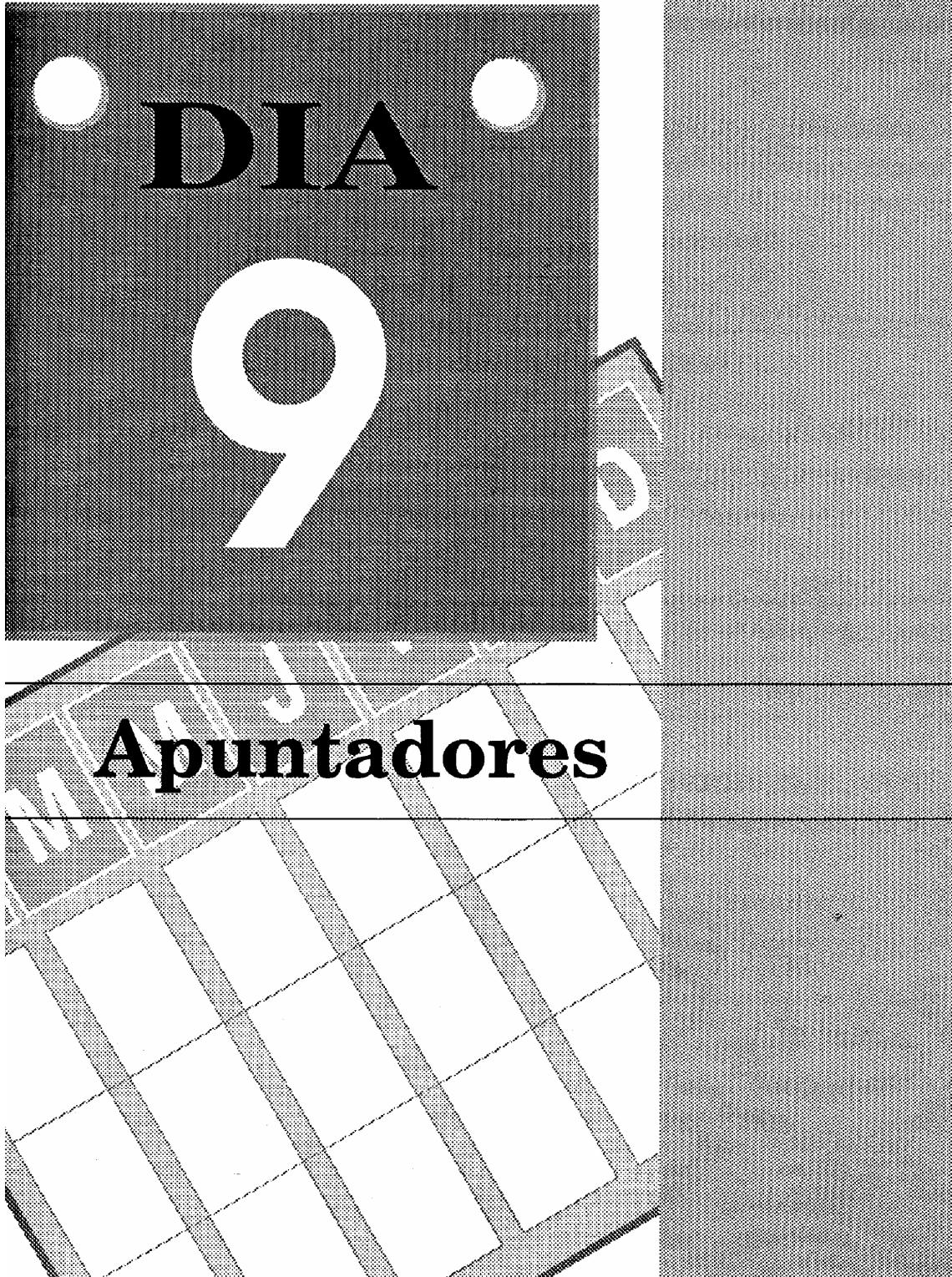
```
int x, y;
int array[10][3];
main()
{
    for ( x = 0; x < 3; x++ )
        for ( y = 0; y < 10; y++ )
            array[x][y] = 0;
}
```

6. BUSQUEDA DE ERRORES: ¿Qué error tiene lo siguiente?

```
int array[10];
int x = 1;

main()
{
    for ( x = 1; x <= 10; x++ )
        array[x] = 99;
}
```

7. Escriba un programa que ponga números al azar en un arreglo bidimensional de 5 por 4. Imprima en la pantalla los valores en columnas. (Consejo: Use la función rand() del listado 8.3.)
8. Vuelva a escribir el listado 8.3 para usar un arreglo de una sola dimensión. Imprima el promedio de las 1,000 variables antes de imprimir los valores individuales. Nota: No olvide hacer una pausa después de imprimir cada 10 valores.
9. Escriba un programa que inicialice un arreglo de 10 elementos. El valor de cada elemento debe ser igual a su subíndice. Luego, el programa debe imprimir cada uno de los 10 elementos.
10. Modifique el programa del ejercicio 9. Después de imprimir los valores inicializados, el programa debe copiar los valores a un nuevo arreglo y sumar 10 a cada valor. Luego se deben imprimir los valores del nuevo arreglo.





## Apuntadores

Este capítulo le presenta los apuntadores, que son una parte importante del lenguaje C. Los apuntadores proporcionan un método poderoso y flexible para manejar los datos en el programa. Hoy aprenderá

- La definición de un apuntador.
- Los usos de los apuntadores.
- La manera de declarar e inicializar apuntadores.
- La manera de usar apuntadores con variables simples y arreglos.
- La manera de usar apuntadores para pasar arreglos a funciones.

Conforme lea el capítulo, tal vez no sean evidentes inmediatamente las ventajas del uso de apuntadores. Las ventajas caen en dos categorías: cosas que se pueden hacer mejor con apuntadores que sin ellos, y cosas que pueden ser hechas solamente con apuntadores. Los puntos específicos se irán aclarando conforme lea este capítulo y los siguientes. Por el momento, simplemente sepa que debe entender los apuntadores, si quiere ser un programador en C hábil.

## ¿Qué es un apuntador?

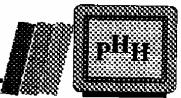
Para comprender a los apuntadores se necesita un conocimiento básico sobre la manera en que la computadora guarda la información en memoria. Lo que se indica a continuación es una explicación algo simplificada sobre el almacenamiento de memoria de la PC.

### La memoria de la computadora

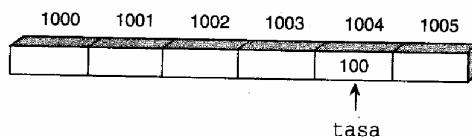
La RAM de la PC consiste en muchos miles de posiciones de almacenamiento secuenciales, y cada posición se identifica por una dirección única. Las direcciones de memoria en una computadora dada van desde 0 al valor máximo, que depende de la cantidad de memoria instalada.

Cuando se está usando una computadora, el sistema operativo usa algo de la memoria del sistema. Cuando se está ejecutando un programa, el código y los datos del programa (las instrucciones de lenguaje de máquina para las diversas tareas del programa y la información que el programa está usando, respectivamente) también usan algo de la memoria del sistema. Esta sección examina el almacenamiento en memoria para los datos del programa.

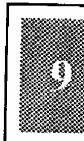
Cuando se declara una variable en un programa C, el compilador reserva una posición de memoria con una dirección única para guardar esa variable. El compilador asocia esa dirección con el nombre de la variable. Cuando el programa usa el nombre de la variable, accesa automáticamente la posición de memoria adecuada. Está siendo usada la dirección de la ubicación pero se encuentra oculta y uno no tiene necesidad de saberlo.



La figura 9.1 muestra esto esquemáticamente. Una variable, llamada `tasa`, ha sido declarada e inicializada a 100. El compilador ha reservado espacio de almacenamiento en la dirección 1004 para la variable, y ha asociado el nombre `tasa` con la dirección 1004.

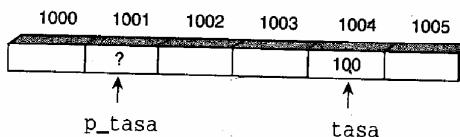


**Figura 9.1.** Una variable de programa es guardada en una dirección de memoria específica



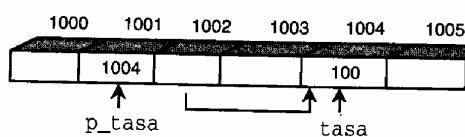
## Creación de un apuntador

Se debe observar que la dirección de la variable `tasa` (o de cualquier otra variable) es un número y puede ser tratado como cualquier otro número en C. Si se sabe la dirección de una variable, se puede crear una segunda variable donde se guarde la dirección de la primera. El primer paso es declarar una variable para guardar la dirección de `tasa`. Démolas el nombre, por ejemplo, `p_tasa`. Al principio `p_tasa` se encuentra sin inicializar. Se ha reservado espacio de almacenamiento para `p_tasa`, pero su valor se encuentra indeterminado. Esto se muestra en la figura 9.2.



**Figura 9.2.** El espacio de almacenamiento de memoria ha sido ubicado para la variable `p_tasa`.

El siguiente paso es guardar la dirección de la variable `tasa` en la variable `p_tasa`. Debido a que `p_tasa` ahora contiene la dirección de `tasa`, indica su posición de almacenamiento en memoria. En la manera de hablar del C, `p_tasa` apunta a `tasa`, o es un apuntador a `tasa`. Esto está diagramado en la figura 9.3.



**Figura 9.3.** La variable `p_tasa` contiene la dirección de la variable `tasa`, y es, por lo tanto, un apuntador a `tasa`.



Resumiendo, un apuntador es una variable que contiene la dirección de otra variable. Ahora podemos ir a los detalles sobre el uso de apuntadores en los programas en C.

# Los apuntadores y las variables simples

En los ejemplos que se han dado, una variable de apuntador apuntaba a una variable simple (esto es, que no es un arreglo). Esta sección le muestra cómo crear y usar apuntadores a variables simples.

## Declaración de apuntadores

Un apuntador es una variable numérica y, de manera similar a todas las variables, debe ser inicializada antes de que pueda ser usada. Los nombres de variables de apuntador siguen las mismas reglas que otras variables y deben ser únicos. Este capítulo usa la convención de que un apuntador a la variable *nombre* es llamado *p\_nombre*. Esto no es necesario, ya que se puede nombrar a los apuntadores en cualquier forma que se desee (siguiendo las reglas del C).

Una declaración de apuntador toma la siguiente forma:

*nombre\_de\_tipo \*nombre\_de\_apuntador;*

*nombre\_de\_tipo* es cualquiera de los tipos de variable del C, e indica el tipo de variable a la cual apunta el apuntador. El asterisco (\*) es el *operador de indirección*, e indica que *nombre\_de\_apuntador* es un apuntador al tipo *nombre\_de\_tipo* y no una variable de tipo *nombre\_de\_tipo*. Los apuntadores pueden ser declarados junto con variables que no son apuntadores. A continuación se presentan algunos ejemplos:

```
char *ch1, *ch2; /* ch1 y ch2 son apuntadores a tipo char */
float *valor, porcentaje; /* valor es un apuntador a tipo float */
/* y porcentaje es una variable float ordinaria */
```

El símbolo \* es usado tanto como *operador de indirección* como operador de multiplicación. No se preocupe pensando que el compilador puede confundirse. El contexto donde se usa el \* siempre proporciona suficiente información, de tal forma que el compilador puede imaginarse si se trata de indirección o de multiplicación.

## Inicialización de apuntadores

Ahora que ya se ha declarado un apuntador, ¿qué se puede hacer con él? No puede hacer nada con él mientras no haga que apunte a algo. De manera similar a las variables regulares, los apuntadores sin inicializar pueden ser usados, pero los resultados son impredecibles y



potencialmente desastrosos. Mientras un apuntador no guarde la dirección de una variable, es inútil. La dirección no se almacena mágicamente en el apuntador, sino que el programa debe ponerla ahí usando el operador de dirección *de*, (&). Cuando es puesto antes del nombre de una variable, el operador de dirección *de* regresa la dirección de la variable. Por lo tanto, se inicializa un apuntador con un enunciado de la forma

```
apuntador = &variable;
```

Regresemos al ejemplo de la figura 9.3. El enunciado de programa para inicializar a la variable *p\_tasa* para que apunte a la variable *tasa*, debe ser

```
p_tasa = &tasa; /* asigna la dirección de tasa a p_tasa */
```

Antes de la inicialización *p\_tasa* no apuntaba a nada en particular. Después de la inicialización *p\_tasa* es un apuntador a *tasa*.



## Uso de apuntadores

Ahora que ya sabemos cómo declarar e inicializar los apuntadores, probablemente se pregunte cómo se les usa. Aquí entra nuevamente en juego el operador (\*) de indirección. Cuando el \* precede al nombre de un apuntador, hace referencia a la variable a la que apunta.

Continuemos con el ejemplo anterior, donde el apuntador *p\_tasa* ha sido inicializado para que apunte a la variable *tasa*. Si se escribe *\*p\_tasa* se hace referencia a la variable *tasa*. Si se quiere imprimir el valor de *tasa* (que es de 100 en este ejemplo) se podría escribir

```
printf(%d, tasa);
```

o se podría escribir

```
printf(%d, *p_tasa);
```

En C los dos enunciados son equivalentes. Accesar el contenido de una variable usando el nombre de la variable es llamado *acceso directo*. Accesar el contenido de una variable usando un apuntador a la variable es llamado *acceso indirecto o indirección*. La figura 9.4 ilustra que un nombre de apuntador precedido por el operador de indirección hace referencia al valor de la variable apuntada.

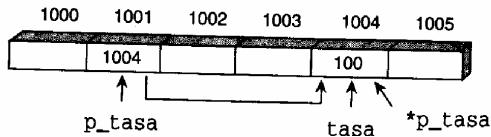


Figura 9.4. Uso del operador de indirección con apuntadores.



## Apuntadores

---

Haga una pausa y piense acerca de esto. Los apuntadores son una parte integral del lenguaje C y es esencial que los comprenda. Los apuntadores han confundido a mucha gente, por lo que no se preocupe si a usted también le pasa. Si necesita revisarlo, es correcto. Tal vez el siguiente resumen le ayude. Si tiene un apuntador llamado `ptr`, que ha sido inicializado para que apunte a la variable `var`, entonces

`*ptr` y `var` hacen referencia al contenido de `var` (esto es, a cualquier valor que el programa haya almacenado ahí).

`ptr` y `&var` hacen referencia a la dirección de `var`.

Como puede ver, un nombre de apuntador sin el operador de indirección accesa el propio valor del apuntador, el cual es, por supuesto, la dirección de la variable a la que apunta.

El programa que se encuentra en el listado 9.1 muestra el uso básico de los apuntadores. Usted debe teclear, compilar y ejecutar este programa.



### Listado 9.1. Ilustración del uso básico de apuntadores.

---

```
1: /* Demuestra el uso básico de apuntadores */
2:
3: #include <stdio.h>
4:
5: /* Declara e inicializa una variable int */
6:
7: int var = 1;
8:
9: /* Declara un apuntador a int */
10:
11: int *ptr;
12:
13: main()
14: {
15:     /* Inicializa ptr para que apunte a var */
16:
17:     ptr = &var;
18:
19:     /* Accesa var directa e indirectamente */
20:
21:     printf("\nDirect access, var = %d", var);
22:     printf("\nIndirect access, var = %d", *ptr);
23:
24:     /* Despliega la dirección de var de dos maneras */
25:
26:     printf("\n\nThe address of var = %d", &var);
27:     printf("\n\nThe address of var = %d", ptr);
28: }
```

---



A continuación se muestra la salida de este programa. Las direcciones reportadas para `var` tal vez no sean 96 en su sistema.



```
Direct access, var = 1  
Indirect access, var = 1  
The address of var = 96  
The address of var = 96
```



En este listado se declaran dos variables. En la línea 7 es declarada var como int e inicializada a 1. En la línea 11 es declarado un apuntador a una variable de tipo int y denominado ptr. En la línea 17, al apuntador ptr le es asignada la dirección de var usando el operador de *dirección de* (&). El resto del programa imprime en la pantalla los valores de estas dos variables. La línea 21 imprime el valor de var, y la línea 22 imprime el valor guardado en la posición apuntada por ptr. En este programa el valor es 1. La línea 26 imprime la dirección de var usando al operador de *dirección de*. Este es el mismo valor que es impreso por la línea 27 usando la variable de apuntador, ptr.



Es conveniente estudiar este listado. En él se muestra la relación entre una variable, su dirección, un apuntador y la referencia a un apuntador.

### DEBE

### NO DEBE

**DEBE** Entender lo que son los apuntadores y la manera en que funcionan. El dominio de C requiere el dominio de los apuntadores.

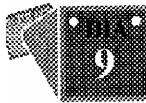
**NO DEBE** Usar apuntadores sin inicializar. Si lo hace, los resultados pueden ser desastrosos.

## Los apuntadores y los tipos de variables

La discusión anterior ignora el hecho de que tipos diferentes de variables ocupan cantidades diferentes de memoria. En la mayoría de las PC un int ocupa dos bytes, un float ocupa cuatro bytes y así sucesivamente. Cada byte individual de memoria tiene su propia dirección, por lo que una variable que ocupa varios bytes, de hecho ocupa varias direcciones.

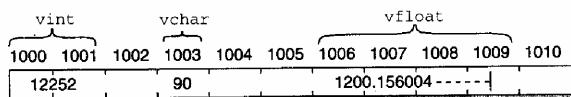
Entonces, ¿cómo manejan los apuntadores las direcciones de las variables de varios bytes? Esta es la manera en que funcionan: la dirección de una variable es, de hecho, la dirección del byte más bajo que ocupa. Esto puede ilustrarse con un ejemplo que declara e inicializa tres variables.

```
int vint = 12252;  
char vchar = 90;  
float vfloat = 1200.156004;
```



## Apuntadores

Estas variables son guardadas en memoria como lo muestra la figura 9.5. La variable `int` ocupa dos bytes, la variable `char` ocupa un byte y la variable `float` ocupa cuatro bytes.

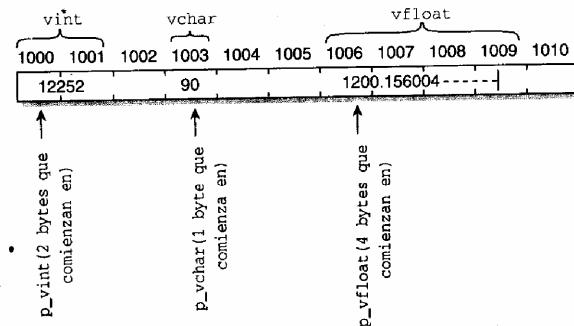


**Figura 9.5.** Los diferentes tipos de variables numéricas ocupan diferente cantidad de espacio de almacenamiento en memoria.

Ahora declare e inicialice apuntadores a estas tres variables.

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
/* aquí va código adicional */
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Cada apuntador es igual a la dirección del primer byte de la variable a la que apunta. Por lo tanto, `p_vint` es igual a 1000, `p_vchar` es igual a 1003, y `p_vfloat` es igual a 1006. Sin embargo, recuerde que cada apuntador fue declarado para que apuntara a determinado tipo de variable. El compilador “sabe” que un apuntador a tipo `int` apunta al primero de los dos bytes, un apuntador de tipo `float` apunta al primero de los cuatro bytes, y así sucesivamente. Esto es diagramado en la figura 9.6.

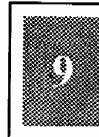


**Figura 9.6.** El compilador “sabe” el tamaño de la variable a la que apunta el apuntador.



Las figuras 9.5 y 9.6 muestran algunas posiciones de almacenamiento de memoria vacías entre las tres variables. Esto es para dar claridad visual. En la práctica real, el compilador de C guarda las tres variables en posiciones de memoria adyacentes sin ningún byte sin usar entre ellas.

## Los apuntadores y los arreglos



Los apuntadores pueden ser útiles cuando se está trabajando con variables simples, pero son más útiles con los arreglos. Hay una relación especial en C entre los apuntadores y los arreglos. De hecho, cuando se usa la notación de subíndices de arreglos, que se aprendió en el Día 8, "Arreglos numéricos", en realidad se están usando apuntadores sin saberlo. Los siguientes párrafos explican cómo funciona esto.

### El nombre del arreglo como un apuntador

Un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Por lo tanto, si se ha declarado un arreglo `datos []`, `datos` es la dirección del primer elemento del arreglo.

Tal vez esté pensando "espere un momento, ¿qué no se necesita el operador de *dirección de* para obtener una dirección?". Sí, también se puede usar la expresión `&datos[0]` para obtener la dirección del primer elemento del arreglo. En el C, la relación `(datos == &datos[0])` es cierta.

Ya ha visto que el nombre de un arreglo es un apuntador al arreglo. Recuerde que esto es una *constante de apuntador*; no puede ser cambiada y permanece fija por toda la duración de la ejecución del programa. Esto tiene sentido, si se cambiara su valor apuntaría a cualquier otro lado y no al arreglo (que permanece en una posición fija en memoria).

Sin embargo, se puede declarar una variable de apuntador e inicializarla para que apunte hacia el arreglo. Por ejemplo, el código

```
int arreglo[100], *p_arreglo;  
/* aquí va código adicional */  
p_arreglo = arreglo;
```

inicializa la variable de apuntador `p_arreglo` con la dirección del primer elemento de `arreglo[]`. Debido a que `p_arreglo` es una variable de apuntador, puede ser modificada para que apunte a cualquier otro lugar. A diferencia de `arreglo`, `p_arreglo` no está atada para que apunte al primer elemento de `arreglo[]`. Podría, por ejemplo, ser apuntada a otro elemento de `arreglo[]`. ¿Cómo se haría eso? Primero, necesitamos ver la manera en que son guardados los elementos del arreglo en la memoria.



### Almacenamiento de elementos de arreglo

Como debe recordar de lo que se dijo el Día 8, "Arreglos numéricos", los elementos de un arreglo son guardados en posiciones secuenciales de memoria, estando el primer elemento en la dirección más baja. Los siguientes elementos del arreglo (aquellos que tienen un índice mayor que 0) son guardados en direcciones más altas. Qué tan altas, depende del tipo de datos del arreglo (char, int, float, etc.).

Tomemos un arreglo de tipo int. Como se aprendió en el Día 3, "Variables y constantes numéricas", una simple variable int puede ocupar dos bytes de memoria. Por lo tanto, cada elemento del arreglo está ubicado dos bytes por encima del elemento anterior, y la dirección de cada elemento del arreglo es mayor en dos que la dirección del elemento anterior. Por otro lado, un tipo float puede ocupar cuatro bytes. En un arreglo de tipo float cada elemento del arreglo está ubicado cuatro bytes por encima del elemento anterior, y la dirección de cada elemento es mayor en cuatro que la dirección del elemento anterior.

La figura 9.7 ilustra la relación entre el almacenamiento del arreglo y las direcciones para un arreglo int de seis elementos y un arreglo float de tres elementos.

```
int x[6];
1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011
x[0] | x[1] | x[2] | x[3] | x[4] | x[5]

float gastos[3];
1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261
gastos[0] | gastos[1] | gastos[2]
```

Figura 9.7. Almacenamiento de arreglos para diferentes tipos de arreglos.

Viendo la figura 9.7 se debe ser capaz de ver el porqué las siguientes relaciones son ciertas:

- 1:  $x == 1000$
- 2:  $\&x[0] == 1000$
- 3:  $\&x[1] == 1002$
- 4:  $gastos == 1250$
- 5:  $\&gastos[0] == 1250$
- 6:  $\&gastos[1] == 1254$

$x$  sin los corchetes de arreglo es la dirección del primer elemento. En la figura 9.7 se puede ver que  $x[0]$  tiene la dirección 1000. La línea 2 también muestra esto. Puede leerse como que la dirección del primer elemento del arreglo  $x$  es igual a 1000. La línea 3 muestra que la dirección del segundo elemento (con subíndice 1 del arreglo) es 1002. Nuevamente la figura puede confirmar esto. Las líneas 4, 5 y 6 son virtualmente idénticas a las 1, 2 y 3,



respectivamente. Varían en la diferencia que hay entre las direcciones de los dos elementos de arreglo. En el arreglo `x`, tipo `int`, la diferencia es de dos bytes, y en el arreglo `gastos`, tipo `float`, la diferencia es de cuatro bytes.

¿Cómo se acceden estos elementos de arreglos sucesivos usando un apuntador? Puede ver en estos ejemplos que un apuntador debe ser aumentado en 2 para accesar los elementos sucesivos de un arreglo tipo `int`, y en 4 para accesar los elementos sucesivos de un arreglo tipo `float`. Se puede generalizar y decir que para accesar elementos sucesivos de un arreglo de un tipo de datos particular, el apuntador debe ser incrementado en `sizeof(tipo_de_dato)`. Recuerde del Día 3, "Variables y constantes numéricas", que el operador `sizeof()` regresa el tamaño en bytes de un tipo de datos del C.

El programa del listado 9.2 ilustra la relación entre las direcciones y los elementos de arreglos de diferente tipo, declarando arreglos de tipo `int`, `float` y `double`, y desplegando las direcciones de sus elementos sucesivos.



### Listado 9.2. Despliegado de las direcciones de elementos sucesivos de arreglo.

```
1: /* Demuestra la relación entre direcciones y */
2: /* elementos de arreglos de diferentes tipos de datos */
3:
4: #include <stdio.h>
5:
6: /* Declara tres arreglos y una variable de contador */
7:
8: int i[10], x;
9: float f[10];
10: double d[10];
11:
12: main()
13: {
14:     /* Imprime el encabezado de la tabla */
15:
16:     printf("\t\tInteger\t\tFloat\t\tDouble");
17:
18:     printf("\n=====");
19:     printf("=====");
20:
21:     /* Imprime las direcciones de cada elemento de arreglo */
22:
23:     for (x = 0; x < 10; x++)
24:         printf("\nElement %d:\t%d\t%d\t%d", x, &i[x],
25:                &f[x], &d[x]);
26:
27:     printf("\n=====");
28:     printf("=====");
29: }
```



## Apuntadores



Aquí se muestra la salida del programa. Las direcciones exactas que despliega su sistema pueden ser diferentes, pero las relaciones son las mismas: 2 bytes entre elementos int, 4 bytes entre elementos float y 8 bytes entre elementos double.  
(Nota: Algunas máquinas usan diferentes tamaños para los tipos de variables. Si la máquina es diferente, la siguiente salida puede tener diferentes tamaños de intervalo, sin embargo, los intervalos serán consistentes.)

	Integer	Float	Double
=====			
Element 0:	1392	1414	1454
Element 1:	1394	1418	1462
Element 2:	1396	1422	1470
Element 3:	1398	1426	1478
Element 4:	1400	1430	1486
Element 5:	1402	1434	1494
Element 6:	1404	1438	1502
Element 7:	1406	1442	1510
Element 8:	1408	1446	1518
Element 9:	1410	1450	1526
=====			



Este listado aprovecha los caracteres de escape que se aprendieron en el Día 7, "Entrada/salida básica". Las llamadas a `printf()` en las líneas 16 y 24 usan el carácter de escape tab (\t), para ayudarse en el formateo de la tabla alineando las columnas.

Viendo más de cerca el listado, se puede observar que son creados tres arreglos en las líneas 8, 9 y 10. La línea 8 declara al arreglo `i` de tipo int, la línea 9 declara al arreglo `f` de tipo float y la línea 10 declara al arreglo `d` de tipo double. La línea 16 imprime los encabezados de columna para la tabla que será desplegada. Las líneas 18 y 19, junto con las líneas 27 y 28, imprimen líneas de guiones en la parte superior e inferior de la tabla de datos. Este es un toque agradable para un informe. Las líneas 23, 24 y 25, son un ciclo `for` que imprime cada uno de los renglones de la tabla. El número del elemento, `x`, es escrito primero. Esto es seguido por la dirección del elemento en cada uno de los tres arreglos.

## Aritmética de apuntadores

Se tiene un apuntador al primer elemento del arreglo. El apuntador debe ser incrementado por una cantidad igual al tamaño del tipo de dato guardado en el arreglo. ¿Cómo se acceden los elementos del arreglo usando notación de apuntadores? Con *aritmética de apuntadores*.

Tal vez piense "precisamente lo que no necesito, ¡otro tipo de aritmética que tengo que aprender!" No se preocupe. La aritmética de apuntadores es simple, y facilita el uso de apuntadores en los programas. Sólo se tiene que tratar con dos operaciones de apuntadores: incremento y decremento.



## Incremento de apuntadores

Cuando se incrementa un apuntador se está incrementando su valor. Por ejemplo, cuando se incrementa un apuntador en 1, la aritmética de apuntadores incrementa automáticamente el valor del apuntador para que apunte al siguiente elemento del arreglo. Dicho de otra forma, el C sabe el tipo de dato al que apunta el apuntador (a partir de la declaración del apuntador) e incrementa la dirección guardada en el apuntador en el tamaño del tipo de dato.

Supongamos que `p_a_int` es una variable de apuntador a algún elemento de un arreglo `int`. Si se ejecuta el enunciado

```
p_a_int++;
```

el valor de `p_a_int` es aumentado en el tamaño de tipo `int` (que es por lo general, de 2 bytes) y `p_a_int` apunta ahora al siguiente elemento del arreglo. De manera similar, si `p_a_float` apunta a un elemento de un arreglo de tipo `float`, luego

```
p_a_float++;
```

incrementa el valor de `p_a_float` en el tamaño del tipo `float` (que por lo general es de 4 bytes).

Lo mismo se mantiene para incrementos mayores de 1. Si se suma el valor `n` a un apuntador, el C incrementa el apuntador en `n` elementos de arreglo del tipo de datos asociado. Por lo tanto,

```
p_a_int += 4;
```

incrementa el valor guardado en `p_a_int` en ocho, para que de esta forma apunte a cuatro elementos de arreglo adelante. De manera similar

```
p_a_float += 10;
```

incrementa el valor guardado en `p_a_float` en 40, para que apunte a 10 elementos de arreglo adelante.

## Decremento de apuntadores

Se aplica el mismo concepto para el decremento de un apuntador como para el incremento. El decremento de un apuntador es, de hecho, un caso de incremento sumando un valor *negativo*. Si se decremente un apuntador con los operadores `--` o `-=`, la aritmética de apuntadores se ajusta automáticamente al tamaño de los elementos del arreglo.

El listado 9.3 presenta un ejemplo sobre la manera en que puede ser usada la aritmética de apuntadores para accesar elementos de un arreglo. Incrementando apuntadores, el programa puede ir paso a paso por todos los elementos del arreglo en forma eficiente.



## Apuntadores



### Listado 9.3. Uso de la aritmética de apuntadores y de la notación de apuntadores para accesar elementos de arreglo.

```
1: /* Demuestra el uso de aritmética de apuntadores para accesar */
2: /* elementos de arreglo con notación de apuntador */
3:
4: #include <stdio.h>
5: #define MAX 10
6:
7: /* Declara e inicializa un arreglo de enteros */
8:
9: int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9 };
10:
11: /* Declara un apuntador a int y una variable int */
12:
13: int *i_ptr, count;
14:
15: /* Declara e inicializa un arreglo de flotantes */
16:
17: float f_array[MAX] = { .0, .1, .2, .3, .4, .5, .6, .7, .8, .9 };
18:
19: /* Declara un apuntador a float */
20:
21: float *f_ptr;
22:
23: main()
24: {
25:     /* Inicializa los apuntadores */
26:
27:     i_ptr = i_array;
28:     f_ptr = f_array;
29:
30:     /* Imprime los elementos del arreglo */
31:
32:     for (count = 0; count < MAX; count++)
33:         printf("\n%d\t%f", *i_ptr++, *f_ptr++);
34: }
```



0	0.000000
1	0.100000
2	0.200000
3	0.300000
4	0.400000
5	0.500000
6	0.600000
7	0.700000
8	0.800000
9	0.900000



En este programa, una constante definida llamada MAX es puesta a 10 en la línea 5 y es usada por todo el listado. En la línea 9, MAX es usada para poner la cantidad de elementos en un arreglo de ints llamado i\_array. Los elementos en este arreglo son inicializados al mismo tiempo que es declarado el arreglo. La línea 13 declara dos variables int adicionales. La primera es un apuntador llamado i\_ptr. Se sabe que es un apuntador, debido a que se usa un operador de indirección (\*). La otra variable es una variable simple de tipo int llamada count. En la línea 17 es definido e inicializado un segundo arreglo. Este arreglo es de tipo float, contiene MAX valores y es inicializado con valores float. La línea 21 declara un apuntador a un float llamado f\_ptr.

La función main () se encuentra en las líneas 23 a 34. El programa asigna la dirección inicial de los dos arreglos a los apuntadores de sus respectivos tipos en las líneas 27 y 28. Recuerde que un nombre de arreglo sin el subíndice es lo mismo que la dirección del comienzo del arreglo. Un enunciado for, en las líneas 32 y 33, usa la variable int count para contar de 0 al valor de MAX. Para cada cuenta, la línea 33 hace referencia a los dos apuntadores e imprime sus valores con una llamada a la función printf (). El operador de incremento luego incrementa cada uno de los apuntadores para que apunte al siguiente elemento del arreglo, antes de continuar con la siguiente iteración del ciclo for.

Tal vez piense que el programa del listado 9.3 podría también haber usado notación de subíndices para los arreglos y evitarnos el manejo de apuntadores. Esto es cierto, y en tareas simples de programación como ésta, el uso de la notación de apuntadores no ofrece ninguna ventaja. Sin embargo, cuando comience a escribir programas más complejos encontrará útil el uso de apuntadores.

Por favor, recuerde que no puede ejecutar operaciones de incremento y decremento sobre constantes de apuntador. (Un nombre de arreglo sin corchetes es una constante de apuntador.) También recuerde que cuando se están manejando apuntadores a elementos de arreglo, el compilador C no lleva cuenta del comienzo y fin del arreglo. Si no se tiene cuidado, se puede incrementar o decrementar el apuntador de tal forma que apunte a cualquier lugar en memoria antes o después del arreglo. Hay algo guardado ahí, pero no es un elemento de arreglo. Se debe llevar cuenta de los apuntadores y hacia dónde están apuntando.

## Otras manipulaciones de apuntadores

La única otra operación de aritmética de apuntadores es llamada *diferencia*, que se refiere a la resta de dos apuntadores. Si se tienen dos apuntadores hacia diferentes elementos del mismo arreglo, se les puede restar y encontrar qué tan distantes se encuentran. Nuevamente, la aritmética de apuntadores escala automáticamente la respuesta para que haga referencia a elementos de arreglo. Por lo tanto, si ptr1 y ptr2 apuntan a elementos de un arreglo (de cualquier tipo), la expresión

$ptr1 - ptr2$

le dice qué tan distantes se encuentran los elementos.



## Apuntadores

---

Las comparaciones de apuntadores son válidas solamente entre apuntadores que apunten al mismo arreglo. Bajo estas circunstancias, los operadores relacionales ==, !=, >, <, >= y <= funcionan adecuadamente. Los elementos inferiores del arreglo (esto es, los que tienen un subíndice menor) siempre tienen direcciones menores que los elementos superiores del arreglo. Por lo tanto, si ptr1 y ptr2 apuntan a elementos del mismo arreglo, la comparación

ptr1 < ptr2

es cierta si ptr1 apunta a un miembro anterior del arreglo que al que apunta ptr2.

Con esto se tratan todas las operaciones permitidas de apuntadores. Muchas operaciones aritméticas que pueden ser ejecutadas con variables regulares, como la multiplicación o la división, no tienen sentido con los apuntadores. El compilador C no las permite. Por ejemplo, si ptr es un apuntador, el enunciado

ptr \*= 2;

genera un mensaje de error. Se pueden hacer un total de seis operaciones con un apuntador, y todas han sido tratadas en este capítulo:

*Asignación.* Se puede asignar un valor a un apuntador. El valor debe ser una dirección, obtenida con el operador dirección de (&) o a partir de una constante de apuntador (un nombre de arreglo).

*Indirección.* El operador de indirección (\*) le da el valor guardado en la posición apuntada.

*dirección de.* Se puede usar el operador de dirección de para encontrar la dirección de un apuntador, por lo que se pueden tener apuntadores hacia apuntadores. Esto es un tema avanzado y se trata en el Día 15, “Más sobre apuntadores”.

*Incremento.*

*Diferencia.*

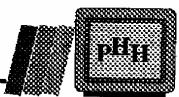
*Comparación.* Sólo son válidas con dos apuntadores que apunten al mismo arreglo.

## Precauciones con los apuntadores

Cuando se está escribiendo un programa que usa apuntadores se debe evitar un error serio: el uso de un apuntador sin inicializar al lado izquierdo de un enunciado de asignación. Por ejemplo, el enunciado

int \*ptr;

declara a un apuntador de tipo int. Este apuntador todavía no está inicializado, por lo que no apunta a ningún lado. Para ser más exactos, no apunta a ningún lado *conocido*. Un



apuntador sin inicializar tiene algún valor, pero simplemente uno no sabe cuál es. En muchos casos es cero. Por lo tanto, si se usa un apuntador sin inicializar en un enunciado de asignación, esto es lo que pasa:

\*ptr = 12;

El valor 12 es asignado a donde esté apuntando `ptr`. Esta dirección puede ser cualquiera en memoria, donde está guardado el sistema operativo o en algún lugar del código de programa. El 12 que es guardado ahí puede sobreescribir alguna información importante, y el resultado puede ir desde un error extraño del programa hasta una caída completa del sistema.

El lado izquierdo de un enunciado de asignación es el lugar más peligroso para usar un apuntador no inicializado. Otros errores, aunque menos serios, pueden también ser resultado del uso de un apuntador sin inicializar en cualquier lado del programa, por lo que asegúrese de que los apuntadores de su programa estén correctamente inicializados antes de usarlos. Se debe hacer esto por uno mismo. ¡El compilador no puede verlo por uno!

DEBE	NO DEBE
	<b>NO DEBE</b> Tratar de hacer operaciones matemáticas, como la división, multiplicación y módulo, sobre apuntadores. La suma (incremento) y la resta (diferencia) de apuntadores es aceptable.
	<b>NO DEBE</b> Olvidar que restar o sumar a un apuntador incrementa el apuntador basándose en el tamaño del tipo de dato al que apunta, y no por 1 o el número que es sumado (a menos que sea un apuntador a un carácter de un byte).
	<b>DEBE</b> Comprender el tipo de variables que tiene la PC. Como podrá ver, se necesita saber el tamaño de las variables cuando se trabaja con apuntadores y memoria.
	<b>NO DEBE</b> Tratar de incrementar o decrementar una variable de arreglo. Asigne un apuntador a la dirección inicial del arreglo e increméntela (véase el listado 9.3).

## Notación de subíndices de arreglo y apuntadores

Un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Por lo tanto, se puede accesar el primer elemento del arreglo usando el operador de indirección. Si `arreglo[]` es un arreglo declarado, la expresión `*arreglo` es el primer elemento del



## Apuntadores

arreglo, `* (arreglo + 1)` es el segundo elemento del arreglo, y así sucesivamente. Si se generaliza para el arreglo completo, las siguientes relaciones son ciertas:

```
*(arreglo) == arreglo[0]
*(arreglo + 1) == arreglo[1]
*(arreglo + 2) == arreglo[2]
...
*(arreglo + n) == arreglo[n]
```

Esto ilustra la equivalencia de la notación de subíndices de arreglo y de apuntadores de arreglo. Se puede usar cualquiera de ellas en el programa, ya que el compilador las ve como dos diferentes maneras de accesar los datos del arreglo usando apuntadores.

## Paso de arreglos a funciones

Este capítulo ya ha tratado la relación especial que existe en C entre apuntadores y arreglos. Esta relación viene al caso cuando se necesita pasar un arreglo como argumento a una función. La única manera en que se puede pasar un arreglo a una función es por medio de un apuntador.

Como aprendió en el Día 5, “Funciones: lo básico”, un argumento es un valor que el programa llamador pasa a una función. Puede ser un `int`, un `float` o cualquier otro tipo de dato simple, pero tiene que ser un solo valor numérico. Puede ser un solo elemento de arreglo, pero no puede ser un arreglo completo.

¿Qué pasa si necesita pasar un arreglo completo a una función? Bueno, se puede tener un apuntador hacia el arreglo, y ese apuntador es un solo valor numérico (la dirección del primer elemento del arreglo). Si se pasa ese valor a la función, la función “sabe” la dirección del arreglo, y puede accesar los elementos del arreglo usando notación de apuntador.

Sin embargo, considere otro problema. Si se escribe una función que toma a un arreglo como argumento, se quiere que la función sea capaz de manejar arreglos de diferentes tamaños. Por ejemplo, se podría escribir una función que encuentre al elemento más grande en un arreglo de enteros. La función no sería de mucha utilidad si estuviera limitada a tratar con arreglos de tamaño fijo (cantidad de elementos).

¿Cómo hace la función para saber el tamaño del arreglo, cuya dirección se pasa? Recuerde, el valor pasado a una función es un apuntador al primer elemento del arreglo. Puede ser el primero de 10 elementos o el primero de 10,000. Hay dos métodos para permitir que una función “sepa” el tamaño del arreglo.

Se puede indentificar al último elemento del arreglo, guardando en él algún valor especial. Conforme la función procesa al arreglo revisa cada elemento para ver si contiene el valor. Si lo contiene, se ha llegado al final del arreglo. La desventaja de este método es que lo fuerza

a reservar algún valor como el indicador de fin de arreglo, reduciendo la flexibilidad que se tiene para guardar datos reales en el arreglo.

El otro método es más flexible y directo: pase a la función el tamaño del arreglo como argumento. Este puede ser un simple argumento de tipo `int`. Por lo tanto, a la función se le pasan dos argumentos: un apuntador al primer elemento del arreglo, y un entero especificando la cantidad de elementos del arreglo. Este segundo método es usado en este libro.

El listado 9.4 acepta una lista de valores del usuario y los guarda en un arreglo. Luego llama a una función, `largest()`, pasándole el arreglo (tanto el apuntador como el tamaño). La función encuentra el valor más grande en el arreglo y lo regresa al programa llamador.

#### **Listado 9.4. Demostración del paso de un arreglo a una función.**

```

1: /* Paso de un arreglo a una función */
2:
3: #include <stdio.h>
4:
5: #define MAX 10
6:
7: int array[MAX], count;
8:
9: int largest(int x[], int y);
10:
11: main()
12: {
13:     /* Recibe del teclado MAX valores */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:     }
20:
21:     /* Llama a la función y despliega el valor de retorno */
22:
23:     printf("\n\nLargest value = %d", largest(array, MAX));
24: }
25:
26: /* La función largest() regresa el valor más grande */
27: /* de un arreglo entero */
28:
29: int largest(int x[], int y)
30: {
31:     int count, biggest = -12000;
32:
33:     for ( count = 0; count < y; count++) .

```

## Apuntadores

### Listado 9.4. continuación

```
34:      {
35:          if (x[count] > biggest)
36:              biggest = x[count];
37:      }
38:
39:      return biggest;
40: }
```

**Añadido:** Hay un prototipo de función en la línea 9 y un encabezado de función en la línea 29 que son idénticos. Ellos dicen

```
int largest(int x[], int y)
```

La mayor parte de esta línea debe serle familiar: `largest()` es una función que regresa un `int` al programa llamador, y su segundo argumento es un `int`, representado por el parámetro `y`. La única cosa nueva es el primer parámetro, `int x[]`, que indica que el primer argumento es un apuntador a tipo `int`, representado por el parámetro `x`. También se podría escribir la declaración de función y el encabezado de la manera siguiente:

```
int largest(int *x, int y);
```

Esto es equivalente a la primera forma, ya que tanto `int x[]` como `int *x` significan “apuntador a `int`”. La primera forma puede ser preferible, ya que le recuerda que el parámetro representa un apuntador a un arreglo. Por supuesto que el apuntador no sabe si apunta a un arreglo, pero la función lo usa de esa manera.

Ahora veamos la función `largest()`. Cuando es llamada por primera vez, el parámetro `x` guarda el valor del primer argumento, y es, por lo tanto, un apuntador al primer elemento del arreglo. Se puede usar `*x` en cualquier lugar en que pueda ser usado un apuntador de arreglo. En `largest()` los elementos de arreglo son accesados usando notación de subíndices en las líneas 35 y 36. También se podría haber usado notación de apuntadores, reescribiendo el ciclo `if` para que dijera

```
for ( count = 0; count < y; count++)
{
    if (*x+count) > biggest)
        biggest = *(x+count);
}
```

El listado 9.5 muestra la otra manera de pasar arreglos a funciones.

**Listado 9.5. Una forma alterna para pasar un arreglo a una función.**

```
1: /* Paso de un arreglo a una función. Método alterno. */
2:
3: #include <stdio.h>
4:
5: #define MAX 10
6:
7: int array[MAX+1], count;
8:
9: int largest(int x[]);
10:
11: main()
12: {
13:     /* Recibe del teclado MAX valores. */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:
20:         if (array[count] == 0)
21:             count = MAX;                  /* Terminará el ciclo for */
22:     }
23:     array[MAX] = 0;
24:
25:     /* Llama a la función y despliega el valor de retorno. */
26:
27:     printf("\n\nLargest value = %d", largest(array));
28: }
29:
30: /* La función largest() regresa el valor más grande */
31: /* de un arreglo entero */
32:
33: int largest(int x[])
34: {
35:     int count, biggest = -12000;
36:
37:     for (count = 0; x[count] != 0; count++)
38:     {
39:         if (x[count] > biggest)
40:             biggest = x[count];
41:     }
42:
43:     return biggest;
44: }
```



## Apuntadores



Este programa usa una función `largest()` que tiene la misma funcionalidad que el listado anterior. La diferencia es que solamente es necesario el nombre del arreglo. El ciclo `for` de la línea 37 continúa buscando el valor más grande hasta que encuentra un 0, ya que en ese momento sabe que ha terminado.

Viendo las primeras partes del listado se pueden ver las diferencias entre el listado 9.4 y el 9.5. En primer lugar, en la línea 7 se necesita añadir un elemento adicional al arreglo, para guardar el valor que indica el final. En las líneas 20 y 21 es añadido un enunciado `if`, para ver si el usuario ha tecleado un cero, señalando con esto que ya ha terminado de dar valores. Si se teclea un cero, `count` es puesto a su valor máximo para que pueda salir del ciclo `for` limpiamente. La línea 23 asegura que el último elemento es un cero, en caso de que el usuario haya dado la máxima cantidad de valores (`MAX`).

Añadiendo los comandos adicionales, cuando se teclean los datos se puede hacer que la función `largest()` funcione con cualquier tamaño de arreglo, mas, sin embargo, hay un pero. ¿Qué pasa si se olvida de poner un cero al final del arreglo? Entonces `largest()` continúa más allá del final del arreglo, comparando valores en memoria hasta que encuentra un cero.

Como puede ver, el pasar un arreglo a una función no tiene gran dificultad. Simplemente se pasa un apuntador al primer elemento del arreglo. En la mayoría de las situaciones, también es necesario pasar la cantidad de elementos del arreglo. En la función el valor del apuntador puede ser usado para accesar los elementos del arreglo, ya sea con notación de subíndice o de apuntador.

Recuerde del Día 5, “Funciones: lo básico”, que cuando es pasada una variable simple a una función sólo se pasa una copia del valor de la variable. La función puede usar el valor pero no puede cambiar a la variable original, debido a que no tiene acceso a la variable misma. Cuando se pasa un arreglo a una función las cosas son diferentes. A la función se le pasa la dirección del arreglo, y no simplemente una copia de los valores del arreglo. El código de la función está trabajando con los elementos actuales del arreglo y puede modificar los valores guardados en el arreglo.

## Resumen

Este capítulo le presentó a los apuntadores, que son una parte central de la programación en C. Un apuntador es una variable que guarda la dirección de otra variable. Se dice que un apuntador “apunta” a la variable cuya dirección guarda. Los dos operadores necesarios con los apuntadores son el operador de *dirección de* (&) y el operador de *indirección* (\*). Cuando es puesto antes de un nombre de variable, el operador de *dirección de* regresa la dirección de la variable. Cuando es puesto antes de un nombre de apuntador, el operador de *indirección* regresa la dirección de la variable apuntada.

Los apuntadores y los arreglos tienen una relación especial. Un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Las características especiales de



la aritmética de apuntadores facilitan el acceso a los elementos del arreglo usando apuntadores. La notación de subíndices de arreglo es, de hecho, una forma especial de notación de apuntadores.

También aprendió la manera de pasar arreglos como argumentos a funciones, pasando un apuntador hacia el arreglo. Una vez que la función "sabe" la dirección del arreglo y su longitud, puede accesar los elementos del arreglo usando notación de apuntadores o notación de subíndices.



## Preguntas y respuestas

1. ¿Por qué son tan importantes los apuntadores en C?

Los apuntadores le dan mayor control sobre la computadora y los datos. Cuando se usan con funciones, los apuntadores le permiten cambiar el valor de variables que fueron pasadas sin tomar en cuenta dónde han sido originadas. En el Día 15, "Más sobre apuntadores", aprenderá usos adicionales de los apuntadores.

2. ¿Cómo sabe el compilador la diferencia entre \* para multiplicación, \* para referencia y \* para la declaración de un apuntador?

El compilador interpreta los usos diferentes del asterisco, basándose en el contexto donde es usado. Si el enunciado que está evaluando comienza con un tipo de variable, puede suponerse que el asterisco es para declarar un apuntador. Si el asterisco es usado con una variable que ha sido declarada como apuntador, pero no en una declaración de variable, el asterisco es interpretado como referencia. Si es usado en una expresión matemática pero no con una variable de apuntador, se puede suponer que el asterisco corresponde al operador de multiplicación.

3. ¿Qué pasa si uso el operador de *dirección de* sobre un apuntador?

Se obtiene la dirección de la variable de apuntador. Recuerde que un apuntador es simplemente otra variable, que guarda la dirección de la variable a la que apunta.

4. ¿Son guardadas las variables siempre en la misma dirección?

No. Cada vez que se ejecuta un programa sus variables pueden ser guardadas en diferentes direcciones. Nunca se debe asignar un valor constante de dirección a un apuntador.

## Taller

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



## Apuntadores

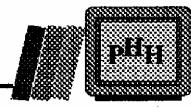
---

### Cuestionario

1. ¿Qué operador se usa para determinar la dirección de una variable?
2. ¿Qué operador se usa para determinar el valor de la dirección apuntada por un apuntador?
3. ¿Qué es un apuntador?
4. ¿Qué es indirección?
5. ¿Cómo son guardados en memoria los elementos de un arreglo?
6. Muestre dos maneras para obtener la dirección del primer elemento del arreglo datos [ ].
7. Si se le pasa un arreglo a una función, ¿cuáles son las dos maneras de saber dónde se encuentra el fin del arreglo?
8. ¿Cuáles son las seis operaciones tratadas en este capítulo que pueden realizarse con un apuntador?
9. Supongamos que tiene dos apuntadores. Si el primero apunta al tercer elemento de un arreglo de int y el segundo apunta al cuarto elemento, ¿qué valor se obtiene si se resta el primer apuntador del segundo?
10. Supongamos que el arreglo de la pregunta anterior es de valores float. ¿Qué valor se obtiene si se restan los dos apuntadores?

### Ejercicios

1. Haga una declaración de un apuntador a una variable tipo char. Llame al apuntador char\_ptr.
2. Si se tiene una variable tipo int llamada costo, ¿cómo declararía e inicializaría un apuntador llamado p\_costo que apuntara a esa variable?
3. Continuando con el ejercicio 2, ¿cómo asignaría el valor 100 a la variable costo usando acceso directo y acceso indirecto?
4. Continuando con el ejercicio 3, ¿cómo imprimiría el valor del apuntador y el valor al que está apuntando?
5. Muestre la manera de asignar la dirección de un valor float llamado radio a un apuntador.
6. Muestre dos maneras de asignar el valor de 100 al tercer elemento de datos [ ].



7. Escriba una función, llamada `totarreglo()`, que acepte dos arreglos como argumentos, obtenga el total de todos los valores de ambos arreglos y regrese el total al programa llamador.
8. Use la función creada en el ejercicio siete en un programa simple.
9. Escriba una función, llamada `suma_arreglo()`, que acepte dos arreglos que sean del mismo tamaño. La función debe sumar los elementos correspondientes de los arreglos y dejar los valores en un tercer arreglo.
10. Modifique la función del ejercicio 9 para que regrese un apuntador al arreglo que contiene los totales. Ponga esta función en un programa que también imprima los valores de los tres arreglos.







## Carácteres y cadenas

Un *carácter* es una simple letra, número, signo de puntuación o cualquier otro símbolo. Una *cadena* es cualquier secuencia de caracteres. Las cadenas se usan para guardar datos de texto, que constan de letras, números, signos de puntuación y otros símbolos. No hay duda de que los caracteres y las cadenas son muy útiles en muchas aplicaciones de programación. Hoy aprenderá

- Cómo usar el tipo de dato `char` del C para guardar caracteres solos.
- Cómo crear arreglos de tipo `char` para guardar cadenas de varios caracteres.
- Cómo inicializar caracteres y cadenas.
- Cómo usar apuntadores con cadenas.
- Cómo imprimir y capturar caracteres y cadenas.

## El tipo de dato `char`

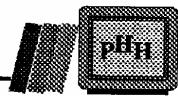
El C usa el tipo de dato `char` para guardar caracteres. Ya vio en el Día 3, “Variables y constantes numéricas”, que `char` es uno de los tipos de datos numéricos enteros del C. Si `char` es un tipo numérico, ¿cómo puede ser usado para guardar caracteres?

La respuesta se encuentra en la manera en que el C guarda los caracteres. La memoria de la computadora guarda todos los datos en forma numérica. No hay una manera directa de guardar caracteres. Sin embargo, existe un código numérico para cada carácter. Este se llama el *código ASCII* o el juego de caracteres ASCII. (ASCII es la abreviatura de American Standard Code for Information Interchange.) El código asigna valores, que van de 0 a 255, para las letras mayúsculas y minúsculas, los dígitos numéricos, los signos de puntuación y otros símbolos. El juego de caracteres ASCII se lista en el apéndice A, “Tabla de caracteres ASCII”.

Por ejemplo, 97 es el código ASCII para la letra `a`. Cuando se guarda el carácter `a` en una variable tipo `char`, en realidad se está guardando el valor 97. Como el rango numérico permitido para el tipo `char` corresponde al juego de caracteres ASCII estándar, `char` se presta adecuadamente para guardar caracteres.

Tal vez en este momento se sienta un poco confundido. Si el C guarda caracteres como números, ¿cómo sabe el programa cuando una variable tipo `char` es un carácter o un número? Como aprenderá más adelante, declarar una variable como tipo `char` no es suficiente, sino que se debe hacer algo más con la variable.

- Si se usa una variable `char` en alguna parte de un programa en C donde se espera un carácter, es interpretada como carácter.
- Si se usa una variable `char` en alguna parte de un programa en C donde se espera un número, es interpretada como número.



Esto le da una idea de la manera en que el C usa el tipo de datos numéricos para guardar datos de carácter. Ahora podemos pasar a los detalles.

## Uso de variables de carácter

De manera similar a las demás variables, se debe declarar a las de tipo `char` antes de usarlas, y se les puede inicializar al momento de la declaración. A continuación se presentan algunos ejemplos:

```
char a, b, c;      /* Declara tres variables char sin inicializar */
char codigo = 'x'; /* Declara la variable char llamada código           */
...                 /* y guarda ahí el carácter x                   */
codigo = '!';     /* Guarda ! en la variable llamada código       */
```

Para crear constantes literales de carácter encierre un solo carácter entre comillas simples. El compilador traduce automáticamente las constantes literales de carácter a sus códigos ASCII correspondientes, y el valor numérico del código es asignado a la variable.

Se pueden crear constantes de carácter simbólicas usando la directiva `#define` o la palabra clave `const`:

```
#define EX 'x'
char codigo = EX;      /* Hace que código sea igual a 'x' */
const char A = 'Z';
```

Ahora que ya sabe cómo declarar e inicializar variables de carácter, es tiempo de una demostración. El programa del listado 10.1 ilustra la naturaleza numérica del almacenamiento de caracteres, usando la función `printf()` que se aprendió en el Día 7, "Entrada/salida básica". La función `printf()` puede ser usada para imprimir caracteres y números. El formato `%c` le da instrucciones a `printf()` para que imprima un carácter, y el formato `%d` le da instrucciones de que imprima un entero decimal. El listado 10.1 inicializa dos variables tipo `char` e imprime cada una, primero como carácter y luego como número.



### Listado 10.1. Demostración de la naturaleza numérica de las variables tipo `char`.

```
1: /* Demuestra la naturaleza numérica de las variables char*/
2:
3: #include <stdio.h>
4:
5: /* Declara e inicializa dos variables char */
6:
7: char c1 = "a";
8: char c2 = 90;
9:
10: main()
11: {
```

**Listado 10.1. continuación**

```

12:     /* Imprime la variable c1 como un carácter y luego como un número */
13:
14:     printf("\nAs a character, variable c1 is %c", c1);
15:     printf("\nAs a number, variable c1 is %d", c1);
16:
17:     /* Hace lo mismo para la variable c2 */
18:
19:     printf("\nAs a character, variable c2 is %c", c2);
20:     printf("\nAs a number, variable c2 is %d", c2);
21: }
```



As a character, variable c1 is a  
 As a number, variable c1 is 97  
 As a character, variable c2 is Z  
 As a number, variable c2 is 90



Se aprendió en el Día 3, “Variables y constantes numéricas”, que el rango permitido para las variables de tipo char llega solamente hasta 127, aunque el código ASCII va hasta 255. Los códigos ASCII están, de hecho, divididos en dos partes. Los códigos ASCII estándares van hasta 127; este rango incluye todas las letras, números, signos de puntuación y otros símbolos del teclado. Los códigos del 128 al 255 son los códigos ASCII extendidos, y representan caracteres especiales, como letras extranjeras y símbolos gráficos (véase el apéndice A, “Tabla de caracteres ASCII”, para una lista completa). Por lo tanto, para los datos de texto estándar se pueden usar variables tipo char. Si se quieren imprimir los caracteres ASCII extendidos, se debe usar unsigned char.

El programa del listado 10.2 muestra la impresión de algunos de los caracteres ASCII extendidos.

**Listado 10.2. Impresión de caracteres ASCII extendidos.**

```

1: /* Demuestra la impresión de caracteres ASCII extendidos */
2:
3: #include <stdio.h>
4:
5: unsigned char x;    /* Debe ser unsigned para el ASCII extendido */
6:
7: main()
8: {
9:     /* Imprime caracteres ASCII extendidos del 180 al 203 */
10:
11:    for (x = 180; x < 204; x++)
12:    {
13:        printf("\nASCII code %d is character %c", x, x);
14:    }
15: }
```



Con este programa se ve que la línea 5 declara una variable de carácter sin signo, `unsigned x`. Esto nos da un rango de 0 a 255. De manera similar a otros tipos de datos numéricos, no se debe inicializar una variable `char` a un valor que se encuentra fuera del rango permitido, ya que de hacerlo así se obtendrán resultados impredecibles. En la línea 11, `x` no se inicializa fuera de rango, sino que, en vez de ello, se inicializa a 180. En el enunciado `for`, `x` se incrementa en 1 hasta que llega a 204. Cada vez que se incrementa `x`, la línea 13 imprime el valor de `x` y del valor del carácter de `x`. Recuerde que `%c` imprime el valor del carácter, o ASCII, de `x`.

### DEBE

### NO DEBE

**DEBE** Usar `%c` para imprimir el valor de carácter de un número.

**NO DEBE** Usar comillas dobles cuando inicialice una variable de carácter.

10

### DEBE

### NO DEBE

**DEBE** Usar comillas simples cuando inicialice una variable.

**NO DEBE** Tratar de poner valores de caracteres ASCII extendidos en una variable de tipo `char` con signo.

**DEBE** Estudiar la tabla ASCII del apéndice A para ver los caracteres interesantes que pueden ser impresos.

## Uso de cadenas

Las variables de tipo `char` pueden guardar un solo carácter únicamente, por lo que tienen una utilidad limitada. También se necesita una manera de guardar *cadenas*, que son una secuencia de caracteres. El nombre de una persona o una dirección son ejemplos de cadenas. Aunque no hay un tipo de datos especial para las cadenas, el C maneja este tipo de información con arreglos de caracteres.

## Arreglos de caracteres

Por ejemplo, para guardar una cadena de seis caracteres se necesita declarar un arreglo de tipo `char` con siete elementos. Los arreglos de tipo `char` son declarados de manera similar a los arreglos de otros tipos de datos. Por ejemplo, el enunciado

```
char cadena[10];
```

declara un arreglo de 10 elementos de tipo `char`. Este arreglo pudiera ser usado para guardar una cadena de nueve o menos caracteres.

Tal vez esté pensando, “espere un momento, si es un arreglo de 10 elementos, ¿por qué puede guardar solamente nueve caracteres?”. En C, una cadena está definida como una secuencia de caracteres que termina con el carácter nulo, un carácter especial representado por `\0`. Aunque es representado por dos caracteres (diagonal inversa-cero) el carácter nulo es interpretado como un solo carácter, y tiene el valor ASCII de 0. Es una de las secuencias de escape del C que han sido tratadas el Día 7, “Entrada/salida básica”.

Cuando un programa en C guarda, por ejemplo, la palabra Veracruz, de hecho, guarda los ocho caracteres `V`, `e`, `r`, `a`, `c`, `r`, `u` y `z` seguidos del carácter nulo, `\0`, haciendo un total de nueve caracteres. Por lo tanto, un arreglo de caracteres puede guardar una cadena de caracteres que sea menor en uno que el número total de elementos del arreglo.

Una variable de tipo `char` es de un byte de tamaño, por lo que la cantidad de bytes en las variables de arreglo tipo `char` es la misma que la cantidad de elementos del arreglo.

## Inicialización de arreglos de caracteres

De manera similar a otros tipos de datos del C, los arreglos de caracteres pueden ser inicializados cuando son declarados. Se puede asignar valor, elemento por elemento, a los arreglos de caracteres, como se muestra aquí:

```
char cadena[10] = { 'V', 'e', 'r', 'a', 'c', 'r', 'u', 'z', '\0' };
```

Sin embargo, es más conveniente usar una *cadena literal*, que es una secuencia de caracteres encerrada entre comillas dobles:

```
char cadena[10] = "Veracruz";
```

Cuando se usa una cadena literal en el programa, el compilador añade automáticamente el carácter nulo de terminación al final de la cadena. Si no se especifica el número del subíndice cuando se declara al arreglo, el compilador calcula el tamaño del arreglo. Por lo tanto,

```
char cadena[] = "Veracruz";
```

crea e inicializa un arreglo de nueve elementos.

Recuerde que las cadenas requieren el carácter nulo de terminación. Las funciones del C que manejan las cadenas (tratadas en el Día 17, “Manipulación de cadenas”) determinan la longitud de la cadena buscando el carácter nulo. Las funciones no tienen otra manera de reconocer el final de la cadena. Si falta el carácter nulo, el programa piensa que la cadena se extiende hasta donde aparezca en memoria el siguiente carácter nulo. A causa de este error pueden resultar molestos errores de programa.



## Cadenas y apuntadores

Ya ha visto que las cadenas son guardadas en arreglos tipo `char`, con el final de la cadena marcado por el carácter nulo (ya que probablemente la cadena no ocupa el arreglo completo). Debido a que el final de la cadena se encuentra marcado, todo lo que se necesita para definir una cadena dada es algo que apunte a su inicio. (¿Es *apuntar* la palabra correcta? ¡Claro que sí!)

Con esta pista tal vez se esté adelantando. Por lo que se dijo en el Día 9, "Apuntadores", ya sabe que el nombre de un arreglo es un apuntador al primer elemento del arreglo. Por lo tanto, para una cadena que se encuentra guardada en un arreglo sólo necesita el nombre del arreglo para poder accesarlo. De hecho, el uso del nombre del arreglo es el método estándar del C para accesar cadenas.

Para ser más precisos, el uso del nombre del arreglo para accesar cadenas es el método esperado por las funciones de biblioteca del C. La biblioteca estándar del C incluye varias funciones que manejan cadenas. (Estas funciones se tratan en el Día 17, "Manipulación de cadenas".) Para pasar una cadena a alguna de estas funciones se pasa el nombre del arreglo. Lo mismo se aplica para las funciones de desplegado de cadenas `printf()` y `puts()`, tratadas posteriormente, en este capítulo.

Tal vez se haya dado cuenta de que se usa la frase "las cadenas son guardadas en arreglos". ¿Implica esto que algunas cadenas no son guardadas en arreglos? Así es, y la siguiente sección explica cómo.

10

## Cadenas sin arreglos

En la sección anterior se dijo que una cadena está definida por el nombre del arreglo de carácter, que es un apuntador tipo `char` que apunta al inicio de la cadena, y por un carácter nulo, que indica su fin, y que el espacio que ocupa la cadena en el arreglo es banal. De hecho, para lo único que sirve el arreglo es para proporcionar espacio a la cadena.

¿Qué tal si se encontrara algún otro espacio de almacenamiento de memoria sin asignar un arreglo? Se podría entonces guardar ahí una cadena con su carácter nulo de terminación. Un apuntador al primer carácter podría servir para especificar el comienzo de la cadena, de manera similar a que si la cadena estuviera en un arreglo asignado. ¿Cómo se haría para encontrar espacio de almacenamiento de memoria? Hay dos métodos: uno asigna espacio para un texto literal cuando el programa es compilado, y el otro usa la función `malloc()` para asignar espacio mientras el programa está ejecutando (un proceso llamado *asignación dinámica*).

## Asignación de espacio para la cadena en la compilación

El comienzo de una cadena, como se dijo anteriormente, está indicado por un apuntador a una variable de tipo `char`. Sería bueno recordar la manera en que se declara este tipo de apuntador:

```
char *mensaje;
```

Este enunciado declara un apuntador a una variable de tipo `char` llamado `mensaje`. En este momento no apunta a nada, pero qué tal si cambiamos la declaración de apuntador para que diga:

```
char *mensaje = "¡Qué buen día!";
```

Cuando este enunciado ejecuta, la cadena `¡Qué buen día!` (con su carácter nulo de terminación) es guardada en algún lugar de memoria, y el apuntador `mensaje` es inicializado para que apunte al primer carácter de la cadena. No hay que preocuparse por el lugar de memoria donde se encuentra guardada la cadena, ya que esto es manejado automáticamente por el compilador. Una vez que ha sido definido, `mensaje` es un apuntador a la cadena y puede ser utilizado como tal.

La declaración/inicialización anterior es equivalente a:

```
char mensaje[] = "¡Qué buen día!";
```

y las dos notaciones `*mensaje` y `mensaje[]` son equivalentes. Ambas significan “un apuntador a”.

Este método de asignación de espacio para el almacenamiento de cadenas es conveniente cuando se sabe lo que se necesita mientras se está escribiendo el programa. ¿Qué pasaría si el programa tuviera necesidades variables de almacenamiento de cadenas, dependiendo de los datos del usuario o de otros factores que son desconocidos al momento de escribir el programa? Se usa la función `malloc()`, que le permite asignar espacio de almacenamiento “al vuelo”.

## La función `malloc()`

`malloc()` es una de las funciones de asignación de memoria del C (acrónimo de *memory allocation*). Cuando se usa `malloc()` se le pasa la cantidad de bytes de memoria que se necesita. `malloc()` encuentra y reserva un bloque de memoria del tamaño pedido y regresa la dirección del primer byte del bloque. No hay por qué preocuparse sobre qué parte de memoria se usa, ya que esto es manejado automáticamente.

La función `malloc()` regresa una dirección, y su tipo de retorno es un apuntador a tipo `void`. ¿Por qué `void`? Un apuntador a tipo `void` es compatible con todos los tipos de datos. Como la memoria asignada por `malloc()` puede ser usada para guardar cualquiera de los tipos de datos del C, es adecuado el tipo de retorno `void`.

**Sintaxis****La función malloc()**

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

malloc() asigna un bloque de memoria de la cantidad de bytes indicada en tamaño. Usando malloc() para asignar memoria en el momento en que se necesita, en vez de asignarla toda cuando el programa se inicia, se puede usar más eficientemente la memoria de la computadora. Cuando se usa malloc() se necesita incluir el archivo de encabezado STDLIB.H. Algunos compiladores tienen otros archivos de encabezado que pueden ser incluidos, mas, sin embargo, por compatibilidad, es mejor incluir el STDLIB.H.

malloc() regresa un apuntador al bloque de memoria asignado. Si malloc() no fue capaz de asignar la cantidad de memoria solicitada, regresa nulo. Cada vez que trate de asignar memoria debe revisar el valor de retorno, aunque la cantidad de memoria por asignar sea pequeña.

**Ejemplo 1**

```
#include <stdlib.h>
main()
{
    /* asigna memoria para una cadena de 100 caracteres */
    char *string;
    if ((str = (char *) malloc(100)) == NULL)
        string = (char *) malloc(100*sizeof(char));
    printf("No hay suficiente memoria para asignar buffer\n");
    exit(1);
}
printf(";La cadena fue asignada!");
return 0;
}
```

**Ejemplo 2**

```
/* asigna memoria para un arreglo de 50 enteros */
int *numbers;
numbers = (int *) malloc(50 * sizeof(int));
```

**Ejemplo 3**

```
/* asigna memoria para un arreglo de 10 valores float */
float *numbers;
numbers = (float *) malloc(10 * sizeof(float));
```

Se puede usar malloc() para asignar memoria que almacene un solo espacio tipo char. Primero declare un apuntador a tipo char:

```
char *ptr;
```

## Caracteres y cadenas

Luego, llame a `malloc()` y pase el tamaño del bloque de memoria deseado. Como el tipo `char` ocupa un solo byte, se necesita un bloque de un byte. El valor regresado por `malloc()` es asignado al apuntador.

```
ptr = malloc(1);
```

Este enunciado asigna un bloque de memoria de un byte, y asigna su dirección a `ptr`. A diferencia de las variables que son declaradas en el programa, este byte de memoria no tiene nombre. Solamente el apuntador puede hacer referencia a la variable. Por ejemplo, para guardar ahí el carácter 'x' se podría escribir

```
*ptr = 'x';
```

La asignación de espacio de almacenamiento con `malloc()` para una cadena es casi idéntica al uso de `malloc()` para asignar espacio para una sola variable de tipo `char`. La principal diferencia es que se necesita saber la cantidad de espacio por asignar, es decir, la cantidad máxima de caracteres de la cadena. Este máximo depende de las necesidades del programa. Para este ejemplo, digamos que se quiere asignar espacio para una cadena de 99 caracteres, más 1 para el carácter nulo terminal, dando un total de 100. Primero se declara un apuntador a tipo `char` y luego se llama a `malloc()`.

```
char *ptr;
ptr = malloc(100);
```

Ahora `ptr` apunta al bloque reservado de 100 bytes, que puede ser usado para el almacenamiento y manejo de la cadena. Se puede usar `ptr` de manera similar a si el programa hubiera asignado explícitamente ese espacio con la siguiente declaración de arreglo.

```
char ptr[100];
```

El uso de `malloc()` le permite al programa asignar el espacio de almacenamiento conforme se necesita, en respuesta a los requerimientos. Por supuesto que el espacio disponible no es ilimitado, ya que depende de la cantidad de memoria instalada en la computadora y de los requerimientos de almacenamiento de otros programas. Si no se tiene suficiente memoria disponible, `malloc()` regresa 0 (nulo). El programa debe revisar el valor de retorno de `malloc()`, para estar seguro de que la memoria solicitada fue asignada satisfactoriamente. Siempre se debe comparar el valor de retorno de `malloc()` contra la constante simbólica `NULL`, definida en `STDLIB.H`. El listado 10.3 ilustra el uso de `malloc()`. Cualquier programa que use a `malloc()` debe incluir con `#include <stdlib.h>`



### Listado 10.3. Uso de la función `malloc()` para asignar espacio de almacenamiento para datos de cadena.

```
1: /* Demuestra el uso de malloc() para asignar espacio de */
2: /* almacenamiento para datos de cadena */
3:
4: #include <stdio.h>
```

programa estaría usando una computadora que guarda las variables tipo `char` en un byte. Recuerde del Día 3, "Variables y constantes numéricas", que diferentes compiladores pueden usar variables de diferente tamaño. Usar el operador `sizeof()` es una manera fácil de crear código portable.

Nunca suponga que `malloc()` obtiene la memoria que se le dice que obtenga. De hecho, no se le está diciendo que obtenga memoria, sino que se le está pidiendo. La línea 16 muestra la manera fácil de revisar para ver si `malloc()` proporcionó la memoria. Si la memoria fue asignada, `ptr` apunta a ella y, en caso contrario, `ptr` es nulo. Si el programa falla al tratar de conseguir memoria, las líneas 18 y 19 despliegan un mensaje de error y terminan el programa.

La línea 29 inicializa el otro apuntador declarado en la línea 7, `p`. A él se le asigna el mismo valor de dirección que `ptr`. Un ciclo `for` usa este nuevo apuntador para poner valores en la memoria asignada. En la línea 31 se ve que `count` es inicializado a 65 e incrementado en 1 hasta que llega a 91. Para cada ciclo del enunciado `for`, el valor de `count` es asignado a la dirección apuntada por `p`. Observe que cada vez que `count` se incrementa también se incrementa la dirección a la que apunta `p`. Esto significa que, en memoria, cada valor es puesto uno tras otro.

Ya debe haberse dado cuenta de que están siendo asignados números a `count`, que es una variable tipo `char`. No olvide la discusión acerca de los caracteres ASCII y sus equivalentes numéricos. El número 65 es equivalente a `A`, `66 = B`, `67 = C`, etc. El ciclo `for` termina después de que el alfabeto es asignado a las posiciones de memoria apuntadas. La línea 36 termina los valores de carácter apuntados, poniendo un nulo en la dirección final a la que apunta `p`. Añadiendo el nulo ahora se pueden usar estos valores como una cadena. Recuerde que `ptr` todavía apunta al primer valor, `A`, por lo que si lo usa como una cadena imprime cada uno de los caracteres hasta que llega al nulo. La línea 40 usa a `puts()` para probar este punto y para mostrar el resultado de lo que se ha hecho.

## DEBE

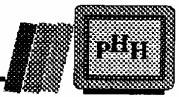
## NO DEBE

**NO DEBE** Asignar más memoria de la que necesita. No todos tienen gran cantidad de memoria, por lo que debe tratar de usarla con mesura.

**NO DEBE** Tratar de asignar una cadena nueva a un arreglo de caracteres que fue asignado anteriormente sólo con la memoria suficiente para contener una cadena más pequeña. Por ejemplo,

```
char una_cadena[] = "MAL";
```

En esta declaración `una_cadena` apunta a "MAL". Si se trata de asignar "BIEN" a este arreglo, se pueden tener serios problemas. El arreglo inicialmente puede contener solamente cuatro caracteres, 'M', 'A', 'L', y un nulo. "BIEN" es de cinco caracteres, 'B', 'I', 'E', 'N', y un nulo. No se puede saber lo que el quinto carácter, el nulo, sobreescribe.



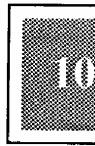
## Desplegado de cadenas y caracteres

Si el programa usa datos de cadena, probablemente necesita desplegar los datos en la pantalla en algún momento. El desplegado de cadena se hace, por lo general, con las funciones `puts()` o `printf()`.

### La función `puts()`

Ya ha visto la función de biblioteca `puts()` en algunos de los programas dados en este libro. La función `puts()` pone una cadena en la pantalla, y a esto se debe su nombre. El único argumento que toma `puts()` es un apuntador a la cadena que ha de ser desplegada. Debido a que una cadena literal evalúa como un apuntador a una cadena, `puts()` puede ser usado para desplegar cadenas literales y variables de cadena. La función `puts()` inserta automáticamente un carácter de nueva línea al final de cada cadena que despliega, por lo que cada cadena subsecuente desplegada con `puts()` se encuentra en su propia línea.

El programa del listado 10.4 ilustra el uso de `puts()`.



**Listado 10.4. Uso de la función `puts()` para desplegar texto en la pantalla.**

```
1: /* Demuestra el desplegado de cadenas con puts(). */
2:
3: #include <stdio.h>
4:
5: char *message1 = "C";
6: char *message2 = "is the";
7: char *message3 = "best";
8: char *messáge4 = "programming";
9: char *message5 = "language!!";
10:
11: main()
12: {
13:     puts(message1);
14:     puts(message2);
15:     puts(message3);
16:     puts(message4);
17:     puts(message5);
18: }
```



## 10 Caracteres y cadenas



C  
is the  
best  
programming  
language!!



Este es un listado bastante simple de seguir. Debido a que `puts()` es una función estándar de entrada/salida, "se necesita" incluir el archivo de encabezado `STDIO.H`, como se hace en la línea 3. Las líneas 5 a 9 declaran e inicializan cinco variables de mensaje diferentes. Cada una de estas variables es un apuntador a carácter o una variable de cadena. Las líneas 13 a 17 usan la función `puts()` para imprimir cada cadena.

### La función `printf()`

También puede desplegar cadenas con la función de biblioteca `printf()`. Recuerde del Día 7, "Entrada/salida básica", que `printf()` usa un formato y especificadores de conversión para darle forma a su salida. Para desplegar una cadena use el especificador de conversión `%s`.

Cuando `printf()` encuentra un `%s` en su formato, la función aparea al `%s` con el argumento correspondiente de su lista de argumentos. Cuando se trata de una cadena, este argumento debe ser un apuntador a la cadena que se quiere desplegar. La función `printf()` despliega la cadena en la pantalla, deteniéndose cuando llega al carácter nulo terminal de la cadena. Por ejemplo,

```
char *cad = "Un mensaje por desplegar";
printf("%s", cad);
```

También se pueden desplegar varias cadenas y mezclarlas con texto literal y/o variables numéricas.

```
char *banco = "Banco S.A.";
char *nombre = "Juan Pérez";
int saldo = 1000;
printf("El saldo en %s de %s es %d.", banco, nombre, saldo);
```

La salida resultante es

```
El saldo en Banco S.A. de Juan Pérez es 1000.
```

Por ahora esta información debe serle suficiente para que sea capaz de desplegar datos de cadena en los programas. En el Día 14, "Trabajando con la pantalla, la impresora y el teclado", se dan detalles completos sobre el uso de `printf()`.

## Lectura de cadenas desde el teclado

Además de desplegar cadenas, los programas necesitan frecuentemente aceptar la entrada de datos de cadenas del usuario por medio del teclado. La biblioteca del C tiene dos funciones que pueden usarse para este objetivo, `gets()` y `scanf()`. Sin embargo, antes de que pueda leer una cadena del teclado debe tener algún lugar donde ponerla. El espacio para guardar la cadena puede ser creado con cualquiera de los métodos tratados anteriormente, el día de hoy, una declaración de arreglo o la función `malloc()`.

### Entrada de cadenas con la función `gets()`

La función `gets()` obtiene una cadena del teclado. Cuando se llama a `gets()`, ésta lee todos los caracteres tecleados hasta que encuentra el primer carácter de nueva línea (que se genera cuando se oprime Enter). La función desecha a la nueva línea, añade un carácter nulo y le da la cadena al programa llamador. La cadena es guardada en la posición indicada por un apuntador de tipo `char` que ha sido pasado a `gets()`. Un programa que usa `gets()` debe incluir con `#include` el archivo `STDIO.H`. El listado 10.5 presenta un ejemplo.



**Listado 10.5. Uso de `gets()` para aceptar datos de cadena del teclado.**

```

1: /* Demuestra el uso de la función de biblioteca gets() */
2:
3: #include <stdio.h>
4:
5: /* Asigna un arreglo de caracteres para guardar la entrada */
6:
7: char input[81];
8:
9: main()
10: {
11:     puts("Enter some text, then press Enter");
12:     gets(input);
13:     printf("You entered %s", input);
14: }
```



En este ejemplo, el argumento para `gets()` es la expresión `input`, que es el nombre de un arreglo tipo `char` y, por lo tanto, un apuntador al primer elemento del arreglo. El arreglo fue declarado de 81 elementos en la línea 7. Debido a que la longitud de línea máxima posible en la mayoría de las pantallas de computadora es de 80 caracteres, este tamaño de arreglo proporciona espacio para la línea de entrada más larga posible (más el carácter nulo que `gets()` añade al final).

La función `gets()` tiene un valor de retorno que fue ignorado en el ejemplo anterior. `gets()` regresa un apuntador a tipo `char` con la dirección donde fue guardada la cadena de entrada. Sí, es el mismo valor que le fue pasado a `gets()`, pero el tener de esta manera el valor regresado le permite al programa revisar si se trata de una línea en blanco. El listado 10.6 muestra la manera de hacer esto.

## Captura

Listado 10.6. Uso del valor de retorno de `gets()` para probar la entrada de una línea en blanco.

```

1: /* Demuestra el uso del valor de retorno de gets(). */
2:
3: #include <stdio.h>
4:
5: /* Declara un arreglo de carácter para la entrada, y un apuntador. */
6:
7: char input[81], *ptr;
8:
9: main()
10: {
11:     /* Despliega instrucciones. */
12:
13:     puts("Enter text a line at a time, then press Enter.");
14:     puts("Enter a blank line when done.");
15:
16:     /* Hace ciclo mientras no se dé una línea en blanco. */
17:
18:     while (*ptr = gets(input)) != NULL)
19:         printf("You entered %s\n", input);
20:
21:     puts("Thank you and good-bye");
22: }
```

## Análisis

Ahora puede ver la manera en que funciona el programa. Si se da una línea en blanco (esto es, si simplemente se oprime Enter) en respuesta a la línea 18, la cadena (~~que~~ contiene 0 caracteres) todavía es guardada con un carácter nulo en la primera posición. Esta es la posición a la que apunta el valor de retorno de `gets()`, por lo que si se revisa ~~esa~~ posición y se encuentra un carácter nulo se sabe que fue dada una línea en blanco.

El listado 10.6 ejecuta esta prueba en el enunciado `while` que se encuentra en la línea 18. Este enunciado es un poco complicado, por lo que observe cuidadosamente sus detalles en orden.

La figura 10.1 etiqueta los componentes del enunciado para una referencia más fácil.

```

    4   3   1   2   6   5
    ↓   ↓   ↓   ↓   ↓   ↓
while (*ptr = gets(input)) != NULL)

```

**Figura 10.1.** Los componentes de un enunciado while que prueban la entrada de una línea en blanco.

1. La función `gets()` acepta entrada del teclado hasta que llega un carácter de nueva línea.
2. La cadena de entrada, menos la nueva línea y con el carácter nulo al final, es guardada en la posición de memoria apuntada por `input`.
3. La dirección de la cadena (el mismo valor que `input`) es regresada al apuntador `ptr`.
4. Un enunciado de asignación es una expresión que evalúa al valor de su variable al lado izquierdo del operador de asignación. Por lo tanto la expresión completa, `ptr = gets(input)`, evalúa al valor de `ptr`. Encerrando esta expresión en paréntesis, y precediéndola con el operador de indirección (`*`), es obtenido el valor guardado en la dirección apuntada. Esto es, por supuesto, el primer carácter de la cadena de entrada.
5. `NULL` es una constante simbólica definida en el archivo de encabezado `STDIO.H`. Tiene el valor del carácter nulo (cero).
6. Si el primer carácter de la cadena de entrada no es el carácter nulo (si no se ha tecleado una línea en blanco), el operador de comparación regresa cierto y ejecuta el ciclo `while`. Si el primer carácter es el carácter nulo (si ha tecleado una línea en blanco), el operador de comparación regresa falso y termina el ciclo `while`.

Cuando use a `gets()`, o a cualquier otra función que guarde datos usando un apuntador, asegúrese de que el apuntador apunta a algún espacio asignado. Es fácil cometer un error como el siguiente:

```

char *ptr;
gets(ptr);

```

El apuntador `ptr` ha sido declarado pero no inicializado. Apunta a algún lado pero no sabemos dónde. La función `gets()` no puede saber esto, por lo que simplemente continúa y guarda la cadena de entrada en la dirección contenida en `ptr`. La cadena puede sobreescribir algo importante, como el código de programa o el sistema operativo. El compilador no puede detectar este tipo de errores, por lo que usted, el programador, debe estar vigilando.

**Sintaxis****La función gets()**

```
#include <stdio.h>
char *gets(char *cad);
```

La función `gets()` obtiene una cadena, `cad`, a partir del dispositivo estándar de entrada, que por lo general es el teclado. La cadena consiste en cualesquier caracteres tecleados hasta que se lee un carácter de nueva línea. En este momento se añade un nulo al final de la cadena.

Luego la función `gets()` también regresa un apuntador a la cadena acabada de leer. Si hay algún problema en la obtención de la cadena, `gets()` regresa nulo.

**Ejemplo**

```
/* ejemplo de gets() */
#include <stdio.h>

char linea[256];

void main()
{
    printf( "Teclee una cadena:\n" );
    gets( linea );
    printf( "\nSe tecleó la siguiente cadena:\n" );
    printf( "%s", linea );
}
```

## Entrada de cadenas con la función `scanf()`

Se vio en el Día 7, “Entrada/salida básica”, que la función de biblioteca `scanf()` acepta entrada de datos numéricos del teclado. Esta función también puede aceptar cadenas. Recuerde que `scanf()` usa un *formato* que le indica la manera de leer la entrada. Para leer una cadena incluya el especificador `%s` en el formato de `scanf()`. De manera similar a `gets()`, a `scanf()` se le pasa un apuntador a la posición de almacenamiento de la cadena.

¿Cómo decide `scanf()` cuándo comienza y termina la cadena? El comienzo es el primer carácter diferente de espacio en blanco que encuentra. El final puede ser especificado de alguna de dos maneras. Si se usa `%s` en el formato, la cadena llega hasta (pero no incluye) el siguiente carácter de espacio en blanco (espacio, tabulador o nueva línea). Si se usa `%ns` (donde `n` es una constante entera que especifica el ancho de campo), `scanf()` da entrada a los siguientes `n` caracteres o hasta que encuentre un carácter de espacio en blanco, lo que suceda primero.

Se pueden leer varias cadenas con `scanf()` incluyendo más de un `%s` en el formato. Para cada `%s` del formato, `scanf()` usa las reglas anteriores para encontrar la cantidad de cadenas pedidas a la entrada. Por ejemplo,

```
scanf ("%s%s%s", s1, s2, s3);
```

- Si en respuesta a este enunciado se teclea enero febrero marzo, enero es asignado a la cadena s1, febrero es asignado a s2 y Marzo es asignado a s3.

¿Qué pasa si se usa el especificador de ancho de campo? Si se ejecuta el enunciado

```
scanf ("%3s%3s%3s", s1, s2, s3);
```

y en respuesta se teclea

noviembre

nov es asignado a s1, iem es asignado a s2 y bre es asignado a s3.

¿Qué pasa si se teclean menos o más cadenas que las que espera la función scanf ()? Si se teclean menos cadenas, scanf () continúa "esperando" las cadenas faltantes y el programa no continúa sino hasta que sean tecleadas. Por ejemplo, si en respuesta al enunciado

```
scanf ("%s%s%s", s1, s2, s3);
```

se teclea

enero febrero

el programa se sienta a esperar la tercera cadena especificada en el formato de scanf (). Si se teclean más cadenas que las pedidas, las cadenas sobrantes permanecen pendientes (esperando en el buffer de teclado) y son leídas por cualquier scanf () subsecuente o cualquier otro enunciado de entrada. Por ejemplo,

```
scanf ("%s%s", s1, s2);
scanf ("%s", s3);
```

Si se teclea

enero febrero marzo

el resultado es que enero es asignado a la cadena s1, febrero es asignado a s2 y marzo a s3.

La función scanf () tiene un valor de retorno, un valor entero que es igual a la cantidad de conceptos recibidos satisfactoriamente. Frecuentemente es ignorado el valor de retorno. Cuando se está leyendo solamente texto, es preferible, por lo general, usar la función gets () en vez de scanf (). La función scanf () es mejor usada cuando se está leyendo una combinación de texto y datos numéricos. Esto es ilustrado por el programa del listado 10.7. Recuerde del Día 7, "Entrada/salida básica", que se debe usar el operador de dirección de (&) cuando se reciban variables numéricas con scanf () .

```
scanf ("%s%s%s", s1, s2, s3);
```

- Si en respuesta a este enunciado se teclea enero febrero marzo, enero es asignado a la cadena s1, febrero es asignado a s2 y Marzo es asignado a s3.

¿Qué pasa si se usa el especificador de ancho de campo? Si se ejecuta el enunciado

```
scanf ("%3s%3s%3s", s1, s2, s3);
```

y en respuesta se teclea

noviembre

nov es asignado a s1, iem es asignado a s2 y bre es asignado a s3.

¿Qué pasa si se teclean menos o más cadenas que las que espera la función scanf ()? Si se teclean menos cadenas, scanf () continúa "esperando" las cadenas faltantes y el programa no continúa sino hasta que sean tecleadas. Por ejemplo, si en respuesta al enunciado

```
scanf ("%s%s%s", s1, s2, s3);
```

se teclea

enero febrero

el programa se sienta a esperar la tercera cadena especificada en el formato de scanf (). Si se teclean más cadenas que las pedidas, las cadenas sobrantes permanecen pendientes (esperando en el buffer de teclado) y son leídas por cualquier scanf () subsecuente o cualquier otro enunciado de entrada. Por ejemplo,

```
scanf ("%s%s", s1, s2);
scanf ("%s", s3);
```

Si se teclea

enero febrero marzo

el resultado es que enero es asignado a la cadena s1, febrero es asignado a s2 y marzo a s3.

La función scanf () tiene un valor de retorno, un valor entero que es igual a la cantidad de conceptos recibidos satisfactoriamente. Frecuentemente es ignorado el valor de retorno. Cuando se está leyendo solamente texto, es preferible, por lo general, usar la función gets () en vez de scanf (). La función scanf () es mejor usada cuando se está leyendo una combinación de texto y datos numéricos. Esto es ilustrado por el programa del listado 10.7. Recuerde del Día 7, "Entrada/salida básica", que se debe usar el operador de dirección de (&) cuando se reciban variables numéricas con scanf ().

## Captura

Listado 10.7. Entrada de datos numéricos y texto con `scanf()`.

```

1: /* Muestra el uso de scanf() para entrada de datos numéricos y texto */
2:
3: #include <stdio.h>
4:
5: char lname[81], fname[81];
6: int count, id_num;
7:
8: main()
9: {
10:    /* Da indicaciones al usuario. */
11:
12:    puts("Enter last name, first name, ID number separated");
13:    puts("by spaces, then press Enter.");
14:
15:    /* Recibe los tres conceptos de datos. */
16:
17:    count = scanf("%s%s%d", lname, fname, &id_num);
18:
19:    /* Despliega los datos. */
20:
21:    printf("%d items entered: %s %s %d", count, fname, lname,
22:           id_num);

```

## Análisis

Recuerde que `scanf()` requiere las direcciones de las variables como parámetros. En el listado 10.7 `lname` y `fname` son apuntadores (esto es, direcciones), por lo que no necesitan el operador de *dirección de* (`&`). Por el contrario, `id_num` es un nombre de variable regular, por lo que requiere el `&` cuando es pasado a `scanf()` en la línea 17.

Algunos programadores sienten que la entrada de datos con `scanf()` propicia los errores. Ellos prefieren recibir todos los datos, numéricos y de cadena, usando `gets()`, y luego hacer que el programa separe los números y los convierta a variables numéricas. Estas técnicas están más allá del alcance de este libro, pero harían un buen ejercicio de programación. Para esta tarea se necesitan las funciones para manipulación de cadenas, tratadas en el Día 17, "Manipulación de cadenas".

## Resumen

Este capítulo trató el tipo de dato `char` del C. Un uso para las variables de tipo `char` es el almacenamiento de caracteres individuales. Se vio que, de hecho, los caracteres son guardados como números: el código ASCII tiene asignado un código numérico para cada carácter. Por lo tanto se puede usar al tipo `char` para guardar también valores enteros pequeños. Se encuentran disponibles los tipos `char` con signo y sin signo.

Una cadena es una secuencia de caracteres terminada por el carácter nulo. Las cadenas pueden usarse para datos de texto. El C guarda las cadenas en arreglos de tipo `char`. Para guardar una cadena de longitud  $n$ , se necesita un arreglo de tipo `char` con  $n+1$  elementos.

Se pueden usar funciones de asignación de memoria, como `malloc()`, para hacer más dinámico el programa. Usando `malloc()` se puede asignar la cantidad correcta de memoria para el programa. Sin estas funciones se tendría que adivinar la cantidad de espacio de memoria que necesita el programa. Los estimados son, por lo general, altos, por lo que se asigna más memoria que la necesaria.

## Reguntas y respuestas

1. ¿Cuál es la diferencia entre una cadena y un arreglo de caracteres?

Una cadena está definida como una secuencia de caracteres que termina en el carácter nulo. Un arreglo es una secuencia de caracteres. Por lo tanto, una cadena es un arreglo de caracteres terminado en nulo.

Si se define un arreglo de tipo `char`, el espacio actual de almacenamiento asignado para el arreglo es el tamaño especificado, y no el tamaño -1. Se está limitado a ese tamaño. No se puede guardar una cadena más grande. A continuación se presenta un ejemplo:

```
char estado[10] = "Aguascalientes"; /* ;Error! La cadena es más larga */
                                         /* que el arreglo */
char estado[10] = "Ags";                /* Correcto, pero desperdicia espacio porque */
                                         /* la cadena es más corta que el arreglo. */
```

Si, por otro lado, se define un apuntador a tipo `char`, estas restricciones no se aplican. La variable es un espacio de almacenamiento solamente para el apuntador. Las cadenas actuales son guardadas en cualquier lugar de memoria (pero no hay de qué preocuparse acerca de dónde están en la memoria). No hay restricciones de longitud o espacio desperdiciado. La cadena actual es guardada en cualquier lado. Un apuntador puede ser apuntado a una cadena de cualquier longitud.

2. ¿Por qué no debo simplemente declarar arreglos grandes para guardar valores, en vez de usar una función para asignación de memoria, como `malloc()`?

Aunque puede parecer más fácil declarar arreglos grandes, no es un uso eficiente de memoria. Cuando se escriben programas pequeños, como los que se muestran en este capítulo, puede parecer trivial el uso de una función como `malloc()` en vez de arreglos, pero conforme los programas se hacen más grandes, se querrá ser

capaz de asignar memoria solamente cuando se necesita. Cuando se termina de usar la memoria se la puede regresar, *liberándola*. Cuando se libera memoria, alguna otra variable o arreglo que se encuentra en otra parte del programa puede usar la memoria. (El Día 7, “Entrada/salida básica”, trata la liberación de memoria asignada.)

3. ¿Todas las computadoras aceptan el juego de caracteres ASCII extendido?  
No. La mayoría de las PC aceptan el juego ASCII extendido. Algunas PC antiguas no, pero la cantidad de PC antiguas a las que les falta este soporte está disminuyendo. La mayoría de los programadores usan los caracteres de línea y bloque del juego extendido.
4. ¿Qué pasa si pongo en un arreglo de caracteres una cadena que es más grande que el arreglo?  
Esto causa un error difícil de encontrar. Se puede hacer esto en C, pero es sobreescrita cualquier cosa que se encuentra guardada en la memoria inmediatamente después del arreglo de caracteres. Esta puede ser un área de memoria que no está siendo usada, algunos otros datos o alguna información vital del sistema. Los resultados dependen de lo que se sobreesciba. Muchas veces no sucede nada... por un tiempo. Usted no querrá que suceda eso.

## Taller

### Cuestionario

1. ¿Qué es el rango de valores numéricos en el juego de caracteres ASCII?
2. Cuando el compilador C encuentra un solo carácter encerrado en comillas simples, ¿cómo es interpretado?
3. ¿Cuál es la definición del C para una cadena?
4. ¿Qué es una cadena literal?
5. Para guardar una cadena de  $n$  caracteres se necesita un arreglo de caracteres de  $n + 1$  elementos. ¿Para qué se necesita el elemento extra?
6. Cuando el compilador C encuentra una cadena literal, ¿cómo es interpretada?
7. Usando la tabla ASCII del apéndice A, “Tabla de caracteres ASCII”, encuentre los valores numéricos guardados para cada uno de los siguientes.
  - a. a
  - b. A

- c. 9
  - d. un espacio
  - e.
  - f. ♠
8. Usando la Tabla ASCII del apéndice A, “Tabla de caracteres ASCII”, traduzca los siguientes valores numéricos a sus caracteres equivalentes.

- a. 73
- b. 32
- c. 99
- d. 97
- e. 110
- f. 0
- g. 2

9. ¿Qué tantos bytes de almacenamiento se encuentran asignados para cada una de las siguientes variables?

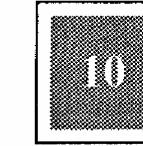
- a. char \*cad1 = { "Cadena 1"};
- b. char cad2[] = { "Cadena 2"};
- c. char cadena3;
- d. char cad4[20] = { "Esta es la cadena 4" };
- e. char cad5[20];

10. Usando la siguiente declaración:

```
char *cadena = "¡Una cadena!";
```

¿Cuáles son los valores de lo siguiente?

- a. cadena[0];
- b. \*cadena
- c. cadena[9]
- d. cadena[33]
- e. \*cadena+8
- f. cadena





### Ejercicios

1. Escriba una línea de código que declare una variable tipo `char` llamada `letra e` inicialícela al carácter `$`.
2. Escriba una línea de código que declare un arreglo de tipo `char`, `e` inicialícelo a la cadena "Los apuntadores son divertidos". Haga que el arreglo sea sólo lo bastante grande como para que guarde la cadena.
3. Escriba una línea de código que asigne almacenamiento para la cadena "Los apuntadores son divertidos", como en el ejercicio 2, pero sin usar un arreglo.
4. Escriba código que asigne espacio para una cadena de 80 caracteres, y luego reciba la cadena del teclado y guárdela en el espacio asignado.
5. Escriba una función que copie un arreglo de caracteres a otro. (Consejo: hágalo de manera parecida a los programas que hizo en el Día 9, "Apuntadores".)
6. Escriba una función que acepte dos cadenas. Cuente la cantidad de caracteres en cada una y regrese un apuntador a la cadena más larga.
7. Escriba una función que acepte dos cadenas. Use la función `malloc()` para asignar suficiente memoria para guardar las dos cadenas después de que hayan sido concatenadas (puestas juntas). Regrese un apuntador a esta nueva cadena.

Por ejemplo, si paso "Hola" y "Amigos", la función regresa un apuntador a "Hola Amigos". Haciendo que el valor concatenado sea la tercera cadena, es más fácil. (Tal vez pueda usar las respuestas de los ejercicios 5 y 6.)

8. **BUSQUEDA DE ERRORES:** ¿Hay algo erróneo en lo siguiente?

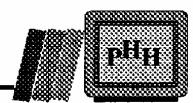
```
char a_string[10] = "This is a string";
```

9. **BUSQUEDA DE ERRORES:** ¿Hay algo erróneo en lo siguiente?

```
char *quote[100] = { "Smile, Friday is almost here!" };
```

10. **BUSQUEDA DE ERRORES:** ¿Hay algo erróneo en lo siguiente?

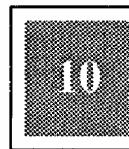
```
char *string1;
char *string2 = "Second";
string1 = string2;
```

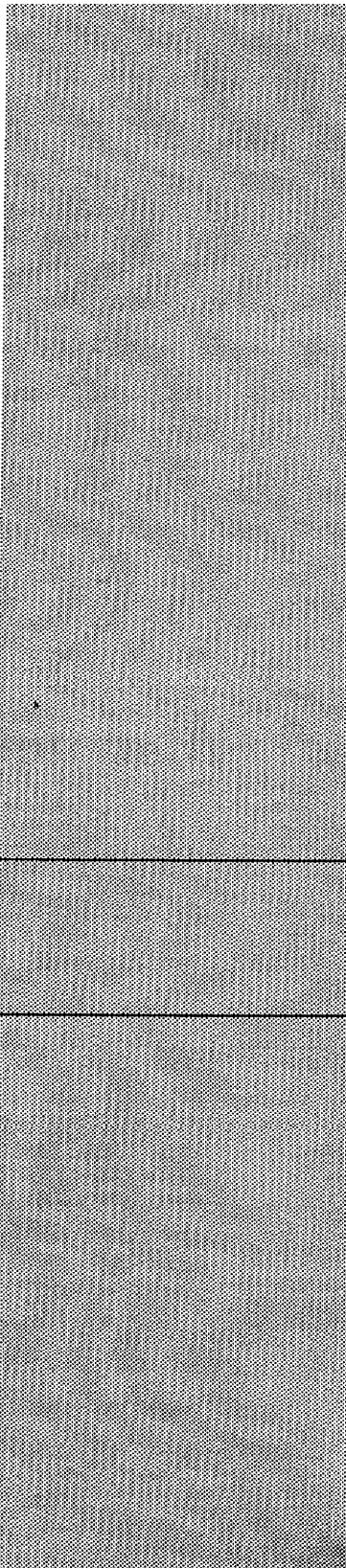


11. BUSQUEDA DE ERRORES: ¿Hay algo erróneo en lo siguiente?

```
char string1[];  
char string2[] = "Second";  
string1 = string2;
```

12. Usando la tabla de caracteres ASCII escriba un programa que trace un cuadro en la pantalla usando los caracteres de doble línea.







## Estructuras

Muchas tareas de programación se simplifican con la construcción de datos del C llamada *estructura*. Una estructura es un método de almacenamiento de datos designado por usted, el programador, para que se ajuste exactamente a las necesidades de programación. Hoy aprenderá

- Qué son las estructuras simples y complejas.
- Cómo definir y declarar estructuras.
- Cómo accesar datos en estructuras.
- Cómo crear estructuras que contienen arreglos y arreglos de estructuras.
- Cómo declarar apuntadores en estructuras y apuntadores a estructuras.
- Cómo pasar estructuras como argumentos a funciones.
- Cómo definir, declarar y usar uniones.
- Cómo usar estructuras para crear listas encadenadas.

## Estructuras simples

Una estructura es una colección de una o más variables, agrupadas bajo un solo nombre para facilitar su manejo. Las variables en la estructura, a diferencia de las que se encuentran en arreglos, pueden ser de diferentes tipos de variable. Una estructura puede contener cualquiera de los tipos de datos del C, incluyendo arreglos y otras estructuras. Cada variable dentro de una estructura es llamada un *miembro* de la estructura. La siguiente sección muestra un ejemplo simple.

Se debe comenzar con estructuras simples. Observe que el lenguaje C no hace distinción entre estructuras simples y complejas, pero es más fácil explicar las estructuras de esta manera.

## Definición y declaración de estructuras

Si se está escribiendo un programa para gráficos, el código necesita manejar las coordenadas de los puntos en la pantalla. Las coordenadas de pantalla son escritas como un valor *x*, que da la posición horizontal, y un valor *y* que da la posición vertical. Se puede definir una estructura, llamada *coord*, que contiene los valores *x* y *y* de una posición de pantalla, de la manera siguiente:

```
struct coord{  
    int x;  
    int y;  
};
```

La palabra clave **struct**, que identifica el comienzo de una definición de estructura, debe ser seguida inmediatamente por el nombre de la estructura o *etiqueta* (que sigue las mismas reglas que los otros nombres de variables del C). Dentro de las llaves que se encuentran a continuación del nombre de la estructura va una lista de las variables miembro de la estructura. Se debe dar un tipo de variable y un nombre para cada miembro.

Los enunciados anteriores definen un tipo de estructura, llamada **coord**, que contiene dos variables enteras, **x** y **y**. Sin embargo, ellos no crean, de hecho, ninguna instancia de la estructura **coord**. Esto es, no declaran (reservan espacio para) ninguna estructura. Hay dos maneras de declarar estructuras. Una es poner a continuación de la definición de la estructura una lista de uno o más nombres de variables, como se hace aquí:

```
struct coord{
    int x;
    int y;
} primera, segunda;
```

Estos enunciados definen la estructura tipo **coord** y declaran dos estructuras, llamadas **primera** y **segunda**, de tipo **coord**. **primera** y **segunda** son cada una *instancias* de tipo **coord**; **primera** contiene dos miembros enteros llamados **x** y **y**, y lo mismo sucede con **segunda**.

El método anterior para declarar estructuras combina la definición con la declaración. El segundo método es declarar las variables de estructura en una posición diferente de la definición en el código fuente. Los siguientes enunciados también declaran dos instancias de tipo **coord**:

```
struct coord{
    int x;
    int y;
};
/* Aquí puede ir código adicional */
struct coord primera, segunda;
```

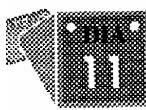
## Acceso de los miembros de la estructura

Los miembros individuales de la estructura pueden usarse de manera similar a otras variables del mismo tipo. Los miembros de estructura son accesados usando el *operador de miembro de estructura* (**.**), también llamado el *operador de punto*, entre el nombre de la estructura y el nombre del miembro. Por lo tanto, para hacer que la estructura llamada **primera** haga referencia a una posición de pantalla con coordenadas **x = 50**, **y = 100**, se podría escribir

```
primera.x = 50;
primera.y = 100;
```

Para desplegar las posiciones de pantalla guardadas en la estructura **segunda**, se podría escribir

```
printf( "%d,%d", segunda.x, segunda.y);
```



## Estructuras

En este momento tal vez se pregunte cuál es la ventaja de usar estructuras en vez de usar variables individuales. Una gran ventaja es la habilidad de copiar información entre estructuras del mismo tipo con un simple enunciado de asignación. Continuando con el ejemplo anterior, el enunciado

primera = segunda;

es equivalente a

```
primera.x = segunda.x;  
primera.y = segunda.y;
```

Cuando el programa usa estructuras complejas con muchos miembros esta notación puede ahorrar mucho tiempo. Otras ventajas de las estructuras serán evidentes conforme aprenda más capacidades avanzadas. En general, encontrará que las estructuras son útiles en cualquier momento en que información de diferentes tipos de variables necesita ser tratada como un grupo. Por ejemplo, en una base de datos de listas de direcciones cada registro puede ser una estructura, y cada pieza de información (nombre, dirección, ciudad, etc.), un miembro de la estructura.

### Sintaxis

```
struct etiqueta {  
    miembros_de_estructura  
    /* aquí pueden ir enunciados adicionales */  
} instancia;
```

La palabra clave **struct** es usada para la declaración de estructuras. Una estructura es una colección de una o más variables (*miembros\_de\_estructura*) que han sido agrupadas bajo un solo nombre para facilitar su manejo. Las variables no tienen que ser del mismo tipo ni deben ser variables simples. Las estructuras también pueden contener arreglos, apuntadores y otras estructuras.

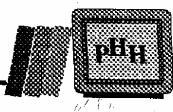
La palabra clave **struct** identifica el inicio de una definición de estructura. Es seguida por una etiqueta, que es el nombre dado a la estructura. A continuación de la etiqueta se encuentran los miembros de la estructura encerrados entre llaves. Una *instancia*, la declaración actual de una estructura, también puede ser definida. Si se define la estructura sin la instancia es simplemente una plantilla, que puede ser usada posteriormente en un programa para declarar estructuras. A continuación se presenta el formato de la plantilla.

```
struct etiqueta {  
    miembros_de_estructura  
    /* aquí pueden ir enunciados adicionales */  
};
```

Para usar la plantilla se podría usar el siguiente formato:

```
struct etiqueta instancia;
```

Para usar este formato se debe haber declarado previamente una estructura con la etiqueta dada.



### Ejemplo 1

```
/* Declara una plantilla de estructura llamada RFC */
struct RFC {
    char tres_primeros;
    char guion1;
    char dos_segundos;
    char guion2;
    char cuatro_últimos;
}
/* Usa la plantilla de la estructura */
struct RFC rfc_cliente;
```

### Ejemplo 2

```
/* Declara una estructura y una instancia */
struct fecha {
    char mes[2];
    char día[2];
    char año[4];
} fecha_actual;
```

### Ejemplo 3

```
/* Declara e inicializa una estructura */
struct tiempo {
    int horas;
    int minutos;
    int segundos;
} hora_de_nacimiento = { 8, 45, 0 };
```

11

## Estructuras más complejas

Ahora que ya ha visto las estructuras simples, puede usted pasar a los tipos de estructuras más interesantes y complejos. Estas son estructuras que contienen como miembros a otras estructuras, y estructuras que contienen arreglos como miembros.

### Estructuras que contienen estructuras

Como se dijo anteriormente, una estructura de C puede contener cualquiera de los tipos de datos del C. Por ejemplo, una estructura puede contener otras estructuras. Los ejemplos anteriores pueden ser extendidos para ilustrar esto.

Suponga que el programa de gráficos ahora necesita manejar rectángulos. Un *rectángulo* puede ser definido por las coordenadas de sus dos esquinas diagonalmente opuestas. Ya ha visto cómo definir una estructura que pueda guardar las dos coordenadas que se requieren para un solo punto. Necesitará dos de esta estructuras para definir un rectángulo. Se puede

definir una estructura de la manera siguiente (suponiendo, por supuesto, que ya haya definido la estructura tipo `coord`):

```
struct rectangulo {
    struct coord arribaizquierda;
    struct coord abajoderecha;
};
```

Este enunciado define una estructura de tipo `rectángulo` que contiene dos estructuras de tipo `coord`. Estas dos estructuras de tipo `coord` son llamadas `arribaizquierda` y `abajoderecha`.

El enunciado anterior define solamente la estructura tipo `rectángulo`. Para declarar una estructura se debe incluir luego un enunciado como

```
struct rectángulo micuadro;
```

Se podría haber combinado la definición y la declaración, como se hizo anteriormente para el tipo `coord`.

```
struct rectángulo {
    struct coord arribaizquierda;
    struct coord abajoderecha;
} micuadro;
```

Para accesar las posiciones actuales de datos (los miembros tipo `int`) se debe aplicar dos veces al operador de miembro `(.)`. Por lo tanto, la expresión

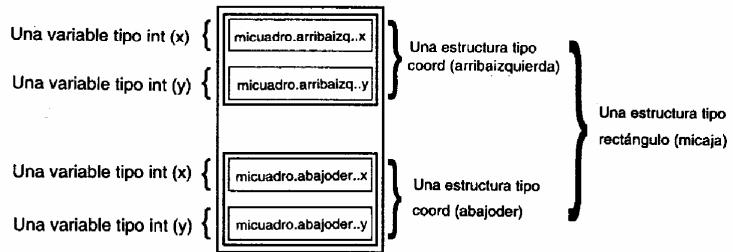
```
micuadro.arribaizquierda.x
```

hace referencia al miembro `x` del miembro `arribaizquierda` de la estructura tipo `rectángulo` llamada `micuadro`. Para definir un `rectángulo` con las coordenadas `(0, 10)`, `(100, 200)`, se podría escribir

```
micuadro.arribaizquierda.x = 0;
micuadro.arribaizquierda.y = 10;
micuadro.abajoderecha.x = 100;
micuadro.abajoderecha.y = 200;
```

Esto puede ser algo confuso. Tal vez lo comprenda mejor si ve la figura 11.1, que muestra la relación entre la estructura tipo `rectángulo`, las dos estructuras tipo `coord` que contiene y las dos variables tipo `int` que contiene cada estructura tipo `coord`. Las estructuras son nombradas igual que en el ejemplo anterior.

Es tiempo de ver un ejemplo sobre el uso de estructuras que contienen otras estructuras. El programa en el listado 11.1 recibe datos del usuario para las coordenadas de un `rectángulo`, y luego calcula y despliega el área del `rectángulo`. Observe las hipótesis del programa, que se dan en comentarios al principio del código (líneas 3 a 8).



**Figura 11.1.** Un diagrama de la relación entre una estructura, estructuras dentro de una estructura y los miembros de estructura.



### Listado 11.1. Una demostración de estructuras que contienen otras estructuras.

```
1: /* Demuestra estructuras que contienen otras estructuras. */
2:
3: /* Recibe entrada de coordenadas de esquina de un rectángulo y
4: calcula el área. Supone que la coordenada y de la esquina superior
5: izquierda es mayor que la coordenada y de la esquina inferior
6: derecha, y que la coordenada x de la esquina inferior derecha es
7: mayor que la coordenada x de la esquina superior izquierda, y que
8: todas las coordenadas son positivas. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16:     int x;
17:     int y;
18: };
19:
20: struct rectangle{
21:     struct coord topleft; int x1, int y1
22:     struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27:     /* Recibe las coordenadas */
28:
29:     printf("\nEnter the top left x coordinate: ");
30:     scanf("%d", &mybox.topleft.x);
31:
```





## Estructuras

### Listado 11.1. continuación

```
32:     printf("\nEnter the top left y coordinate: ");
33:     scanf("%d", &mybox.topleft.y);
34:
35:     printf("\nEnter the bottom right x coordinate: ");
36:     scanf("%d", &mybox.bottomrt.x);
37:
38:     printf("\nEnter the bottom right y coordinate: ");
39:     scanf("%d", &mybox.bottomrt.y);
40:
41:     /* Calcula la longitud y el ancho */
42:
43:     width = mybox.bottomrt.x - mybox.topleft.x;
44:     length = mybox.bottomrt.y - mybox.topleft.y;
45:
46:     /* Calcula y despliega el área */
47:
48:     area = width * length;
49:     printf("The area is %ld units.", area);
50: }
```



Enter the top left x coordinate: 1  
Enter the top left y coordinate: 1  
Enter the bottom right x coordinate: 10  
Enter the bottom right y coordinate: 10  
The area is 81 units.



La estructura coord se encuentra definida en las líneas 15 a 18 con sus dos miembros, x y y. Las líneas 20 a 23 declaran una instancia, llamada mybox, de la estructura rectangle. Los dos miembros de la estructura rectangle son topleft y bottomrt, siendo ambos estructuras de tipo coord.

Las líneas 29 a 39 rellenan los valores para la estructura mybox. Al principio puede parecer que hay solamente dos valores que llenar, debido a que mybox tiene solamente dos miembros. Sin embargo, cada uno de los miembros de mybox tiene sus propios miembros. topleft y bottomrt tienen dos miembros cada uno, x y y, de la estructura coord. Esto nos da un total de cuatro miembros que deben ser llenados. Después de que los miembros son llenados con valores, el área es calculada usando los nombres de estructura y de miembro. Cuando se usan los valores x y y se debe incluir el nombre de instancia de estructura. Debido a que x y y están en una estructura dentro de otra estructura, se deben usar los nombres de instancia de ambas estructuras, mybox.bottomrt.x, mybox.bottomrt.y, mybox.topleft.x y mybox.topleft.y, en los cálculos.

El C no pone límites sobre el anidado de estructuras. Mientras lo permita la memoria, se pueden definir estructuras que contengan estructuras que contengan estructuras que contengan estructuras ... Bien, ¡ya tiene usted la idea! Por supuesto que hay un límite que, al rebasarlo, el anidado llega a ser improductivo. Es muy raro que se usen más de tres niveles de anidado en cualquier programa del C.



## Estructuras que contienen arreglos

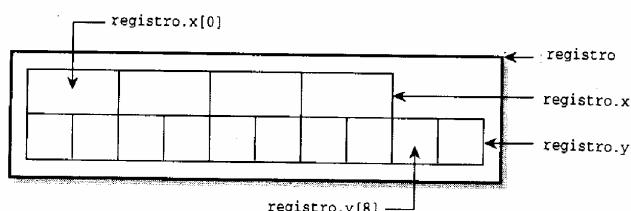
Se puede definir una estructura que contiene uno o más arreglos como miembros. El arreglo puede ser de cualquier tipo de datos del C (entero, carácter, etc.). Por ejemplo, los enunciados

```
struct datos{  
    int x[4];  
    char y[10];  
};
```

definen una estructura de tipo **datos** que contiene un miembro de arreglo entero de 4 elementos, llamado **x**, y un miembro de arreglo de carácter de 10 elementos, llamado **y**. Luego se puede declarar una estructura, llamada **registro**, de tipo **datos**, de la manera siguiente:

```
struct datos registro;
```

La organización de esta estructura se muestra en la figura 11.2. Observe que en esta figura los elementos del arreglo **x** ocupan el doble de espacio que los elementos del arreglo **y**. Esto se debe (como lo aprendió en el Día 3, “Variables y constantes numéricas”) a que un tipo **int** por lo general requiere dos bytes de almacenamiento y un tipo **char** por lo general requiere solamente de un byte.



**Figura 11.2.** La organización de una estructura que contiene arreglos como miembros.

El acceso a los miembros individuales de los arreglos que son miembros de estructuras se hace con una combinación del operador de miembro y de los subíndices de arreglo:

```
registro.x[2] = 100;  
registro.y[1] = 'x';
```

Probablemente recuerde que los arreglos de caracteres se usan, por lo general, para guardar cadenas. También debe recordar (del Día 9, “Apuntadores”) que el nombre de un arreglo sin corchetes es un apuntador al arreglo. Debido a que esto es cierto para los arreglos que son miembros de estructuras, la expresión

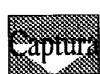
```
registro.y
```

es un apuntador al primer elemento del arreglo **y** [] en la estructura **registro**. Por lo tanto, se podría imprimir el contenido de **y** [] en la pantalla con el enunciado

## Estructuras

```
puts(registro.y);
```

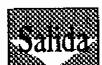
Ahora veamos otro ejemplo. El programa del listado 11.2 usa una estructura que contiene una variable tipo float y dos arreglos tipo char.



### Listado 11.2. Una demostración de una estructura que contiene miembros de arreglo.

```

1: /* Demuestra una estructura que tiene miembros de arreglo. */
2:
3: #include <stdio.h>
4:
5: /* Define y declara una estructura para guardar los datos. */
6: /* Contiene una variable float y dos arreglos char. */
7:
8: struct data{
9:     float amount;
10:    char fname[30];
11:    char lname[30];
12: } rec;
13:
14: main()
15: {
16:     /* Recibe los datos del teclado. */
17:
18:     printf("Enter the donor_s first and last names,\n");
19:     printf("separated by a space: ");
20:     scanf("%s %s", rec.fname, rec.lname);
21:
22:     printf("\nEnter the donation amount: ");
23:     scanf("%f", &rec.amount);
24:
25:     /* Despliega la información. */
26:     /* Nota: %.2f especifica un valor de punto flotante */
27:     /* a ser desplegado con dos dígitos a la derecha */
28:     /* del punto decimal. */
29:
30:     /* Despliega los datos en la pantalla. */
31:
32:     printf("\nDonor %s %s gave $%.2f.", rec.fname, rec.lname,
33:           rec.amount);
34: }
```



```

Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```



Este programa incluye una estructura que contiene miembros de arreglo llamados fname [30] y lname [30]. Ambos son arreglos de caracteres que guardan el nombre y el apellido de una persona, respectivamente. La estructura declarada en las líneas 8 a 12 es llamada **data**. Contiene los arreglos de carácter fname y lname con una variable tipo float llamada amount. Esta estructura es ideal para guardar el nombre de una persona (en dos partes, nombre y apellido) y un valor, como, por ejemplo, la cantidad que una persona ha donado a una organización de caridad.

Una instancia del arreglo, llamada **rec**, también ha sido declarada en la línea 12. El resto del programa usa **rec** para obtener valores del usuario (líneas 18 a 23) y luego imprimirlas (líneas 32 a 33).

## Arreglos de estructuras

Si se tienen estructuras que contienen arreglos, ¿se podrá también tener arreglos de estructuras? ¡Claro que se puede! De hecho, los arreglos de estructuras son una herramienta de programación muy poderosa. Aquí está cómo se hace.

Ya ha visto la manera en que una definición de estructura puede ser acondicionada para que le quepan los datos con los cuales necesita trabajar el programa. Por lo general, un programa necesita trabajar con más de una instancia de los datos. Por ejemplo, en un programa que va a manejar una lista de números de teléfonos se puede definir una estructura para que guarde el nombre y el número de cada persona.

```
struct entrada{
    char nombre[10];
    char apellido[12];
    char telefono[8];
};
```

Sin embargo, una lista de teléfonos debe guardar muchas entradas, por lo que una sola instancia de la estructura no sirve de mucho. Lo que necesita es un arreglo de estructuras de tipo **entrada**. Después de que la estructura ha sido definida, se puede declarar un arreglo de la manera siguiente:

```
struct entrada lista[1000];
```

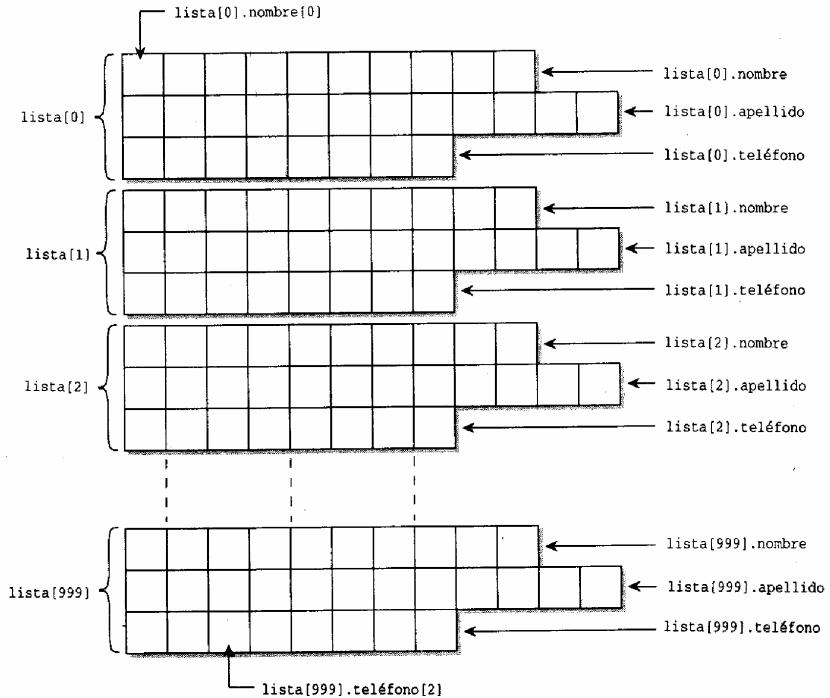
Este enunciado declara un arreglo llamado **lista** que contiene 1,000 elementos. Cada elemento es una estructura de tipo **entrada**, y es identificada con subíndices de manera similar a cualesquier otros tipos de elementos de arreglo. Cada una de estas estructuras tiene tres elementos, siendo cada uno de ellos un arreglo de tipo **char**. Esta creación compleja se diagrama por completo en la figura 11.3.

Cuando se ha declarado el arreglo de estructuras se pueden manejar los datos de muchas maneras. Por ejemplo, para asignar los datos de un elemento de arreglo a otro elemento de arreglo, se escribe

```
lista[1] = lista[5];
```



## Estructuras



**Figura 11.3.** Un diagrama de la relación entre una estructura, estructuras dentro de una estructura y los miembros de estructura.

Este enunciado asigna a cada miembro de la estructura `lista[1]` los valores contenidos en los miembros correspondientes de `lista[5]`. También se pueden mover datos entre miembros individuales de la estructura. El enunciado

```
strcpy(lista[1].teléfono, lista[5].teléfono);
```

copia la cadena que se encuentra en `lista[5].teléfono` a `lista[1].teléfono`. (La función de biblioteca `strcpy()` copia una cadena a otra cadena. Aprenderá los detalles de esto en el Día 17, “Manipulación de cadenas”.) También puede, si lo desea, mover datos entre elementos individuales de los arreglos miembros de la estructura:

```
lista[5].teléfono[1] = lista[2].teléfono[3];
```

Este enunciado mueve el segundo carácter del número de teléfono de `lista[5]` a la cuarta posición del número de teléfono de `lista[2]`. (No olvide que los subíndices se iniciaron en 0.)

El programa del listado 11.3 demuestra el uso de arreglos de estructuras. Es más, demuestra arreglos de estructuras que contienen arreglos como miembros.



### Listado 11.3. Demostración de arreglos de estructuras.

```
1: /* Demuestra el uso de arreglos de estructuras. */
2:
3: #include <stdio.h>
4:
5: /* Define una estructura para guardar las entradas. */
6:
7: struct entry {
8:     char fname[20];
9:     char lname[20];
10:    char phone[10];
11: };
12:
13: /* Declara un arreglo de estructuras. */
14:
15: struct entry list[4];
16:
17: int i;
18:
19: main()
20: {
21:
22:     /* Hace ciclo para recibir datos de cuatro personas. */
23:
24:     for (i = 0; i < 4; i++)
25:     {
26:         printf("\nEnter first name: ");
27:         scanf("%s", list[i].fname);
28:         printf("Enter last name: ");
29:         scanf("%s", list[i].lname);
30:         printf("Enter phone in 123-4567 format: ");
31:         scanf("%s", list[i].phone);
32:     }
33:
34:     /* Imprime dos líneas en blanco. */
35:
36:     printf("\n\n");
37:
38:     /* Hace ciclo para desplegar los datos. */
39:
40:     for (i = 0; i < 4; i++)
41:     {
42:         printf("Name: %s %s", list[i].fname, list[i].lname);
43:         printf("\t\tPhone: %s\n", list[i].phone);
44:     }
45: }
```



## Estructuras

---



```
Enter first name: Bradley  
Enter last name: Jones  
Enter phone in 123-4567 format: 555-1212
```

```
Enter first name: Peter  
Enter last name: Aitken  
Enter phone in 123-4567 format: 555-3434
```

```
Enter first name: Melissa  
Enter last name: Jones  
Enter phone in 123-4567 format: 555-1212
```

```
Enter first name: John  
Enter last name: Smith  
Enter phone in 123-4567 format: 555-1234
```

Name: Bradley Jones	Phone: 555-1212
Name: Peter Aitken	Phone: 555-3434
Name: Melissa Jones	Phone: 555-1212
Name: John Smith	Phone: 555-1234



Este listado sigue el mismo formato general que la mayoría de los otros listados. Comienza con los comentarios en la línea 1, y para las funciones de entrada/salida el #include para el archivo STDIO.H en la línea 3. Las líneas 7 a 11 definen una plantilla de estructura, llamada entry, que contiene tres arreglos de carácter, fname, lname y phone. La línea 15 usa la plantilla para definir un arreglo de cuatro variables de la estructura entry llamadas list. La línea 17 define una variable de tipo int para ser usada como contador en el programa. main() se inicia en la línea 19. La primera función de main() es ejecutar un ciclo cuatro veces con un enunciado for. Este ciclo es usado para obtener información para el arreglo de estructuras. Esto puede verse en las líneas 24 a 32. Observe que list es usado con subíndice, de manera similar a como fueron usadas con subíndice las variables de arreglo que se vieron en el Día 8, "Arreglos numéricos".

La línea 36 proporciona un corte para la entrada, antes de comenzar la salida. Ella imprime dos líneas en blanco, en una forma que no debe serle extraña. Las líneas 40 a 44 despliegan los datos que el usuario tecleó en el paso anterior. Los valores en el arreglo de estructuras son impresos con el nombre de arreglo con subíndices, seguido por el operador de miembro (.) y el nombre del miembro de la estructura.

Debe familiarizarse con las técnicas usadas en el listado 11.3. Muchas tareas de programación reales se logran mejor usando arreglos de estructuras que contiene arreglos como miembros.



## DEBE

## NO DEBE

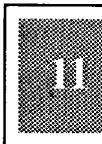
**NO DEBE** Olvidar el nombre de instancia de la estructura y al operador de miembro (.) cuando use miembros de estructuras.

**NO DEBE** Confundir la etiqueta de la estructura con sus instancias! La etiqueta es para declarar la plantilla o formato de la estructura. La instancia es una variable declarada usando la etiqueta.

**NO DEBE** Olvidar la palabra clave struct cuando declare una instancia de una estructura previamente definida.

**DEBE** Declarar instancias de estructuras con las mismas reglas de alcance que otras variables. (En el Día 12, "Alcance de las variables", se trata este tema a profundidad.)

## Inicialización de estructuras



De manera similar a otros tipos de variables del C, las estructuras pueden ser inicializadas cuando son declaradas. El procedimiento es similar al que se usa para inicializar arreglos. La declaración de estructura es seguida por un signo de igual y una lista de valores de inicialización, separados por comas y encerrados entre llaves. Por ejemplo, vea los siguientes enunciados:

```
1: struct venta {  
2:     char cliente[20];  
3:     char concepto[20];  
4:     float importe;  
5: } miventa = { "Industrias Acme",  
6:                 "Escritorio",  
7:                 1000.00  
8:             };
```

Cuando estos enunciados se ejecutan, ejecutan las siguientes acciones:

1. Define un tipo de estructura llamado `venta` (líneas 1 a 5).
2. Declara una instancia de la estructura tipo `venta` llamada `miventa` (línea 5).
3. Inicializa el miembro `miventa.cliente` de la estructura a la cadena "Industrias Acme" (línea 5).
4. Inicializa el miembro `miventa.concepto` de la estructura a la cadena "Escritorio" (línea 6).
5. Inicializa el miembro `miventa.importe` de la estructura al valor 1000.00 (línea 7).



## Estructuras

Para una estructura que contiene estructuras como miembros, liste los valores de inicialización en orden. Ellos son puestos en las estructuras miembro en el orden en que son listados los miembros en la definición de la estructura. A continuación se presenta un ejemplo que expande ligeramente al anterior:

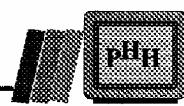
```
1: struct cliente {  
2:     char empresa[20];  
3:     char representante[25];  
4: };  
5:  
6: struct venta {  
7:     struct cliente comprador;  
8:     char concepto[20];  
9:     float importe;  
10: } miventa = { {"Industrias Acme", "Juan Pérez"},  
11:                 "Escritorio",  
12:                 1000.00  
13:             };
```

Estos enunciados ejecutan las siguientes inicializaciones:

1. El miembro `miventa.comprador.empresa` de la estructura es inicializado a la cadena “Industrias Acme” (línea 10).
2. El miembro `miventa.comprador.representante` de la estructura es inicializado a la cadena “Juan Pérez” (línea 10).
3. El miembro `miventa.concepto` de la estructura es inicializado a la cadena “Escritorio” (línea 11).
4. El miembro `miventa.importe` de la estructura es inicializado a la cantidad 1000.00 (línea 12).

También se puede inicializar arreglos de estructuras. Los datos de inicialización que se proporcionan son aplicados en orden a las estructuras en el arreglo. Por ejemplo, para declarar un arreglo de estructuras de tipo `venta` e inicializar los primeros dos elementos del arreglo (esto es, las dos primeras estructuras), se podría escribir:

```
1: struct cliente {  
2:     char empresa[20];  
3:     char representante[25];  
4: };  
5:  
6: struct venta {  
7:     struct cliente comprador;  
8:     char concepto[20];  
9:     float importe;  
10: };
```



```
11:  
12:  
13: struct venta y1990[100] = {  
14:     {{ "Industrias Acme", "Juan Pérez"},  
15:         "Escritorio",  
16:         1000.00  
17:     }  
18:     {{ "Compañía Wilson", "Pedro Wilson"},  
19:         "Silla modelo 12",  
20:         290.00  
21:     }  
22: };
```

1. El miembro `y1990[0].comprador.empresa` de la estructura es inicializado a la cadena "Industrias Acme" (línea 14).
2. El miembro `y1990[0].comprador.representante` de la estructura es inicializado a la cadena "Juan Pérez" (línea 14).
3. El miembro `y1990[0].concepto` de la estructura es inicializado a la cadena "Escritorio" (línea 15).
4. El miembro `y1990[0].importe` de la estructura es inicializado a la cantidad 1000.00 (línea 16).
5. El miembro `y1990[1].comprador.empresa` de la estructura es inicializado a la cadena "Compañía Wilson" (línea 18).
6. El miembro `y1990[1].comprador.representante` de la estructura es inicializado a la cadena "Pedro Wilson" (línea 18).
7. El miembro `y1990[1].concepto` de la estructura es inicializado a la cadena "Silla modelo 12" (línea 19).
8. El miembro `y1990[1].importe` de la estructura es inicializado a la cantidad 290.00 (línea 20).



## Estructuras y apuntadores

Debido a que los apuntadores son una parte tan importante del C, no debe sorprenderle que puedan usarse con las estructuras. Se pueden usar apuntadores como miembros de la estructura, y también se pueden declarar apuntadores a las estructuras. Estos se tratan, a su vez, en los siguientes párrafos.



### Apuntadores como miembros de estructuras

Se tiene una flexibilidad completa para usar apuntadores como miembros de estructura. Los miembros de apuntador son declarados en la misma forma que los apuntadores que no son miembros de estructura, esto es, usando el operador de indirección (\*). A continuación se presenta un ejemplo:

```
struct datos {  
    int *valor;  
    int *tasa;  
} primera;
```

Estos enunciados definen y declaran una estructura cuyos dos miembros son apuntadores a tipo int. Como sucede con todos los apuntadores, no es suficiente declararlos. Se debe, asignándoles la dirección de una variable, inicializarlos para que apunten a algo. Si costo e interés han sido declarados para ser variables de tipo int, se podría escribir

```
primera.valor = &costo;  
primera.tasa = &interés;
```

Ahora que los apuntadores han sido inicializados, se puede usar el operador de indirección (\*) como se explicó en el Día 9, "Apuntadores". La expresión `*primera.valor` evalúa al valor de costo, y la expresión `*primera.tasa` evalúa al valor de interés.

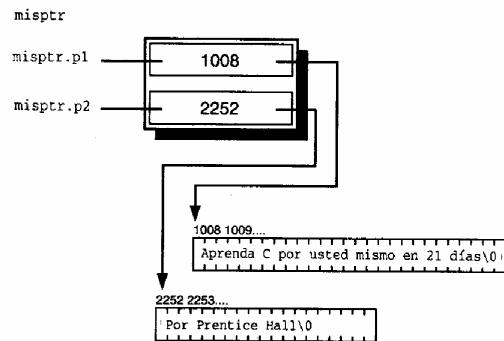
Tal vez el tipo más frecuente de apuntador usado como miembro de estructura es un apuntador a tipo char. Recuerde del Día 10, "Caracteres y cadenas", que una cadena es una secuencia de caracteres delineada por un apuntador que apunta al primer carácter de la cadena y un carácter nulo que indica el fin de la cadena. Por si lo ha olvidado, se puede declarar un apuntador a tipo char e inicializarlo para que apunte a una cadena de la manera siguiente:

```
char *p_mensaje;  
p_mensaje = "Aprenda C por usted mismo en 21 días";
```

Se puede hacer lo mismo con apuntadores a tipo char que son miembros de estructuras:

```
struct msg {  
    char *p1;  
    char *p2;  
} misptr;  
  
misptr.p1 = "Aprenda C por usted mismo en 21 días";  
misptr.p2 = "Por Prentice Hall";
```

La figura 11.4 ilustra el resultado de la ejecución de los enunciados anteriores. Cada apuntador miembro de la estructura apunta al primer byte de una cadena, guardada en cualquier lugar de memoria. Compare esto con la figura 11.3, que muestra la manera en que los datos son guardados en una estructura que contiene arreglos de tipo char.



**Figura 11.4. Una estructura que contiene apuntadores a tipo char.**

Se puede usar apuntadores miembros de estructura en cualquier lugar en que pueda ser utilizado un apuntador. Por ejemplo, para imprimir las cadenas apuntadas se podría escribir

```
printf( "%s %s, misptr.p1, misptr.p2);
```

¿Cuál es la diferencia entre usar un arreglo de tipo char como miembro de estructura y usar un apuntador a tipo char? Ambos son métodos para “guardar” una cadena en una estructura, como se muestra aquí en la estructura msg que emplea ambos métodos:

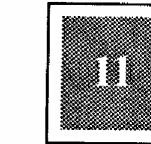
```
struct msg {
    char p1[10];
    char *p2;
} misptr;
```

Recuerde que un nombre de arreglo sin corchetes es un apuntador al primer elemento del arreglo. Por lo tanto, puede usar estos dos miembros de estructura en forma similar:

```
strcpy(misptr.p1, "Aprenda C por usted mismo en 21 días");
strcpy(misptr.p2, "Por Prentice Hall");
/* aquí va código adicional */
puts(misptr.p1);
puts(misptr.p2);
```

¿Cuál es la diferencia entre estos métodos? Es ésta: si se define una estructura que contiene un arreglo de tipo char, cada instancia de ese tipo de estructura contiene espacio de almacenamiento para el arreglo del tamaño especificado. Es más, se está limitado al tamaño especificado y no se puede guardar una cadena más grande en la estructura. Este es un ejemplo:

```
struct msg {
    char p1[10];
    char p2[10];
} misptr;
...
```





## Estructuras

```
strcpy(p1, "Aguascalientes"); /* ¡Erróneo! La cadena es más larga */
                                /* que el arreglo. */
strcpy(p2, "Ags");           /* Correcto, pero desperdicia espacio */
                                /* debido a que la cadena es más corta que el arreglo.*/
```

Si, por otro lado, se define una estructura que contiene apunadores a tipo char, estas restricciones no se aplican. Cada instancia de la estructura contiene espacio de almacenamiento solamente para el apuntador. Las cadenas actuales son guardadas en cualquier lugar de memoria (pero usted no tiene que preocuparse acerca de *dónde* en memoria). No hay restricción de longitud ni espacio desperdiciado. Las cadenas actuales son guardadas en cualquier lado y no como parte de la estructura. Cada apuntador en la estructura puede apuntar a una cadena de cualquier longitud. Esa cadena se convierte en parte de la estructura, incluso aunque no esté guardada en la estructura.

## Apunadores a estructuras

Un programa en C puede declarar y usar apunadores a estructuras, de manera similar a los apunadores que apuntan a cualquier otro tipo de almacenamiento de dato. Los apunadores a estructuras se usan frecuentemente cuando se pasa una estructura como argumento a una función. Los apunadores a estructuras también se usan en un método de almacenamiento de datos muy poderoso, conocido como *listas encadenadas*. Ambos temas se tratan posteriormente en este capítulo.

Por ahora, veamos la manera en que el programa puede crear y usar apunadores a estructuras. Primero defina una estructura.

```
struct parte {
    int número;
    char nombre[10];
};
```

Ahora declare un apuntador al tipo parte.

```
struct parte *p_parte;
```

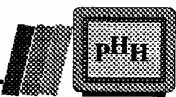
Recuerde, el operador de indirección (\*) en la declaración dice que p\_parte es un apuntador al tipo parte y no una instancia de tipo parte.

¿Puede ser inicializado ahora el apuntador? No, debido a que ha sido definida la estructura parte pero no ha sido declarada ninguna instancia de ella. Recuerde que es la declaración, y no la definición, la que reserva espacio de almacenamiento en memoria para los objetos de datos. Debido a que un apuntador necesita una dirección de memoria a la cual apuntar, se debe declarar una instancia de tipo parte antes de que cualquier cosa pueda apuntar a ella. Por lo tanto, aquí está la declaración:

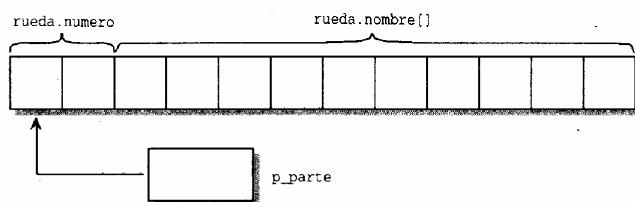
```
struct parte rueda;
```

Ahora se puede ejecutar la inicialización del apuntador.

```
p_parte = &rueda;
```



El enunciado anterior asigna la dirección de rueda a `p_parte`. (Recuerde del Día 9, “Apuntadores”, el operador de dirección de `(&)`.) La relación entre una estructura y un apuntador a la estructura se muestra en la figura 11.5.



**Figura 11.5.** Un apuntador a una estructura apunta al primer byte de la estructura.

Ahora que ya tiene un apuntador a la estructura `rueda`, ¿cómo lo usa? Un método usa el operador de indirección `(*)`. Recuerde del Día 9, “Apuntadores”, que si

`ptr`

es un apuntador a un objeto de dato, la expresión

`*ptr`

se refiere al objeto apuntado. Aplicando esto al ejemplo actual, se sabe que `p_parte` es un apuntador a la estructura `rueda` y, por lo tanto, `*p_parte` se refiere a `rueda`. Luego se aplica el operador de miembro de estructura `(.)` para accesar miembros individuales de `rueda`. Para asignar el valor de 100 a `rueda.numero` se podría escribir

`(*p_parte).numero = 100;`

`*p_parte` debe ser encerrado entre paréntesis, debido a que el operador `(.)` tiene una precedencia mayor que el operador `(*)`.

Un segundo método para accesar miembros de estructura usando un apuntador a la estructura es usar el *operador de membresía indirecta*, que consiste en los símbolos `->`, (un guion seguido por el símbolo de mayor que). (Observe que usándolos de esta manera, el C los trata como un solo operador y no como dos.) El símbolo es puesto entre el nombre del apuntador y el nombre del miembro. Para accesar el miembro `número` de `rueda` con el apuntador `p_parte`, se podría escribir

`p_parte->número`

Viendo otro ejemplo, si `str` es una estructura, `p_str` es un apuntador a `str` y `miem` es un miembro de `str`, se puede accesar a `str.miem` escribiendo

`p_str->miem`



## Estructuras

---

Por lo tanto, hay tres maneras de accesar un miembro de estructura: una, usando el nombre de la estructura, otra, usando un apuntador a la estructura con el operador de indirección (\*) y la tercera, usando un apuntador a la estructura con el operador de membresía indirecta (->). Si `p_str` es un apuntador a la estructura `str`, las siguientes expresiones son equivalentes:

```
str.miem  
(*p_str).miem  
p_str->miem
```

## Apuntadores y arreglos de estructuras

Ya ha visto que los arreglos de estructuras pueden ser una herramienta de programación muy poderosa y, de la misma forma, pueden ser los apuntadores a estructuras. Se pueden combinar los dos usando apuntadores para accesar estructuras que son elementos de arreglo.

Para ilustrarlo, aquí se encuentra una definición de estructura de un ejemplo anterior:

```
struct parte {  
    int numero;  
    char nombre[10];  
};
```

Después de que ha sido definida la estructura `parte`, se puede declarar un arreglo de tipo `parte`:

```
struct parte datos[100];
```

Luego se puede declarar un apuntador al tipo `parte`, e inicializarlo para que apunte a la primera estructura en el arreglo `datos`:

```
struct parte *p_parte;  
p_parte = &datos[0];
```

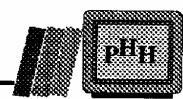
Recuerde que el nombre de un arreglo sin corchetes es un apuntador al primer elemento del arreglo, por lo que la segunda línea también podría haber sido escrita

```
p_parte = datos;
```

Ahora tenemos un arreglo de estructuras de tipo `parte`, y un apuntador al primer elemento de arreglo (esto es, la primera estructura en el arreglo). Se podría, por ejemplo, imprimir el contenido del primer elemento con el enunciado

```
printf("%d %s", p_parte->número, p_parte->nombre);
```

¿Qué pasa si se quiere imprimir todos los elementos del arreglo? Probablemente se usaría un ciclo `for`, imprimiendo un elemento de arreglo en cada iteración del ciclo. Para accesar los miembros usando notación de apuntadores, se debe cambiar el apuntador `p_parte` para que en cada iteración del ciclo apunte al siguiente elemento del arreglo (esto es, la siguiente estructura en el arreglo). ¿Cómo se hace esto?



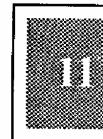
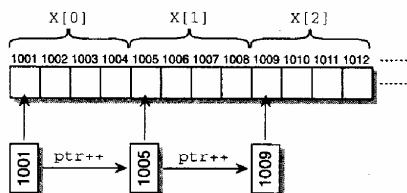
La aritmética de apuntadores del C viene en su auxilio. El operador unario de incremento (`++`) tiene un significado especial cuando es aplicado a un apuntador; significa “incremente el apuntador por el tamaño del objeto al que apunta”. Dicho de otra forma, si se tiene un apuntador `ptr` que apunta a un objeto de datos de tipo `obj`, el enunciado

```
ptr++;
```

tiene el mismo efecto que

```
ptr += sizeof(obj);
```

Este aspecto de la aritmética de apuntadores es particularmente relevante para los arreglos, de la siguiente manera: los elementos de arreglo se encuentran guardados secuencialmente en memoria. Si un apuntador apunta al elemento de arreglo `n`, al incrementar el apuntador con el operador (`++`) se logra que apunte al elemento `n + 1`. Esto se ilustra en la figura 11.6, que muestra un arreglo llamado `x[]` que consiste en elementos de cuatro bytes (por ejemplo, una estructura que contiene dos miembros tipo `int`, cada uno de dos bytes de largo). El apuntador `ptr` fue inicializado para que apuntara a `x[0]`. Cada vez que es incrementado, `ptr` apunta al siguiente elemento de arreglo.



**Figura 11.6.** Con cada incremento, un apuntador “avanza” al siguiente elemento de arreglo.

Lo que esto significa es que el programa puede avanzar por un arreglo de estructuras (o un arreglo de cualquier otro tipo de datos) incrementando un apuntador. Este tipo de notación es, por lo general, más fácil de usar y más concisa que usar subíndices de arreglo para ejecutar la misma tarea.

El programa del listado 11.4 le muestra la manera de hacer esto. A continuación se presenta la salida del programa.



#### Listado 11.4. Acceso de elementos de arreglos sucesivos incrementando un apuntador.

```
1: /* Demuestra el avance paso a paso por un arreglo de estructuras */
2: /* usando notación de apuntadores. */
3:
4: #include <stdio.h>
```

## Estructuras

### Listado 11.4. continuación

```
5:  
6: #define MAX 4  
7:  
8: /* Define una estructura, y luego declara e inicializa */  
9: /* un arreglo de cuatro estructuras */  
10:  
11: struct part {  
12:     int number;  
13:     char name[10];  
14: } data[MAX] = {1, "Smith",  
15:                 2, "Jones",  
16:                 3, "Adams",  
17:                 4, "Wilson"  
18: };  
19:  
20: /* Declara un apuntador al tipo part y una variable de contador. */  
21:  
22: struct part *p_part;  
23: int count;  
24:  
25: main()  
26: {  
27:     /* Inicializa el apuntador al primer elemento del arreglo. */  
28:  
29:     p_part = data;  
30:  
31:     /* Hace ciclo por el arreglo incrementando el apuntador */  
32:     /* en cada iteración. */  
33:  
34:     for (count = 0; count < MAX; count++)  
35:     {  
36:         printf("\nAt address %d: %d %s", p_part, p_part->number,  
37:                 p_part->name);  
38:         p_part++;  
39:     }  
40:  
41: }
```

### Salida

La salida del programa se muestra aquí:

```
At address 96: 1 Smith  
At address 108: 2 Jones  
At address 120: 3 Adams  
At address 132: 4 Wilson
```

### Análisis

Primero, este programa declara e inicializa un arreglo de estructuras en las líneas 11 a 18, llamado `data`. Un apuntador, llamado `p_part`, es definido luego para que apunte a la estructura `data`. La primera tarea de la función `main()` es hacer que el apuntador `p_part` apunte a la estructura `part` que fue declarada. Luego se imprimen todos los



elementos, usando un ciclo `for` que incrementa el apuntador al arreglo en cada iteración. El programa despliega la dirección de cada elemento.

Vea detalladamente las direcciones desplegadas. Los valores exactos pueden diferir en su sistema, pero tienen incrementos del mismo tamaño, el tamaño justo de la estructura `part` (la mayoría de los sistemas tendrán un incremento de 12). Esto ilustra claramente que al incrementar un apuntador se le incrementa por una cantidad igual al tamaño del objeto de datos al que apunta.

## Paso de estructuras como argumentos a funciones

De manera similar a otros tipos de datos, una estructura puede ser pasada como argumento a una función. El programa del listado 11.5 muestra la manera de hacerlo. Este programa es una modificación del programa del listado 11.2, usando una función para desplegar datos en la pantalla (a diferencia del listado 11.2 que usa enunciados que son parte de `main()`).



### Listado 11.5. Paso de una estructura como argumento a una función.

```
1: /* Demuestra el paso de una estructura a una función. */
2:
3: #include <stdio.h>
4:
5: /* Declara y define una estructura para guardar los datos. */
6:
7: struct data{
8:     float amount;
9:     char fname[30];
10:    char lname[30];
11: } rec;
12:
13: /* El prototipo de función. La función no tiene valor de retorno, */
14: /* y toma una estructura de tipo data como su único argumento. */
15:
16: void print_rec(struct data x);
17:
18: main()
19: {
20:     /* Recibe los datos del teclado */
21:
22:     printf("Enter the donor's first and last names,\n");
23:     printf("separated by a space: ");
24:     scanf("%s %s", rec.fname, rec.lname);
25:
26:     printf("\nEnter the donation amount: ");
27:     scanf("%f", &rec.amount);
28:
29:     /* Llama la función de desplegado. */
```





## Estructuras

### Listado 11.5. continuación

```
30:  
31:     print_rec( rec );  
32: }  
33:  
34: void print_rec(struct data x)  
35: {  
36:     printf("\nDonor %s %s gave $%.2f.", x.fname, x.lname,  
37:             x.amount);  
38: }
```



Enter the donor's first and last names,  
separated by a space: Bradley Jones  
Enter the donation amount: 1000.00  
Donor Bradley Jones gave \$1000.00.



Viendo la línea 16, se ve el prototipo de función para la función que va a recibir la estructura. Como se haría con cualquier otro tipo de dato que va a ser pasado, se necesita incluir los argumentos adecuados. En este caso es una estructura de tipo `data`. Esto es repetido en el encabezado para la función, en la línea 34. Cuando se llama a la función se necesita solamente pasar el nombre de instancia de la estructura, que en este caso es `rec` (línea 31). Esto es todo. Pasar una estructura a una función no es muy diferente que pasarle una simple variable.

También se puede pasar una estructura a una función pasando la dirección de la estructura (esto es, un apuntador a la estructura). En versiones antiguas del C, ésta era, de hecho, la única manera de pasar una estructura como argumento. Ya no es necesario, pero tal vez vea programas antiguos que usan todavía este método. Si se pasa un apuntador a una estructura como argumento, recuerde que debe usar el operador de membresía indirecta (`->`) para accesar miembros de estructura en la función.

### DEBE

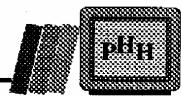
### NO DEBE

**NO DEBE** Confundir los arreglos con las estructuras!

**DEBE** Aprovechar la declaración de apuntadores a estructuras, especialmente cuando use arreglos de estructuras.

**NO DEBE** Olvidar que cuando se incrementa un apuntador, se mueve una distancia equivalente al tamaño del dato al que apunta. En el caso de un apuntador a una estructura, este es el tamaño de la estructura.

**DEBE** Usar el operador de membresía indirecta (`->`) cuando trabaje con un apuntador a una estructura.



## Uniones

- Las *uniones* son similares a las estructuras. Una unión es declarada y usada en la misma forma que una estructura. Una unión se diferencia de una estructura en que, en un momento dado, solamente puede ser usado uno de sus miembros. La razón de esto es simple. Todos los miembros de la unión ocupan la misma área de memoria. Están puestos uno encima de otro.

### Definición, declaración e inicialización de uniones

Las uniones son definidas y declaradas de manera parecida a las estructuras. La única diferencia en la declaración es que se usa la palabra clave `unión` en vez de `struct`. Para definir una unión simple de una variable `char` y una variable entera, se escribiría lo siguiente:

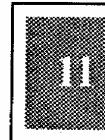
```
unión compartida {  
    char c;  
    int i;  
};
```

Esta unión, `compartida`, puede ser usada para crear instancias de una unión que puede guardar un valor de carácter `c` o un valor entero `i`. Esta es una condición `o`. A diferencia de una estructura, que podría contener ambos valores, la unión sólo puede contener un valor a la vez.

Una unión puede ser inicializada en su declaración. Debido a que solamente un miembro puede ser usado a la vez, solamente uno puede ser inicializado. Para evitar confusiones, sólo puede ser inicializado el primer miembro de la unión. Lo siguiente muestra una instancia de la unión `compartida`, siendo declarada e inicializada:

```
unión compartida variable_genérica = {'@'};
```

Observe que la unión `variable_genérica` fue inicializada de manera similar a como sería inicializado el primer miembro de una estructura.



### Acceso de miembros de la unión

Los miembros individuales de la unión pueden usarse de la misma forma en que se usan los miembros de la estructura, con el operador de miembro `(.)`. Hay una diferencia importante en el acceso a miembros de unión. Solamente un miembro debe ser accesado a la vez. Debido a que una unión guarda a sus miembros uno encima de otro, es importante accesar solamente un miembro a la vez. Esto se muestra mejor con un ejemplo.



## Estructuras

Captura

### Listado 11.6. Un ejemplo del uso erróneo de uniones.

```
1:  /* Ejemplo del uso de más de un miembro de unión a la vez */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      union shared_tag {
7:          char   c;
8:          int    i;
9:          long   l;
10:         float  f;
11:         double d;
12:     } shared;
13:
14:     shared.c = '$';
15:
16:     printf("\nchar c = %c", shared.c);
17:     printf("\nint i = %d", shared.i);
18:     printf("\nlong l = %ld", shared.l);
19:     printf("\nfloat f = %f", shared.f);
20:     printf("\ndouble d = %f", shared.d);
21:
22:     shared.d = 123456789.8765;
23:
24:     printf("\n\nchar c = %c", shared.c);
25:     printf("\nint i = %d", shared.i);
26:     printf("\nlong l = %ld", shared.l);
27:     printf("\nfloat f = %f", shared.f);
28:     printf("\ndouble d = %f", shared.d);
29: }
```



```
char c = $
int i = 4900
long l = 437785380
float f = 0.000000
double d = 0.000000

char c = 7
int i = -30409
long l = 1468107063
float f = 284852666499072.000000
double d = 123456789.876500
```



En este listado se puede ver que una unión, llamada shared, fue definida y declarada en las líneas 6 a 12. shared contiene 5 miembros, cada uno de diferente tipo. Las líneas 14 y 22 inicializan miembros individuales de shared. Las líneas 16 a 20 y 24 a 28 presentan luego los valores de cada miembro usando enunciados printf().

**Sintaxis**

Observe que a excepción de `char c = $` y `double d = 123456789.876500`, la salida no puede ser la misma en su computadora. Debido a que la variable de carácter, `c`, fue inicializada en la línea 14, es el único valor que debe ser usado hasta que un miembro diferente sea inicializado. Los resultados de la impresión de las otras variables miembro de la unión (`i`, `l`, `f` y `d`) pueden ser impredecibles (líneas 16 a 20). La línea 22 pone un valor en la variable `double`, `d`. Observe que la impresión de las variables nuevamente es impredecible para todas, a excepción de `d`. El valor dado para `c` en la línea 14 se ha perdido, debido a que ha sido sobreescrito cuando se dio el valor de `d` en la línea 22. Esta es una evidencia de que todos los miembros ocupan el mismo espacio.

## **La palabra clave *unión***

```
unión etiqueta {
    miembros_de_la_unión
    /* aquí pueden ir enunciados adicionales */
}instancia;
```

La palabra clave `unión` es usada para declarar uniones. Una unión es una colección de una o más variables (`miembros_de_la_unión`) que han sido agrupados bajo un solo nombre. Además, cada uno de estos miembros de la unión ocupa la misma área de memoria.

La palabra clave `unión` identifica el inicio de una definición de unión. Es seguida por una etiqueta, que es el nombre dado a la unión. A continuación de la etiqueta se encuentran los miembros de la unión encerrados en llaves. Una `instancia`, la declaración actual de la unión, también puede ser definida. Si se define la estructura sin la instancia es simplemente una plantilla, que puede ser usada posteriormente en un programa para declarar estructuras. A continuación se presenta un formato de plantilla:

```
unión etiqueta {
    miembros_de_la_unión
    /* aquí pueden ir enunciados adicionales */
};
```

Para usar la plantilla se podría usar el siguiente formato:

```
unión etiqueta instancia;
```

Para usar este formato se debe haber definido previamente una unión con la etiqueta dada.

### **Ejemplo 1**

```
/* Declara una plantilla de unión llamada ??? */
unión etiqueta {
    int num;
    char carácter;
}
/* Usa la plantilla de unión */
union etiqueta variables_mezcladas;
```



## Estructuras

### Ejemplo 2

```
/* Declara al mismo tiempo una unión y una instancia */
unión etiqueta_tipo_genérico {
    char c;
    int i;
    float f;
    double d;
} genérico;
```

### Ejemplo 3

```
/* Inicializa una unión. */
unión etiq_fecha {
    char fecha_completa[9];
    struct etiq_fecha_en_partes {
        char día[2];
        char valor_corte1;
        char mes[2];
        char valor_corte2;
        char año[2];
    } fecha_en_partes;
} fecha = {"01/01/97"};
```

El listado 11.7 muestra un uso más práctico de una unión. Este listado muestra un uso simplista de una unión. Aunque es simplista, es uno de los más comunes de las uniones.

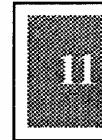
### Captura

### Listado 11.7. Un uso práctico de una unión.

```
1:  /* Ejemplo de un uso típico de una unión */
2:
3:  #include <stdio.h>
4:
5:  #define CHARACTER  'C'
6:  #define INTEGER   'I'
7:  #define FLOAT     'F'
8:
9:  struct generic_tag{
10:    char type;
11:    union shared_tag {
12:        char c;
13:        int i;
14:        float f;
15:    } shared;
16: };
17:
18: void print_function( struct generic_tag generic );
19:
20: main()
21: {
```



```
22:     struct generic_tag var;
23:
24:     var.type = CHARACTER;
25:     var.shared.c = '$';
26:     print_function( var );
27:
28:     var.type = FLOAT;
29:     var.shared.f = 12345.67890;
30:     print_function( var );
31:
32:     var.type = _x_;
33:     var.shared.i = 111;
34:     print_function( var );
35: }
36:
37: void print_function( struct generic_tag generic )
38: {
39:     printf("\n\nThe generic value is...\"");
40:     switch( generic.type )
41:     {
42:         case CHARACTER: printf("%c", generic.shared.c);
43:                         break;
44:         case INTEGER:   printf("%d", generic.shared.i);
45:                         break;
46:         case FLOAT:    printf("%f", generic.shared.f);
47:                         break;
48:         default:       printf("an unknown type: %c",
49:                               generic.type);
50:             break;
51:     }
52: }
```



The generic value is...\$  
The generic value is...12345.678711

The generic value is...an unknown type: x



Este programa es una versión muy simplista de lo que podría ser hecho con una unión. Este programa proporciona una manera de guardar varios tipos de datos en un solo espacio de almacenamiento. La estructura `generic_tag` le permite guardar un carácter o un entero o un número de punto flotante dentro de la misma área. Esta área es una unión llamada `shared`, que funciona de manera similar a los ejemplos del listado 11.6. Observe que la estructura `generic_tag` también añade un campo adicional llamado `type`. Este campo es usado para guardar información sobre el tipo de variable contenida en `shared.type` ayuda a prevenir que `shared` sea usado en forma equivocada y, por lo tanto, ayuda a evitar datos erróneos, como se presentaron en el listado 11.6.



Una vista formal del programa muestra que las líneas 5, 6 y 7 definen las constantes CHARACTER, INTEGER y FLOAT. Estas se usan posteriormente en el programa para hacer más legible el listado. Las líneas 9 a 16 definen una estructura, generic\_tag, que será usada posteriormente. La línea 18 presenta un prototipo para la función print\_function(). La estructura var es declarada en la línea 22, e inicializada por primera vez en las líneas 24 y 25 para que guarde un valor de carácter. Una llamada a la función print\_function(), en la línea 26, permite que se impriman los valores. Las líneas 28 a 30 y 32 a 40 repiten este proceso con otros valores.

La función print\_function() es la parte medular de este listado. Aunque esta función es usada para imprimir el valor de una variable de generic\_tag, se podría haber usado una función similar para inicializarla. print\_function() evaluará la variable type para imprimir un enunciado con el tipo de variable adecuado. Esto impide el que se obtengan datos erróneos, como los del listado 11.6.

DEBE	NO DEBE
NO DEBE Tratar de inicializar más que el primer miembro de la unión.	
DEBE Recordar qué miembro de la unión se está usando. Si se llena un miembro de un tipo y se trata de usar de otro tipo diferente, se pueden obtener resultados impredecibles.	
NO DEBE Olvidar que el tamaño de una unión es igual al de su miembro más grande.	
DEBE Observar que las uniones son un tema avanzado del C.	

## Listas encadenadas

El último tema de este capítulo es una breve introducción a las listas encadenadas. El término *listas encadenadas* se refiere a una clase general de métodos de almacenamiento de datos, en la cual cada concepto de información está encadenado a uno o más conceptos diferentes, usando apuntadores.

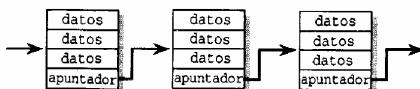
Hay varias clases de listas encadenadas, incluyendo listas con encadenamiento simple, listas con encadenamiento doble y árboles binarios. Cada tipo es adecuado para determinadas tareas de almacenamiento de datos. Lo que tienen en común es que los encadenamientos entre conceptos de datos están definidos por información que se encuentra en los mismos conceptos. En esto se distinguen de los arreglos, donde los encadenamientos entre los conceptos de datos son resultado de la estructura del arreglo. Este capítulo explica el tipo más simple de lista encadenada, la lista con encadenamiento sencillo (a la cual se hace referencia simplemente como lista encadenada).

## La organización de una lista encadenada

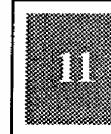
Para ilustrar la manera en que son construidas las listas encadenadas comenzemos con una definición de estructura simple:

```
struct datos {
    char nombre[10];
    struct datos *siguiente;
};
```

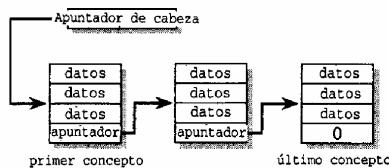
¿No ve algo raro en la definición? El segundo miembro del tipo `datos` es un apuntador al tipo `datos`. En otras palabras, la estructura incluye un apuntador a su propio tipo. Esto significa que una instancia de tipo `datos` puede apuntar a otra instancia del mismo tipo. Esta es la manera en que se crean los encadenamientos en una lista encadenada. Cada estructura apunta a la siguiente estructura de la lista. Esto se ilustra en la figura 11.7.



**Figura 11.7.** La manera en que los encadenamientos se forman en una lista encadenada.



Cada lista debe tener un inicio y un final. ¿Cómo son representados éstos en C? El inicio de la lista está marcado por el *apuntador de cabeza*, que apunta a la primera estructura de la lista. El apuntador de cabeza no es una estructura, sino simplemente un apuntador al tipo de dato que forma la lista. El final de la lista está marcado por una estructura que tiene un valor de apuntador de NULL. Debido a que todas las demás estructuras de la lista contienen un apuntador que no es NULL, que apunta al siguiente concepto de la lista, un valor de apuntador de NULL es una manera inequívoca para indicar el final de la lista. La estructura de una lista encadenada, con su apuntador de cabeza y su último elemento, se muestra en la figura 11.8.



**Figura 11.8.** El comienzo de una lista encadenada está indicado por un apuntador de cabeza, y el fin de la lista está indicado por un valor de apuntador de NULL.

Pareciera que la lista encadenada no presenta ninguna ventaja real sobre un arreglo de estructuras. Sin embargo, hay varias ventajas significativas. Una tiene que ver con la inserción y borrado de elementos en una lista ordenada (que es una tarea común de



programación). Imagine que el programa mantiene una lista de 1,000 conceptos de datos en orden alfabético. Para mantener el orden los nuevos conceptos deben ser insertados en la posición adecuada de la lista: Baez entre Arreola y Cervantes, por ejemplo.

Si se está usando un arreglo, se debe hacer un hueco en la posición adecuada antes de poder insertar los nuevos datos. Digamos, por ejemplo, que Arreola está en la posición 5 y Cervantes en la posición 6. Para insertar a Baez en la posición adecuada (posición 6), Cervantes y todos los elementos de arreglo superiores deben ser movidos un espacio hacia arriba. ¡Esto requiere una gran cantidad de procesamiento! De manera similar, para borrar un concepto de en medio del arreglo, todos los elementos superiores necesitan ser movidos hacia abajo para llenar el hueco.

Las listas encadenadas facilitan la inserción y el borrado de conceptos de datos. Todo lo que se requiere es un poco de manipulación simple de apuntadores. No se necesita, de hecho, mover los datos. Continuando con el ejemplo anterior, se necesita insertar a Baez entre Arreola y Cervantes. En una lista encadenada, inicialmente el registro de Arreola apunta al registro de Cervantes. Para insertar el nuevo concepto, Baez, todo lo que se necesita es hacer que el registro de Arreola apunte al registro de Baez y que el registro de Baez apunte al registro de Cervantes. Este procedimiento es ilustrado en la figura 11.9. (El procedimiento para borrar un concepto es igualmente simple, pero, imaginarse cómo, queda como un ejercicio para usted!).

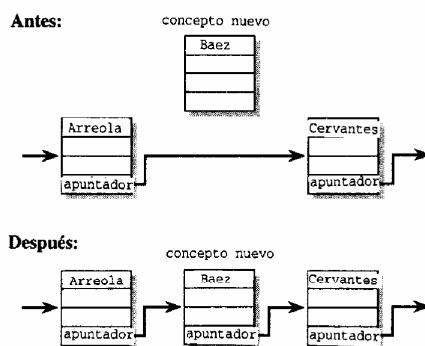


Figura 11.9. Inserción de un nuevo concepto en una lista encadenada ordenada.

Otra ventaja de las listas encadenadas se refiere al espacio de almacenamiento. Para usar un arreglo, su tamaño (cantidad de elementos) debe ser definido cuando el programa es compilado. Si, durante la ejecución, el programa se queda sin espacio de memoria, no hay nada que pueda hacerse. Sin embargo, con una lista encadenada el programa puede asignar espacio de almacenamiento para las estructuras conforme lo necesite, llegando hasta el límite impuesto por el hardware. Esto se hace con la función del C `malloc()`, tratada a continuación.



## La función `malloc()`

Ya aprendió acerca de `malloc()`, la función del C para asignación de memoria, en el Día 10, “Caracteres y cadenas”. El uso de `malloc()` para asignar espacio para una estructura es esencialmente idéntico al uso para asignar espacio para tipo `char`. Si se ha definido un tipo de estructura llamado `datos`, primero se declara un apuntador al tipo

```
struct datos *ptr;  
y luego se llama a malloc()  
ptr = malloc(sizeof(struct datos));
```

Si `malloc()` funciona satisfactoriamente, asigna un bloque de memoria del tamaño adecuado y regresa un apuntador a él. Si `malloc()` falla, es decir, que no hay suficiente memoria disponible, regresa `NULL` (y un programa real debe hacer prueba de ello).

El valor regresado por `malloc()` ha sido asignado a `ptr`, un apuntador al tipo `datos`. Usando este apuntador y el operador de membresía indirecta (`->`) se puede accesar la memoria asignada de acuerdo con los miembros de `datos`. Por lo tanto, si el tipo `datos` tiene un miembro de tipo `float` llamado `valor`, se podría escribir

```
ptr ->valor = 1.205;
```

Las estructuras asignadas con `malloc()` no tienen nombre *por sí mismas*, por lo que nunca puede ser usado el operador de membresía `(.)` para accesar sus miembros. Se debe utilizar siempre un apuntador y el operador de membresía indirecta (`->`).



## Implementación de una lista encadenada

La información dada en este capítulo debe serle suficiente para que implemente un programa de lista encadenada. Este podría ser un ejercicio de programación excelente. Si usted tiene éxito en la escritura de un programa que usa listas encadenadas para guardar datos, se puede sentir muy satisfecho, ¡ya que va por buen camino para llegar a ser un programador de C eficiente!

## `typedef` y las estructuras

Se puede usar la palabra clave `typedef` para crear un sinónimo para un tipo de estructura o unión. Por ejemplo, los enunciados

```
typedef struct {  
    int x;  
    int y;  
} coord;
```



## Estructuras

---

definen coord como sinónimo para la estructura indicada. Luego, se pueden declarar instancias de esta estructura usando el identificador coord.

```
coord arribaizquierda, abajoderecha;
```

Observe que `typedef` no es lo mismo que una etiqueta de estructura, como se describió anteriormente en este capítulo. Si se escribe

```
struct coord {  
    int x;  
    int y;  
};
```

el identificador coord es una etiqueta para la estructura. Se puede usar la etiqueta para declarar instancias de la estructura, pero, a diferencia del `typedef`, se debe incluir la palabra clave `struct`:

```
struct coord arribaizquierda, abajoderecha;
```

El que se use `typedef` o la etiqueta de estructura para declarar a las estructuras casi no tiene diferencia práctica. El usar `typedef` da como resultado un código ligeramente más conciso, debido a que no se necesita usar la palabra clave `struct`. Por otro lado, usar una etiqueta y tener explícita la palabra clave `struct` hace más claro que se está declarando una estructura.

## Resumen

Este capítulo le mostró la manera de usar estructuras, un tipo de dato que el programador diseña para satisfacer las necesidades de su programa. Una estructura puede contener cualquiera de los tipos de datos del C, incluyendo otras estructuras, apuntadores y arreglos. Cada concepto de datos dentro de una estructura, llamado un miembro, es accesado usando el operador de miembro de estructura (`.`) entre el nombre de la estructura y el nombre del miembro. Las estructuras pueden usarse individualmente y también pueden usarse en arreglos.

Los apuntadores a estructuras abren mayores posibilidades. Se pueden usar apuntadores para crear listas encadenadas de estructuras, en las cuales cada elemento de la lista está encadenado al siguiente por medio de un apuntador. Combinando listas encadenadas con la función `malloc()`, un programa puede asignar dinámicamente espacio de almacenamiento conforme lo necesite.

Las uniones fueron presentadas como similares a las estructuras. La principal diferencia entre una unión y una estructura es que la unión guarda a todos sus miembros en la misma área. Esto significa que sólo un miembro individual de la unión puede ser usado en un momento dado.



## Preguntas y respuestas

1. ¿Tiene algún objeto declarar una estructura sin ninguna instancia?

Se mostraron dos maneras de declarar estructuras. La primera fue declarar un cuerpo de estructura, una etiqueta y una instancia al mismo tiempo. La segunda fue declarar un cuerpo de estructura y una etiqueta sin una instancia.

Posteriormente puede ser declarada una instancia usando la palabra clave struct, la etiqueta y un nombre para la instancia. Es una práctica común de programación usar el segundo método. Muchos programadores declararán el cuerpo de la estructura y su etiqueta sin ninguna instancia. Las instancias serán declaradas posteriormente en el programa. En el siguiente capítulo se describe el alcance de las variables. El alcance se aplica a la instancia, pero no a la etiqueta o al cuerpo de la estructura.

2. ¿Qué es más común: usar un `typedef` o una etiqueta de estructura?

Muchos programadores usan `typedef` para hacer que su código sea más fácil de leer, mas, sin embargo, tiene poca diferencia práctica. Se pueden comprar muchas bibliotecas que contienen funciones. Estos productos adicionales tienen muchos `typedef` para hacer único al producto. Esto es especialmente cierto sobre los productos de bases de datos adicionales.



3. ¿Puedo simplemente asignar una estructura a otra con el operador de asignación?

¡Sí y no! Las versiones más recientes de los compiladores de C le permitirán asignar una estructura a otra, aunque puede ser que las versiones antiguas no lo permitan. ¡En las versiones antiguas del C tal vez necesite asignar cada miembro de la estructura individualmente! Esto también es cierto para las uniones.

4. ¿Qué tan grande es una unión?

Debido a que cada uno de los miembros en una unión está guardado en la misma posición de memoria, la cantidad de espacio requerido para guardar la unión es igual al de su miembro más grande.

## Taller

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

**Cuestionario**

1. ¿Qué tan diferente es una estructura de un arreglo?
2. ¿Cuál es el operador de miembro de estructura y para qué sirve?
3. ¿Qué palabra clave se usa en C para crear una estructura?
4. ¿Cuál es la diferencia entre una etiqueta de estructura y una instancia de estructura?
5. ¿Qué hace el siguiente fragmento de código?

```
struct address {
    char name[31];
    char add1[31];
    char add2[31];
    char city[11];
    char state[3];
    char zip[11];
} myaddress = { "Bradley Jones",
                "RTSoftware",
                "P.O. Box 1213",
                "Carmel", "IN", "46032-1213";
```

6. Suponga que ha declarado un arreglo de estructuras y que ptr es un apuntador al primer elemento del arreglo (esto es, la primera estructura del arreglo). ¿Cómo podría cambiar ptr para que apuntara al segundo elemento del arreglo?
7. ¿Cuál es el único miembro necesario en un tipo de estructura que es usado en una lista encadenada?
8. ¿Cuáles son las dos ventajas de usar una lista encadenada en vez de un arreglo?
9. ¿Cuál es el argumento y el valor de retorno de malloc()?
10. ¿Con qué método se borra un concepto de una lista encadenada?

**Ejercicios**

1. Escriba el código que define a una estructura llamada tiempo que contiene tres miembros int.
2. Escriba el código que: a) define una estructura llamada datos, que contiene un miembro tipo int y dos miembros tipo float y b) declara una instancia de tipo datos llamada info.

- 
- 
3. Continuando con la pregunta 2, ¿cómo asignaría el valor de 100 al miembro int de la estructura info?
  4. Escriba el código que declara e inicializa un apuntador a info.
  5. Continuando con la pregunta 4, muestre dos maneras de usar notación de apuntadores para asignar el valor 5.5 al primer miembro float de info.
  6. Escriba la definición para un tipo de estructura llamado datos, que pueda guardar una sola cadena de hasta 20 caracteres y que pueda ser usado en una lista encadenada.
  7. Cree una estructura que contenga cinco cadenas, dirección1, dirección2, ciudad, estado y código\_postal. Cree un typedef, llamado REGISTRO, que pueda ser usado para crear instancias de esta estructura.
  8. Usando el typedef del ejercicio 7, asigne e inicialice un elemento llamado midirección.
  9. **BUSQUEDA DE ERRORES:** ¿Qué hay de erróneo en el siguiente fragmento de código?

```
struct {
    char zodiac_sign[21];
    int month;
} sign = "Leo", 8;
```

10. **BUSQUEDA DE ERRORES:** ¿Qué hay de erróneo en el siguiente fragmento de código?

```
/* setting up a union */
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW", 1000 };
```

11. **BUSQUEDA DE ERRORES:** ¿Qué hay de erróneo en el siguiente fragmento de código?

```
/* a structure to be used in a linked list */
struct data{
    char firstname[10];
    char lastname[10];
    char middlename[10];
    char *next_name;
}
```



