

PROGRAMACIÓN ESTRUCTURADA EN

C



PRENTICE HALL

PROGRAMACIÓN ESTRUCTURADA EN C

James L. Antonakos
Kenneth C. Mansfield JR.
Broome Community College

Traducción:

Jesús Carretero Pérez
Félix García Carballeira
Fernando Pérez Costoya
Facultad de Informática
Universidad Politécnica de Madrid

Revisión Técnica:

Luis Joyanes Aguilar
Facultad de Informática
Universidad Pontificia de Salamanca en Madrid



PRENTICE HALL

Madrid • Upper Saddle River • Londres • México • Nueva Delhi • Rio de Janeiro
Santafé de Bogotá • Singapur • Sydney • Tokio • Toronto

CONSULTORES EDITORIALES:

SEBASTIÁN DORMIDO BENCOMO

Departamento de Informática y Automática

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

LUIS JOYANES AGUILAR

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería del Software

UNIVERSIDAD PONTIFICIA DE SALAMANCA en Madrid

Resumen del contenido

1.	FUNDAMENTOS DE C	1
2.	PROGRAMACIÓN ESTRUCTURADA	51
3.	OPERACIONES SOBRE DATOS Y TOMA DE DECISIONES	103
4.	BUCLES Y RECURSIVIDAD	171
5.	PUNTEROS, ÁMBITO Y CLASES DE VARIABLES	215
6.	CADENAS DE CARACTERES	259
7.	VECTORES Y MATRICES NUMÉRICAS	309
8.	ESTRUCTURAS DE DATOS	365
9.	ENTRADA/SALIDA DE DISCO	443
10.	ASPECTOS AVANZADOS	483
	APÉNDICES	525
	SOLUCIONES	533
	ÍNDICE	563

ANTONAKOS J.L. y MANSFIELD JR., K.C.
Programación estructurada en C
PRENTICE HALL IBERIA. Madrid, 1997

ISBN: 84-89660-23-9

MATERIA:

Informática 681.3

Formato 195 x 250 mm

Páginas 584

J.L. ANTONAKOS y K.C. MANSFIELD JR.
Programación estructurada en C

No está permitida la reproducción total o parcial de esta obra
ni su tratamiento o transmisión por cualquier medio o método
sin autorización escrita de la Editorial.



DERECHOS RESERVADOS

© 1997 respecto a la primera edición en español por:
PRENTICE HALL International (UK) Ltd.
Campus 400, Maylands Avenue
Hemel Hempstead
Hertfordshire, HP2 7EZ
Simon & Schuster International Group

ISBN: 84-89660-23-9

Depósito legal: M. 11.281-1998

1.^a reimpresión, 1998

Traducido de:

APPLICATION PROGRAMMING IN STRUCTURED C.
PRENTICE HALL, INC
Simon & Schuster International Group
A Viacom Company
© MCMXCVI
ISBN 0-13-520487-9

Edición en español:

Editor: Andrés Otero
Diseño de cubierta: DIGRAF
Composición: Dayo, S. L.
Impreso por: Fareso, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicas

Resumen del contenido

1.	FUNDAMENTOS DE C	1
2.	PROGRAMACIÓN ESTRUCTURADA	51
3.	OPERACIONES SOBRE DATOS Y TOMA DE DECISIONES	103
4.	BUCLES Y RECURSIVIDAD	171
5.	PUNTEROS, ÁMBITO Y CLASES DE VARIABLES	215
6.	CADENAS DE CARACTERES	259
7.	VECTORES Y MATRICES NUMÉRICAS	309
8.	ESTRUCTURAS DE DATOS	365
9.	ENTRADA/SALIDA DE DISCO	443
10.	ASPECTOS AVANZADOS	483
	APÉNDICES	525
	SOLUCIONES	533
	ÍNDICE	563

Contenido

1.	FUNDAMENTOS DE C	1
1.1.	El entorno de C	2
1.2.	¿Por qué C?	5
1.3.	Estructura de un programa	7
1.4.	Elementos de C	13
1.5.	La función <code>printf()</code>	15
1.6.	Identificación de cosas	18
1.7.	Declaración de cosas	21
1.8.	Introducción a los operadores de C	24
1.9.	Más <code>printf()</code>	29
1.10.	Leyendo la entrada del usuario	30
1.11.	Implementación y depuración de programas: errores de programación frecuentes	34
1.12.	Programa de aplicación: conversión de temperatura	38
1.13.	Programas de aplicación adicionales	44
1.14.	Programación en ANSI C	46
2.	PROGRAMACIÓN ESTRUCTURADA	51
2.1.	Conceptos de bloque de programa	52
2.2.	Uso de funciones	57
2.3.	Dentro de una función C	64
2.4.	Uso de funciones	69

2.5.	Uso de la directiva <code>#define</code>	79
2.6.	Implementación y depuración de programas: creando sus propios archivos de cabecera	83
2.7.	Programa de aplicación: circuito RL en serie	89
3.	OPERACIONES SOBRE DATOS Y TOMA DE DECISIONES	103
3.1.	Operadores de comparación	104
3.2.	La bifurcación abierta	108
3.3.	La bifurcación cerrada	116
3.4.	Operaciones booleanas binarias	124
3.5.	Operaciones lógicas	130
3.6.	Conversión de tipos	137
3.7.	La sentencia <code>switch</code>	139
3.8.	Más sobre la sentencia <code>switch</code> y el operador condicional	147
3.9.	Depuración y desarrollo de programas	152
3.10.	Programa de aplicación: Reparación de un robot	154
3.11.	Programas de aplicación adicionales	160
4.	BUCLES Y RECURSIVIDAD	171
4.1.	El bucle <code>for</code>	172
4.2.	Bucle <code>while</code>	179
4.3.	El bucle <code>do while</code>	183
4.4.	Bucles anidados	188
4.5.	Implementación y depuración de programas	194
4.6.	Recursividad	198
4.7.	Programa de aplicación: máquina expendedora	202
4.8.	Programas de aplicación adicionales	205
5.	PUNTEROS, ÁMBITO Y CLASES DE VARIABLES	215
5.1.	Organización de la memoria	216
5.2.	Cómo se utiliza la memoria	220
5.3.	Punteros	228
5.4.	Paso de variables	234
5.5.	El ámbito de las variables	242
5.6.	Clases de variables	246
5.7.	Depuración e implementación de programas	249
5.8.	Programas de aplicación	251

6.	CADENAS DE CARACTERES	259
6.1.	Caracteres y cadenas de caracteres	260
6.2.	Inicio de cadenas de caracteres	265
6.3.	Paso de cadenas de caracteres entre funciones	267
6.4.	Trabajo con elementos de cadenas de caracteres	270
6.5.	Funciones para manejar cadenas de caracteres	275
6.6.	Ordenación de cadenas de caracteres	284
6.7.	Programa de aplicación: formateador de texto	288
6.8.	Programas de aplicación adicionales	295
7.	VECTORES Y MATRICES NUMÉRICAS	309
7.1.	Vectores numéricos	310
7.2.	Introducción a las aplicaciones con vectores numéricos	318
7.3.	Ordenación de vectores numéricos	323
7.4.	Matrices numéricas	343
7.5.	Programas de aplicación	351
8.	ESTRUCTURAS DE DATOS	365
8.1.	Tipos enumerados	366
8.2.	Dar nombre a sus propios tipos de datos	371
8.3.	Introducción a las estructuras de datos	376
8.4.	Más detalles acerca de las estructuras de datos	381
8.5.	La union y vectores de estructuras	389
8.6.	Formas de representar estructuras	398
8.7.	Estructuras de datos avanzadas	400
8.8.	Programa de aplicación: MiniMicro	421
8.9.	Programas de aplicación adicionales	425
9.	ENTRADA/SALIDA DE DISCO	443
9.1.	Entrada/salida de disco	444
9.2.	Más sobre la E/S de disco	448
9.3.	Flujos de datos, punteros a archivos y argumentos de la línea de mandatos	462
9.4.	Programa de aplicación: Una base de datos de piezas	470
9.5.	Programas de aplicación adicionales	476

10. ASPECTOS AVANZADOS	483
10.1. Punteros a funciones	484
10.2. Funciones como argumentos	489
10.3. Funciones con número variable de argumentos	492
10.4. Aplicaciones modulares	497
10.5. La utilidad <code>make</code>	506
10.6. Programas de aplicación adicional	513
 APÉNDICES	 525
A. Manual de consulta de C	525
B. Funciones matemáticas estándar de ANSI C	528
C. Juego de caracteres ASCII	531
 SOLUCIONES	 533

Prólogo

Introducción

El lenguaje de programación C se está abriendo paso cada día en un número creciente de currícula de escuelas y facultades técnicas. La necesidad de un libro de texto que enseñe los conceptos fundamentales de programación necesarios para los estudiantes de hoy es grande. Todos los estudiantes de tecnología que usan C descubren que es un lenguaje potente y, sin embargo, fácil de aprender. La habilidad para controlar un computador a nivel hardware usando un lenguaje de alto nivel como C es importante y necesaria, especialmente en la sociedad actual, donde cada vez más tareas son delegadas a los computadores.

El propósito de este libro es describir el lenguaje de programación C a personas sin conocimientos de programación, usando ejemplos. Las cosas básicas que hace un programador (bucles, cálculos, formatos, entrada/salida) son estudiadas en detalle. Casi 200 programas ejecutables y probados son usados para ilustrar los conceptos necesarios de C. Este libro es adecuado para estudiantes de cualquier disciplina tecnológica, particularmente aquéllos que estudian tecnología de los computadores y electrónica.

Contenido de los capítulos

El Capítulo 1 analiza los conceptos fundamentales del lenguaje de programación C. Se describe todo lo relativo a programación estructurada, variables, operadores, funciones y entrada/salida (E/S).

El Capítulo 2 resalta la forma correcta de escribir un programa estructurado en lenguaje C. Los detalles operativos de las funciones son expandidos.

El Capítulo 3 informa acerca del uso de las distintas sentencias de control, tales como `if`, `if...else` y `switch`, y acerca de las operaciones lógicas.

El Capítulo 4 muestra cómo se escriben los bucles en C. Los ejemplos incluyen el uso de `for`, `while` y `do...while`, así como de bucles anidados.

El Capítulo 5 analiza la relación entre variable, punteros a variables y el espacio de memoria necesitado por variables de ciertos tipos. También se analiza el uso de variables locales y globales.

El Capítulo 6 ilustra el manejo de cadenas de caracteres en C. Se explica cómo iniciar cadenas de caracteres, su paso como parámetros de función y las funciones incluidas en C para manejar tiras de caracteres.

El Capítulo 7 estudia los detalles de las matrices numéricas (arrays numéricos) de una o varias dimensiones. Incluye varias aplicaciones para ordenar matrices, para mostrar cómo se accede a los componentes de una matriz, cómo se parten y cómo se pueden pasar entre funciones.

El Capítulo 8 muestra las técnicas básicas de estructuración de datos usando `enum` y `typedef`. También se muestran estructuras avanzadas de datos como las listas enlazadas y los árboles binarios.

El Capítulo 9 explica los detalles de la entrada/salida sobre archivos. Se muestra cómo crear, escribir y leer archivos secuenciales y de acceso directo.

El Capítulo 10 presenta varios temas avanzados, incluyendo el uso de funciones con número variable de parámetros, el paso de funciones como parámetros, la utilización de lenguaje máquina dentro de un programa C y la utilidad `MAKE`.

El disquete adjunto

El disquete adjunto contiene los archivos con el código fuente en lenguaje C de todos los programas presentados en el libro. Existe un subdirectorio por cada capítulo del libro. Cada programa ha sido compilado y ejecutado en tres compiladores distintos para garantizar su corrección.

El directorio raíz contiene un archivo ejecutable llamado `LEAME.COM` que explica el contenido del disquete en detalle.

Agradecimientos

Nos gustaría mostrar nuestro agradecimiento a todos los estudiantes que contribuyeron con sus comentarios durante la preparación del manuscrito.

También nos gustaría mostrar nuestro agradecimiento a nuestro editor Charles Stewart, por sus consejos y su paciencia, y a su ayudante Meryl Chertoff, por sus respuestas a nuestras numerosas preguntas.

James L. Antonakos
antonakos-j@sunybroome.edu

Kenneth C. Mansfield Jr.
mansfield-k@sunybroome.edu

Prólogo a la edición en español

Es un placer para nosotros poder presentar este texto a las personas interesadas en el lenguaje de programación C. Antes de decidimos por su traducción, estudiamos otros libros de programación en lenguaje C, eligiendo este debido a su orientación a programadores noveles y con necesidad de usar el lenguaje C en aplicaciones de tecnología. Otra razón importante para la elección de este texto fue el uso de la metodología de programación estructurada para enseñar este lenguaje. Este aspecto es muy importante, porque permite inculcar en los lectores buenos hábitos de programación, especialmente necesarios cuando se usa el lenguaje C, porque debido a la flexibilidad de este lenguaje, existe una tendencia general a crear programas poco estructurados y difíciles de leer.

A pesar de que ya teníamos experiencia en trabajos de traducción y en programación con el lenguaje C, la traducción de este texto ha sido trabajosa porque hemos realizado una amplia reestructuración del mismo, si bien hemos respetado todos los aspectos fundamentales del texto original. En primer lugar, hemos traducido todos los programas del libro, de forma que sean más fáciles de leer y entender. Además, se han eliminado del texto algunos programas redundantes o poco claros y se han fusionado otros que resultaban demasiado repetitivos con el objetivo de hacer más ligero el texto. En cuanto al contenido de los capítulos, se ha respetado cuidadosamente el de todos ellos, excepto en el caso del Capítulo 10, que está dedicado a los temas avanzados, y que ha sufrido una profunda remodelación. La edición española incluye en este capítulo aspectos del lenguaje C que no estaban contemplados en el texto original como son: *punteros a funciones, funciones como parámetros*, etc. El resultado global es un texto más ligero y con menos dependencias de los compiladores para computadores personales, dependencias estas que eran muy patentes en el texto original, y especialmente, en este capítulo.

El disquete que se adjunta contiene los archivos con el código fuente en lenguaje C de todos los programas presentados en el libro. Existe un subdirectorío por cada capítulo del libro. Cada programa ha sido compilado y ejecutado en computadores personales, con el sistema operativo MS/DOS, y en estaciones de trabajo con el sistema operativo UNIX. Para facilitar la compilación de los programas, se incluye en cada directorío un archivo `makefile` que permite compilar todos los archivos del mismo. El directorío raíz contiene un archivo ejecutable llamado `LEEME.COM` que explica el contenido del disquete en detalle.

Nos gustaría mostrar nuestro agradecimiento a todas las personas que han contribuido con su ayuda y sus comentarios durante la preparación del manuscrito. Este agradecimiento se dedica especialmente a Vicente Luque y José María Uñón, por su ayuda en la compilación de los programas originales.

Jesús Carretero
jcarrete@fi.upm.es

Félix García
fgarcia@dati.fi.upm.es

Fernando Pérez
fperez@fi.upm.es

Fundamentos de C

Objetivos

Este capítulo le da la oportunidad de aprender lo siguiente:

1. Por qué se usa el lenguaje de programación C y para qué es necesario usar C en un computador.
2. Algunas órdenes básicas de C y cómo usarlas para construir un programa estructurado.
3. Cómo reconocer las estructuras de bloque y cómo usarlas para construir programas.
4. Qué es un bloque de programador y para qué es útil.
5. Los elementos básicos necesarios para escribir un programa en C.
6. El propósito de los archivos incluidos (`include`).
7. Cómo funciona la función `printf()`.
8. Qué son los identificadores y cómo usarlos.
9. Las palabras reservadas usadas en C.
10. La importancia de declarar variables y cómo hacerlo.
11. Los métodos que usa C para ejecutar las operaciones aritméticas.
12. Las maneras de formatear la salida usando la función `printf()`.
13. La utilización de la función `scanf()` para recoger datos de entrada del usuario.
14. Algunas razones frecuentes para los problemas que aparecen en un programa C.
15. Los pasos a seguir para desarrollar un programa usando el lenguaje C.

Palabras clave

Entorno C
Editor
Compilador
Código fuente

Archivos para incluir
Archivos biblioteca
Enlazador
Sentencia compuesta

Estructura del programa	Salida estándar
Bloque de programa	Especificador de formato
Comentarios	Argumentos
Diseño descendente	Campo
Memoria del computador	Función
Componentes léxicos	Declaración
Palabra reservada	Identificador
Carácter	Especificador de tipo
Entero	Operador de asignación
Real	Prioridad (Precedencia)
Tipo de datos	Operador de asignación compuesto
Sentencia	Operador aritmético
Expresión	Secuencia de escape
Sentencia simple	Especificador de ancho de campo
	Notación E (Exponencial)

Contenido

1.1. El entorno de C	1.10. Leyendo la entrada del usuario
1.2. ¿Por qué C?	1.11. Implementación y depuración de programas: errores de programación frecuentes
1.3. Estructura de un programa	1.12. Programa de aplicación: conversión de temperatura.
1.4. Elementos de C	1.13. Programas de aplicación adicionales.
1.5. La función <code>printf()</code>	1.14. Programación en ANSI C
1.6. Identificación de cosas	
1.7. Declaración de cosas	
1.8. Introducción a los operadores de C	
1.9. Más <code>printf()</code>	

Introducción

Este capítulo le presenta algunas cosas fundamentales del lenguaje C. Aprenderá cómo visualizar valores en la pantalla del monitor y cómo obtener valores desde el programa de usuario. Este capítulo muestra también cómo hacer operaciones aritméticas básicas en C.

Cuando haya completado este capítulo, será capaz de escribir sus primeros programas relacionados con la tecnología en lenguaje C. El capítulo concluye con el diseño de una aplicación tecnológica real y algunos programas de ejemplo para ilustrar las aplicaciones tecnológicas de C.

1.1. EL ENTORNO DE C

Visión global

Esta sección presenta lo que se denomina el **entorno de C**. Un entorno en programación incluye todas las distintas herramientas de programación necesarias para trabajar con un lenguaje de programación en particular.

Cuando se programa en el lenguaje de programación BASIC, el entorno está habitualmente incluido dentro del microcomputador (en su ROM). En este caso, todo lo que se suele necesitar es encender el computador, empezar a teclear órdenes BASIC y teclear un mandato que dé instrucciones al computador para ejecutar el programa. Este no es el caso de otros lenguajes de programación, especialmente C.

El entorno de C

El entorno de C incluye un **editor**, un **compilador**, **archivos para incluir** (include), **archivos de biblioteca**, un **enlazador** y mucho más. Las funciones de estos componentes son:

- **Editor.** Permite introducir y modificar el código fuente C.
- **Compilador.** Un programa que convierte el programa escrito en C en un código que entiende el computador.
- **Archivos para incluir.** Archivos formados por muchas definiciones separadas e instrucciones que pueden ser útiles al programador en ciertas situaciones.
- **Archivos de biblioteca.** Son programas previamente compilados que realizan funciones específicas. Estos programas se usan para ayudar al programador a desarrollar programas en C. Por ejemplo, la función C que permite imprimir texto en la pantalla del computador (la función `printf`) no está predefinida en el lenguaje C. Su código está en un archivo de biblioteca. Lo mismo ocurre con otras muchas funciones, tales como las gráficas, las de sonido y las de trabajo con discos e impresoras, por nombrar unas cuantas. El programador puede también crear sus propios archivos de biblioteca con rutinas desarrolladas por él mismo para usarlas repetidamente en distintos programas C. Haciendo esto, se pueden ahorrar horas en tiempo de programación y evitar muchos errores de programación.
- **Enlazador.** Esencialmente, el enlazador combina todas las partes necesarias (tales como archivos de biblioteca) de un programa C para producir el código ejecutable final. Los enlazadores juegan un papel importante y necesario en todos los programas C. En programas C grandes, es una práctica habitual romper el programa en partes más pequeñas cada una de las cuáles se desarrolla por separado. El enlazador tendrá que combinarlas para formar el código del programa ejecutable final.

También será necesario algún tipo de sistema operativo en disco para ayudar a almacenar los programas. Las partes principales del entorno del lenguaje C se muestran en la Figura 1.1.

Conclusión

En esta sección se ha presentado el entorno de C. Ahora el usuario puede tener una idea de lo que hace falta para introducir un programa C en el computador y de las otras cosas que son necesarias para que el programa haga lo que quiere el programador (para ejecutar el programa).

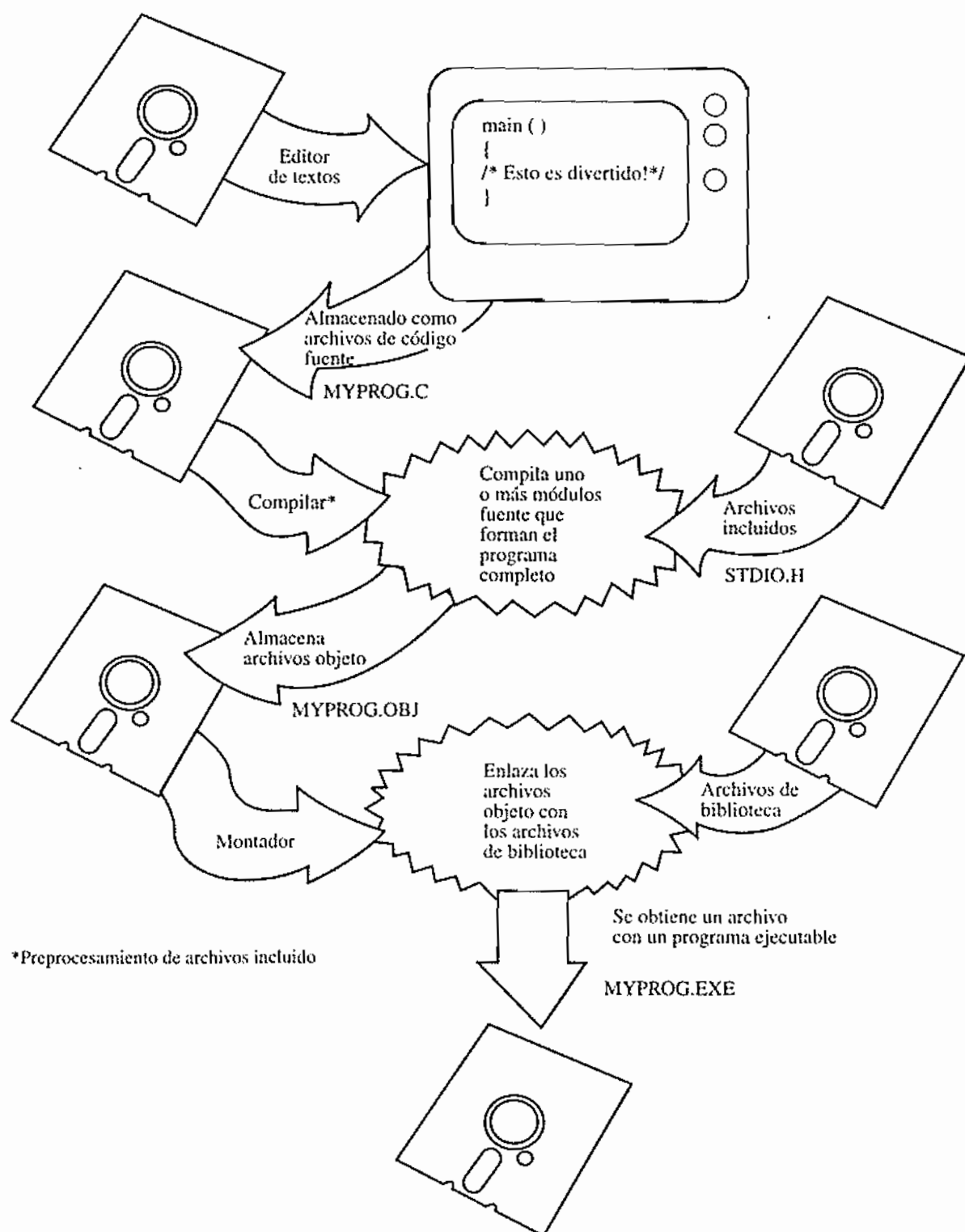


Figura 1.1. El entorno de C.

En la siguiente sección usted aprenderá algunas de las razones por las que C es un lenguaje de programación tan importante y cómo puede usarlo.

Repaso de la Sección 1.1

1. Explique el propósito de un editor. ¿Para qué es necesario?
2. Dé la razón por la que usar un compilador.
3. Explique las acciones de un compilador.

1.2. ¿POR QUÉ C?

Antecedentes

C es un lenguaje de programación cada vez más popular en las industrias, las escuelas y como lenguaje de uso personal. Las diversas razones para que esto sea así se muestran en la Tabla 1.1.

Desventajas de C

Debido a que C es un lenguaje versátil, su código puede ser escrito con tal brevedad y concisión que se vuelve casi ilegible. Este estilo de programación no es recomendado y no es usado en este libro. Debido a su flexibilidad, C le permitirá escribir programas que podrían acabar teniendo **errores** o *pulgas* (bugs) muy difíciles de encontrar. Programar en C es como tener un coche de carreras con una velocidad límite de 500 kph y muy pocas restricciones en la autopista —hay que ser cuidadosos con el manejo de este poder sin restricciones.

Tabla 1.1. La potencia de C

Ventajas	Lo que esto significa para usted
Diseñado para programación arriba-abajo	Su programa será más fácil de diseñar.
Diseñado para ser estructurado	Su programa será más fácil de leer y comprender.
Permite el diseño modular	Mejora la apariencia de los programas, de forma que otros pueden seguirlos y modificarlos fácilmente. Hace más fácil la depuración.
Un lenguaje eficiente	Programas más compactos y más rápidos.
Transportable	Un programa escrito en un tipo de computador servirá en otro tipo con pocos, o ningún cambio.
Control del computador	Usted tiene un control casi absoluto sobre su computador.
Flexibilidad	Se pueden crear fácilmente otros lenguajes y sistemas operativos.

El aspecto de C

Aquí hay un programa C:

Programa 1.1

```
#include <stdio.h>

/* Este es un programa C. Imprime un mensaje en la pantalla
   del computador */
main()
{
    printf("Envíeme una resistencia de 10 ohmios.");
}
```

Se pueden resaltar varias cosas del Programa 1.1. Es fácil de leer. Nadie duda de lo que hará el programa. El programa tiene una estructura que le hace fácil de leer. Hay algunos símbolos extraños (tales como el `/*`, que se explicará en breve). Lo que se ve en este programa es el código fuente. Esto es lo que usted, el programador, tecleará.

En el Programa 1.1 se presentan muchos de los elementos necesarios en un programa C. Estos elementos son las palabras reservadas `include`, `main` y `printf`.

Cuando se ejecuta el programa, se escribe:

```
Envíeme una resistencia de 10 ohmios.
en la pantalla del computador.
```

Las distintas partes

Como puede verse en la Tabla 1.2, todos los programas C deben tener ciertos componentes fijos. No se preocupe de los otros elementos (tales como el `printf`) usados en el Programa 1.1. Han sido usados para indicar en C cómo imprimir la sentencia en el monitor. En breve aprenderá el significado de estas sentencias y cómo usarlas.

Tabla 1.2. Partes principales de un programa C

Elemento	Propósito
<code>#include <stdio.h></code>	Le indica al compilador que debe <i>incluir</i> el archivo de entrada/salida estándar.
<code>main</code>	Marca el punto donde el programa C comienza la ejecución. Es obligatorio en todos los programas.
<code>()</code>	Deben aparecer inmediatamente detrás de <code>main</code> . Habitualmente, entre estos paréntesis hay información que será usada por el programa.
<code>/* */</code>	Estos símbolos son opcionales y se usan para delimitar los comentarios. Los comentarios son anotaciones usadas para clarificar el programa a otras personas. Son ignorados por el compilador.
<code>;</code>	Cada sentencia C termina con un punto y coma. Por ahora puede pensar que una sentencia C está formada por un mandato (orden) C.
<code>{ }</code>	Las llaves son necesarias en todos los programas C. Indican el principio y el final de las instrucciones del programa.

Conclusión

¡Felicidades! Ha visto su primer programa C. Dicho programa imprime un mensaje en la pantalla, algo que cualquier computador puede hacer fácilmente sin usar C. Ha sido pensado para mantener las cosas tan sencillas como sea posible permitiéndole concentrarse en los elementos principales de un programa C.

En la próxima sección aprenderá algunas cuestiones importantes acerca de la estructura y por qué usted debería usarla.

Repaso de la Sección 1.2

1. Describa tres ventajas del lenguaje de programación C.
2. Explique qué significa la transportabilidad en un lenguaje de programación.
3. ¿Con qué mandato deben empezar todos los programas C?
4. Ponga un ejemplo de un comentario en un programa C.
5. ¿Cuál es el propósito de las llaves { } en un programa C?

1.3. ESTRUCTURA DE UN PROGRAMA

Presentación

En este libro, todos los programas C se ajustarán a una estructura específica. Se puede pensar en la **estructura de un programa** como en el formato que se usa cuando se introduce el programa. El Programa 1.1 tenía una estructura:

```
#include <stdio.h>
/* Este es un programa C. Imprime un mensaje en la pantalla
   del computador */
main()
{
printf("Envíeme una resistencia de 10 ohmios.");
}
```

Cuando se compile y ejecute, este programa imprimirá algo en el monitor (en la primera línea empezando en la esquina izquierda de la pantalla)

Envíeme una resistencia de 10 ohmios.

El Programa 1.1 podía haberse escrito con una estructura distinta:

Programa 1.2 `#include <stdio.h>`

```
/* Este es un programa en C. Imprime un mensaje en la */
/* pantalla del computador. */
main () {printf("Envíeme una resistencia de 10 ohmios."); }
```

Cuando se compile y ejecute, el Programa 1.2 hará exactamente lo mismo que el Programa 1.1. La diferencia es que la estructura ha cambiado. El Programa 1.2 es un poco más difícil de leer. Se podría haber escrito el mismo programa así:

Programa 1.3 `#include <stdio.h>`

```
main() {printf("Envieme una resistencia de 10 ohmios."); }
```

Una vez más, cuando se compile y ejecute, el Programa 1.3 hará lo mismo. El asunto es que la estructura de un programa se considera buena cuando permite que la gente lo lea y entienda fácilmente.

Para un programa tan sencillo como el presentado aquí, la estructura del programa no parece introducir una gran diferencia. Sin embargo, en programas más grandes, la estructura del programa es muy importante.

Programación estructurada frente a no estructurada

Una de las medidas de un «buen» programa de computador es que pueda ser leído y comprendido por cualquiera —incluso si esa persona no sabe programar.

Por ejemplo, el primer programa C presentado en este libro era fácil de comprender. Se admite que no hacía mucho, pero la cuestión es que si se mantiene una buena estructura, el programa será más fácil de comprender, modificar y depurar cuando sea necesario. Un programa no estructurado es aquel en el que no se ha hecho ningún esfuerzo, o muy poco, para ayudar a la gente a leerlo y comprenderlo. Recuerde que la estructura de un programa no supone ninguna diferencia para el computador, solamente para la gente que tiene que trabajar con él.

Bloques de un programa

Se puede pensar que un programa estructurado tiene ciertas partes del programa situadas en posiciones particulares en el documento del programa. Se puede pensar en estas posiciones como en *bloques* de información. Una carta contiene bloques de información:

Juan Estudiante
123 Carretera de la Fábrica de Páginas
Villa Programa, EE.UU.
01234

La Compañía de Resistencias
321 Carretera de la Página de Fábricas
Código Fuente, EE.UU.
43210

Estimado Sr.:

Envíeme una resistencia de 10 ohmios.

Sinceramente:

Juan Estudiante

Pta.: Gracias por la rápida entrega de mi último pedido.

La carta anterior podría haberse escrito de la siguiente forma:

Juan Estudiante 123 Carretera de la Fábrica de Páginas Villa Programa, EE.UU.
01234 La Compañía de Resistencias 321 Carretera de la Página de Fábricas Código Fuente, EE.UU. 43210 Estimado Sr.: Envíeme una resistencia de 10 ohmios.
Sinceramente: Juan Estudiante Pta.: Gracias por la rápida entrega de mi último pedido.

La diferencia entre ambas cartas es que una está estructurada y la otra no. La carta no estructurada costaría menos tiempo de escritura, menos tiempo de impresión y usaría menos papel. Si se hicieran todas las cartas de esta forma durante un periodo de tiempo largo incluso podría ahorrarse algo en envíos de correo. Pero serían muy difíciles de leer, no debido al estilo de escritura, sino debido a su estructura.

A continuación se muestra la misma carta, pero esta vez se hace énfasis en la estructura de bloques:

Juan Estudiante
123 Carretera de la Fábrica de Páginas
Villa Programa, EE.UU.
01234

← Bloque de dirección de respuesta

La Compañía de Resistencias
321 Carretera de la Página de Fábricas
Código Fuente, EE.UU.
43210

← Bloque de dirección de destino

Estimado Sr.:

← Bloque de saludo

Envíeme una resistencia de 10 ohmios.

← Cuerpo de la carta

Sinceramente:

Juan Estudiante

← Cierre de la carta

Pta.: Gracias por la rápida entrega de mi último pedido.

La estructura de bloques hace la carta más fácil de leer. De forma similar, cuando se usa la estructura de bloques en programas, también son más fáciles de leer. Al igual que existe una estructura de bloques establecida para las cartas, en este libro se usará una estructura de bloques establecida para escribir los programas en C.

El bloque del programador

Cada programa en C bien documentado debe empezar con un bloque llamado **bloque del programador**. Está compuesto por **comentarios** que contienen la siguiente información:

1. Nombre del programa
2. Programador
3. Descripción del programa
4. Explicación de todas las variables
5. Explicación de todas las constantes

A estas alturas, usted ya sabe suficiente C como para hacer esto: suponga que tiene que escribir un programa C que resuelva la caída de voltaje en una resistencia de 10 ohmios cuando la atraviesa una intensidad determinada. La fórmula para esta relación es:

$$V = I \times R$$

Donde

V = El voltaje a través de la resistencia medido en voltios.

I = La corriente en la resistencia medida en amperios.

R = El valor de la resistencia medida en ohmios.

El programa C debería empezar con el bloque del programador. Esto se ilustra en el Programa 1.4.

```
Programa 1.4 #include <stdio.h>

/*
Programa: Cálculo de voltaje
Desarrollado por : Un buen programador

Descripción: Este programa calcula el voltaje que cae
              en una resistencia de 10 ohmios. El usuario debe
              introducir el valor de la intensidad.

Variables:
    V = Voltaje en la resistencia.
    I = Intensidad en la resistencia.
```

```

Constantes:
    R = 10 (Una resistencia de 10 ohmios)
*/

main()
{
    /* Cuerpo del programa que realiza lo descrito arriba */
}

```

El Programa 1.4 se compilará y ejecutará, pero no ocurrirá nada porque no se ha puesto ningún mandato (orden) dentro del programa. El programa tiene solamente comentarios y las partes esenciales de un programa C. Pero el bloque de programador está completo. Dice exactamente lo que el programa hará y, igual de importante, define todas las variables (V e I) y el valor de la constante R que serán usadas en el programa. Cuando se hace un diseño **descendente** (presentado posteriormente en este capítulo), definir el problema de programación con palabras es el primer paso esencial en el diseño del programa. Seleccionar las variables y definirlas es otro paso importante. Por ahora, sólo debe saber lo que tiene que estar incluido en un bloque de programador y conocer la mecánica para escribir ese bloque usando C.

Ventajas y desventajas de la programación estructurada

La principal desventaja de la programación estructurada afecta principalmente a aquellos que han aprendido a programar de forma no estructurada. Los viejos hábitos son difíciles de eliminar. Si nunca ha programado antes, la programación estructurada no tendrá desventajas para usted. Otra desventaja de la programación estructurada es que hace más largos los programas cortos hechos de forma no estructurada.

En el pasado, la **memoria del computador** era relativamente cara y limitada. Por lo que se pagaba por hacer programas tan breves como era posible para ahorrar espacio en la memoria del computador. Este no es el caso actualmente. Incluso las calculadoras de bolsillo tienen más memoria que muchas de las grandes máquinas antiguas. Por tanto, no hay ya necesidad de ser breve programando. Hay, sin embargo, necesidad de ser claro programando y de desarrollar buenos hábitos de programación que resulten en programas completos que sean fáciles de comprender, modificar y corregir. Este es el propósito de este libro.

Lo que aprenda en este texto podrá ser aplicado a cualquier lenguaje estructurado como Pascal e incluso BASIC. Cuando complete este texto, podrá crear programas de forma eficiente usando técnicas de programación estructurada.

La Figura 1.2 ilustra la estructura de un programa C completo usando el estándar de prototipos ANSI. Los detalles de programación serán tratados en los próximos capítulos. Por ahora, estudie la estructura del programa para tener una idea general.

12 PROGRAMACIÓN ESTRUCTURADA EN C

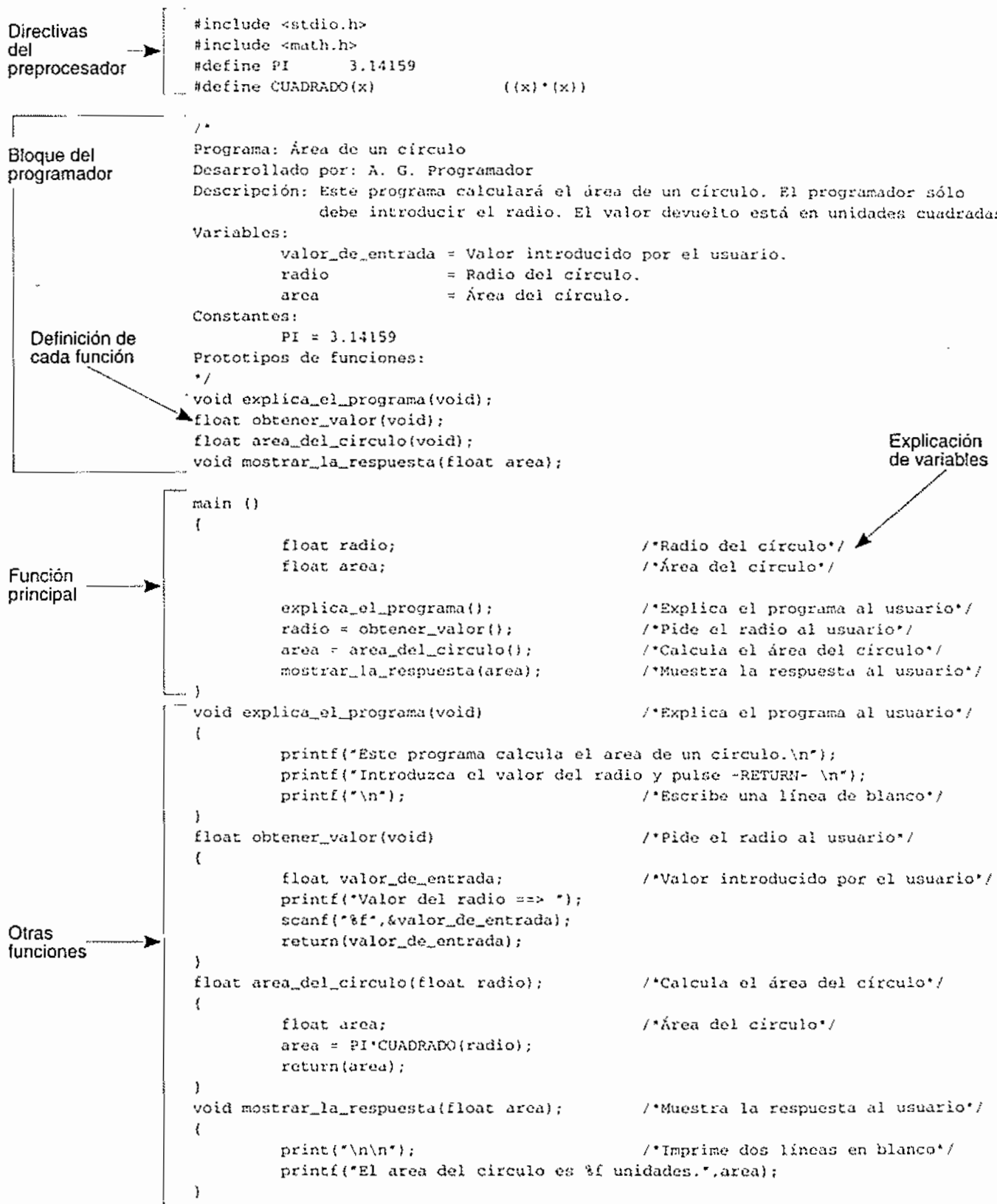


Figura 1.2. Estructura de un programa C.

Conclusión

En esta sección se ha presentado una idea general de las diferencias entre programación estructurada y no estructurada. Aprenderá mucho más sobre estas áreas, incluyendo la potencia de C como lenguaje de programación. Compruebe su nivel de comprensión en la sección de repaso siguiente.

Repaso de la Sección 1.3

1. Defina el término «estructura» como se usa en programación.
2. ¿Es necesario dar estructura a un programa C para compilar sin errores?
3. Describa una estructura de bloque y dé un ejemplo.
4. Explique las razones para usar programación estructurada.
5. Describa un bloque de programador y defina lo que debe contener.

1.4. ELEMENTOS DE C

Esta sección establece las reglas básicas para los elementos fundamentales de todos los programas escritos en C. En esta sección verá muchas definiciones nuevas. Éstas definirán el escenario para el resto de este capítulo, así como para el resto del texto.

Lo que C necesita

Para escribir un programa en C, se usa un conjunto de caracteres. Este conjunto incluye las letras mayúsculas y minúsculas del alfabeto inglés, los diez dígitos decimales del sistema de numeración arábigo y el carácter subrayado (`_`). Los espacios en blanco (tales como espacios entre palabras) se usan para separar los elementos de un programa C, de forma similar a como se usan para separar las palabras en este libro. Los espacios en blanco también incluyen el tabulador y el salto de línea, así como otros caracteres de control que generan espacios en blanco.

Componentes léxicos. En cualquier código fuente C, el elemento más básico reconocido por el compilador es un carácter simple o grupo de caracteres conocido como **componente léxico** (token). Esencialmente, un componente léxico es un texto del programa fuente que el compilador no troceará más —se trata como unidad fundamental. Como ejemplo, en `C main` es un componente léxico; también lo son la llave izquierda (`{`) y el signo de sumar (`+`).

Palabras reservadas en ANSI C. Las palabras reservadas son componentes léxicos predefinidos que tienen un significado especial para el compilador de C. Sus definiciones no pueden ser cambiadas; por tanto no pueden ser usadas para nada más que la finalidad que tienen en el programa en que son usadas. Las palabras reservadas son las siguientes:

auto	double	int	struct	break	else
long	switch	case	enum	register	typedef
char	extern	return	union	const	float
short	unsigned	continue	for	signed	void
default	goto	sizeof	volatile	do	if
static	while				

Tipos de datos. El lenguaje C tiene tres tipos de datos principales: números, caracteres y cadenas de caracteres. Un **carácter** es cualquier elemento del conjunto de caracteres usado por C. Una **cadena de caracteres** es una combinación de estos caracteres.

Números usados por C. C usa un amplio rango de números. Los números usados por C caen en dos grandes categorías generales: **enteros** (números enteros) y **reales** (números con punto decimal). Estas dos grandes categorías puede subdividirse como se ve en la Tabla 1.3.

Como puede verse en la Tabla 1.3, C ofrece una rica variedad de **tipos de datos**. Generalmente hablando, cuanto mayor es el rango de valores de un tipo de datos, más memoria del computador cuesta su almacenamiento. Como regla general, se desea usar tipos de datos que conserven la memoria al tiempo que satisfagan el propósito deseado. Por ejemplo, si se necesita un tipo de datos para contar objetos —tales como el número de resistencias de los pedidos— el tipo de datos `int` será probablemente adecuado. Sin embargo, si se está escribiendo un programa con tanta precisión como sea posible, se podría considerar el uso del tipo `double`, que puede dar una precisión de 15 dígitos.

Como puede verse en la Tabla 1.3, algunos tipos de datos tienen el mismo rango de valores (tal como `int` y `short`). Esto es así en un IBM PC, pero pueden tener rangos distintos en otros computadores.

Tabla 1.3. Subdivisión de los tipos de datos

Identificador de tipo	Significado	Rango de valores (IBM PC)
<code>char</code>	carácter	-128 a 127
<code>int</code>	entero	-32.768 a 32.767
<code>short</code>	entero corto	-32.768 a 32.767
<code>long</code>	entero largo	-2.147.483.648 a 2.147.483.647
<code>unsigned char</code>	carácter sin signo	0 a 255
<code>unsigned</code>	entero sin signo	0 a 65.535
<code>unsigned short</code>	entero corto sin signo	0 a 65.535
<code>unsigned long</code>	entero largo sin signo	0 a 4.294.967.295
<code>enum</code>	enumerado	0 a 65.535
<code>float</code>	real (coma flotante)	3,4E +/- 38 (7 dígitos)
<code>double</code>	real doble	1,7E +/- 308 (15 dígitos)
<code>long double</code>	real doble largo	1,7E +/- 4932 (15 dígitos)

Sentencias de C

Una **sentencia** (instrucción) de C controla el flujo de ejecución de un programa. Está formada por palabras reservadas, expresiones y otras instrucciones. En C, una expresión es una combinación de operandos y operadores que expresa un valor único (tal como `respuesta = 3 + 5;`).

Hay dos tipos de sentencia en C: sentencias **simples y compuestas**. Una *sentencia compuesta* está delimitada por llaves (`{ }`), mientras que una *sentencia simple* termina con un punto y coma (`;`). A medida que vaya progresando con este libro, se harán más referencias a sentencias de C.

Conclusión

En esta sección se han mostrado los elementos de un programa C. El resto de este capítulo le mostrará cómo usarlos. Ponga a prueba lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 1.4

1. Indique qué se usa para escribir un programa en C.
2. Explique lo que es un componente léxico en C.
3. ¿Cuáles son los tipos de datos principales usados en C?
4. Explique qué es una palabra reservada. Dé un ejemplo.
5. ¿Qué tipo de datos maneja el número mayor?

1.5. LA FUNCIÓN `printf()`

Esta sección presenta la poderosa función `printf()` usada en C. Es realmente una función separada (como `main()` es una función) y está incluida en la biblioteca estándar que tienen todos los sistemas con C.

Lo que hace `printf()`

La función `printf()` se usa para escribir información por la salida estándar (normalmente la pantalla del computador). Ya se usó `printf()` en la sección anterior. La estructura de esta función es:

```
printf (cadenas de caracteres con especificadores de formato,
        variables o valores);
```

Los caracteres son delimitados por comillas simples (tal como `'a'`) y las cadenas de caracteres son delimitadas por comillas dobles (tal como `"Esto es una cadena de caracteres"`). Un **especificador de formato** instruye a la función `printf()` sobre la forma de convertir, dar forma e imprimir sus **argumentos**. Por ahora, puede pensar en

un argumento como los valores reales que hay entre los paréntesis de la función. Un especificador de formato empieza con el caracter del porcentaje (%). Por ejemplo:

```
printf ("Esto es una instrucción de C.");
```

cuando se ejecuta produce:

Esto es una instrucción de C.

Con un especificador de formato y un argumento,

```
printf ("El numero 92 en decimal es %d.", 92);
```

cuando se ejecuta produce:

El numero 92 en decimal es 92.

Otra forma de producir la misma salida es:

```
printf ("El numero %d en decimal es %d.", 92, 92);
```

La Tabla 1.4 muestra los distintos tipos de campos usados por la función `printf()` en los especificadores de formato.

Tabla 1.4. Especificadores de formato de tipos usados por `printf()`

Carácter	Argumento	Salida resultante
d	entero	Entero con signo en base decimal.
i	entero	Entero con signo en base decimal.
o	entero	Entero sin signo en base octal.
u	entero	Entero sin signo en base decimal.
x	entero	Entero sin signo en base hexadecimal usando letras minúsculas.
X	entero	Entero sin signo en base hexadecimal usando letras mayúsculas.
f	real	Número real con signo.
e	real	Número real con signo usando notación e.
E	real	Número real con signo usando notación E.
g	real	Número real con signo en formato e ó f, pero de tamaño corto.
G	real	Número real con signo en formato E ó f, pero de tamaño corto.
c	carácter	Un carácter individual.
s	cadena de caracteres	Imprimir cadenas de caracteres.
%	ninguno	Imprime el símbolo %.

El Programa 1.5 ilustra el uso de los distintos **campos** especificadores de tipo de formato.

Programa 1.5 `#include <stdio.h>`

```

main()
{
    printf("El valor 92 usando el tipo de campo d es %d. \n", 92);
    printf("El valor 92 usando el tipo de campo i es %i. \n", 92);
    printf("El valor 92 usando el tipo de campo u es %u. \n", 92);
    printf("El valor 92 usando el tipo de campo o es %o. \n", 92);
    printf("El valor 92 usando el tipo de campo x es %x. \n", 92);
    printf("El valor 92 usando el tipo de campo X es %X. \n", 92);
    printf("El valor 92.0 usando el tipo de campo f es %f. \n", 92.0);
    printf("El valor 92.0 usando el tipo de campo e es %e. \n", 92.0);
    printf("El valor 92.0 usando el tipo de campo E es %E. \n", 92.0);
    printf("El valor 92.0 usando el tipo de campo g es %g. \n", 92.0);
    printf("El valor 92.0 usando el tipo de campo G es %G. \n", 92.0);
    printf("El valor 92 usando el tipo de campo c es %c. \n", 92);
    printf("El caracter '9' usando el tipo de campo c es %c. \n", 9);
    printf("La cadena 92 usando el tipo de campo s es %s. \n", "92");
}

```

La ejecución del Programa 1.5 produce la siguiente salida:

```

El valor 92 usando el tipo de campo d es 92.
El valor 92 usando el tipo de campo i es 92.
El valor 92 usando el tipo de campo u es 92.
El valor 92 usando el tipo de campo o es 134.
El valor 92 usando el tipo de campo x es 5c.
El valor 92 usando el tipo de campo X es 5C.
El valor 92.0 usando el tipo de campo f es 92.000000
El valor 92.0 usando el tipo de campo e es 9.20000e+01.
El valor 92.0 usando el tipo de campo E es 9.20000E+01.
El valor 92.0 usando el tipo de campo g es 92.
El valor 92.0 usando el tipo de campo G es 92.
El valor 92 usando el tipo de campo c es \.
El caracter '9' usando el tipo de campo c es 9.
La cadena 92 usando el tipo de campo s es 92.

```

Por ahora, no se preocupe acerca del símbolo `\n` que se ve en cada función `printf()`. Aprenderá más sobre estos símbolos en este capítulo. Lo que hacen es causar un retorno de carro y una nueva línea de salida. Sin ellos en el Programa 1.5, toda la salida estaría junta.

Dése cuenta de que para el argumento de tipo carácter se han usado las comillas simples mientras que para el argumento de tipo cadenas de caracteres se han usado las comillas dobles. Dése también cuenta de que el valor numérico del *backslash* (\) es 92. (En el apéndice C encontrará la tabla de caracteres ASCII).

Más de uno

Se puede usar más de un especificador de formato en la función `printf()`. Sin embargo, debe haber al menos tantos argumentos como especificadores de formato; si no, los resultados serán impredecibles. Se pueden tener más argumentos que especificadores de formato; sin embargo, los argumentos extra serán ignorados.

Un ejemplo del uso de varios especificadores de formato es:

```
printf ("Un caracter es %c y un numero es %d.", 'a', 53);
```

Dése cuenta de que los argumentos se separan mediante comas.

Conclusión

En esta sección se ha presentado la función `printf()` de C. Se ha visto cómo usar esta función para imprimir cadenas de caracteres, caracteres y números. También se ha visto qué es un especificador de formato y cómo usar varios campos de especificación de formato. Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 1.5

1. Defina el propósito de la función `printf()`.
2. ¿Cómo se distinguen los caracteres de las cadenas de caracteres?
3. ¿Cuál es el propósito de un especificador de formato en la función `printf()`?
4. ¿Qué es un argumento?
5. Defina la regla que relaciona el número de especificadores de formato y de argumentos.

1.6. IDENTIFICACIÓN DE COSAS

Presentación

En esta sección aprenderá cómo usar palabras para especificar partes de su programa C. También se hará una introducción a las funciones. Verá como dar nombres a cadenas de caracteres, valores y partes de su programa. Saber cómo hacer esto hará sus trabajos de programación mucho más sencillos en cualquier lenguaje de programación.

¿Qué es una función?

Una **función** es una colección independiente de **declaraciones** y sentencias (instrucciones). Una declaración establece la relación entre el nombre y el tipo de una variable u otra función. Aprenderá más sobre este tema más tarde en este capítulo. Lo importante por ahora es que se dé cuenta de que una función se diseña habitualmente para realizar una tarea. Todo programa C debe tener al menos una función llamada `main()`. Dividir las tareas en partes separadas hace que los programas sean más fáciles de diseñar, corregir y modificar.

¿Qué necesita identificación?

Cuando se crea una función, es mejor darla un nombre descriptivo. Por ejemplo, una función que calcule la potencia total disipada en una resistencia se podría llamar:

```
potencia_resistencia ()
```

Esto es más descriptivo que llamarla:

```
funcion_1 ()
```

Ambos ejemplos usan **identificadores** para distinguir una función de la otra. Un identificador no es nada más que el nombre que se da a una parte de un programa C. Los identificadores pueden usarse para nombrar partes de una fórmula, como:

```
total = resistencia_1 + resistencia_2;
```

Un identificador puede usarse para asignar un valor constante que describa el valor y pueda ser usado en un programa:

```
PI = 3.14159;  
area_circulo = PI * radio * radio;
```

Como puede verse en los ejemplos anteriores, los identificadores juegan un importante papel en un programa C.

Creación de sus propios identificadores

Hay algunas reglas a seguir cuando cree sus propios identificadores. Primero, todo identificador debe comenzar con una letra del alfabeto (mayúscula o minúscula) o el subrayado `_`. El resto del identificador puede utilizar cualquier conjunto de letras (mayúsculas o minúsculas), dígitos (0 a 9) y el subrayado `_`—y esto es todo— no se permite el uso de otros caracteres. Esto significa que los espacios no están permitidos en los identificadores. La mayoría de los sistemas de C distinguirán, por lo menos, los primeros 31 caracteres de un identificador. El Ejemplo 1.1 ilustra este punto.

Ejemplo 1.1 ¿Cuáles de los siguientes son identificadores legales en C?

A. Este_1	E. Uno por la carretera
B. _Estel	F. El_Siguiente:
C. 1_por_la_carretera	G. E=_IR
D. Uno_por_la_carretera	

Solución

Tenga en mente las reglas para identificadores explicadas previamente a este ejemplo.

A. Este_1	Un identificador legal en C.
B. _Estel	Un identificador legal en C.
C. 1_por_la_carretera	No es un identificador legal en C. No empieza por una letra del alfabeto inglés o subrayado.
D. Uno_por_la_carretera	Un identificador legal en C.
E. Uno por la carretera	No es un identificador legal en C. Los espacios no están permitidos.
F. El_Siguiente:	No es un identificador legal en C. El símbolo : no está permitido.
G. E=_IR	No es un identificador legal en C. El símbolo = no está permitido.

Sensibilidad a las mayúsculas y minúsculas

Los identificadores en C son sensibles a las mayúsculas y minúsculas. Esto significa que C hace distinción entre las letras mayúsculas y minúsculas en un identificador. Así, en lo que concierne a C, los siguientes identificadores son distintos:

pi PI Pi pI

Y los siguientes:

Este ESTE este

Todos los identificadores de arriba son legales en C, pero no son iguales. Esto significa que se debe ser cuidadoso con el uso de los identificadores en C. Por ejemplo, si se define un identificador `pi` igual a 3,14159, se deben usar las letras minúsculas `pi` en cualquier parte del programa donde se espere que el identificador valga 3,14159. Si se usa `PI` o `Pi`, ninguno de los cuales tiene valor asignado, el programa contendrá errores.

Palabras reservadas

Un identificador no puede deletrearse igual y tener el mismo tipo de letra (mayúscula o minúscula) que una palabra reservada. El Ejemplo 1.2 ilustra este punto.

Ejemplo 1.2 Indique cuáles de los siguientes identificadores son legales y cuáles son palabras reservadas e indique si algunos de ellos son iguales.

A. Ley_de_Ohm	D. reconstruir
B. continue	E. Reconstruir
C. Ley de Ohm	F. _continue

Solución

Tenga en mente las reglas para identificadores explicadas previamente a este ejemplo.

A. Ley_de_Ohm	Un identificador legal en C, no reservado.
B. continue	Una palabra reservada legal en C.
C. Ley de Ohm	No es un identificador legal en C. Los espacios no están permitidos.
D. reconstruir	Un identificador legal en C, no reservado.
E. Reconstruir	Un identificador legal en C, no reservado.
F. _continue	Un identificador legal en C, no reservado. (Si embargo, se parece tanto a una palabra reservada que no se recomienda su uso).

Conclusión

En esta sección se han presentado conceptos importantes para nombrar las partes de un programa C. Ha aprendido lo que es un identificador y cómo puede usarse para nombrar (identificar) partes de sus programas C. Compruebe su nivel de comprensión de esta sección realizando la siguiente sección de repaso.

Repaso de la Sección 1.6

1. ¿Qué es una función C?
2. ¿Qué es un identificador?
3. ¿Cuáles son las reglas para crear sus propios identificadores?
4. ¿Cuántos caracteres de cualquier identificador son reconocidos por C?

1.7. DECLARACIÓN DE COSAS

En esta sección aprenderá que todas las variables deben ser declaradas. Esto significa que el compilador debe conocer, por adelantado, antes de usar las variables, el identificador que se va a usar para cada variable así como el tipo de variable que se está usando. Al principio, esto puede parecer trabajo extra. Pero encontrará que haciendo esto se reducen mucho las posibilidades de cometer errores de programación.

¿Qué es una variable?

Se puede pensar en una variable como en una posición específica de memoria reservada para un tipo específico de datos y con un nombre para referenciarla fácilmente. Esencialmente, se usan variables para permitir que la misma posición de memoria pueda tener diferentes valores del mismo tipo en instantes de tiempo distintos. Por ejemplo, si se estuviera calculando el voltaje entre los extremos de una resistencia fija a medida que cambia la corriente, la variable voltaje tendría diferentes valores cada vez que cambia la corriente.

Declaración de variables

En C, todas las variables deben ser declaradas antes de ser usadas. Para declarar una variable, es necesario declarar su tipo e identificador. La Tabla 1.5 presenta los **especificadores de tipo** fundamentales que se usarán para declarar variables.

El Programa 1.6 ilustra cómo declarar variables.

Programa 1.6 #include <stdio.h>

```
main()
{
    char un_caracter;          /* Esto declara un carácter. */
    int un_entero;             /* Esto declara un entero. */
    float un_real;             /* Esto declara un real. */

    un_caracter = 'a';
    un_entero = 15;
    un_real = 27.62;
    printf("%c es el caracter.\n", un_caracter);
    printf("%d es el entero.\n", un_entero);
    printf("%f es el real.\n", un_real);
}
```

Tabla 1.5. Especificadores de tipos fundamentales en C

Enteros	Reales	Otros tipos
char	double	const
enum	float	void
int	long double	volatile
long		
short		
signed		
unsigned		

Cuando se ejecuta, el Programa 1.6 produce:

```
a es el caracter.
15 es el entero.
27.620000 es el real.
```

En el programa, las declaraciones de variables son:

```
char un_caracter;      /* Esto declara un caracter. */
int un_entero;         /* Esto declara un entero. */
float un_real;         /* Esto declara un real. */
```

Observe que la declaración de una variable consiste en definir su tipo y su identificador. En el cuerpo del programa, a cada variable se le ha asignado un valor:

```
un_caracter = 'a';
un_entero = 15;
un_real = 27.62;
```

Lo que ha sido hecho con el **operador asignación (=)**. Aprenderá más sobre este operador en la próxima sección.

Iniciación de variables

Se puede combinar una declaración de variable con el operador asignación, dando valor a la variable al mismo tiempo que es declarada. Este caso puede verse en el Programa 1.7.

```
Programa 1.7 #include <stdio.h>

main()
{
    char un_caracter = 'a';    /* Esto declara y asigna un caracter.
    */
    int un_entero = 15;        /* Esto declara y asigna un entero. */
    float un_real = 27.62;     /* Esto declara y asigna un real. */

    printf("%c es el caracter.\n", un_caracter);
    printf("%d es el entero.\n", un_entero);
    printf("%f es el real.\n", un_real);
}
```

Obsérvese que, en cada caso, a la declaración de variable le sigue un comentario que define el propósito de cada variable. Adquirir el hábito de escribir este tipo de documentación del programa es muy bueno.

Más del mismo tipo

En C, si hay variables del mismo tipo, se pueden declarar como sigue:

```
int numero_1, numero_2, numero_3;
```

Aunque esto es legal en C, esta forma de programar no se usará en este texto porque dificulta la escritura de comentarios para cada variable usada en el programa.

¿Por qué declarar?

Cuando se declaran variables, se recoge toda la información acerca de las mismas en un lugar determinado del programa. Esto permite a cualquiera leer el código fuente e identificar rápidamente los datos que se van a utilizar (asumiendo que se han añadido comentarios que dan una buena explicación). Esto fuerza también a hacer algo de planificación antes de sumergirse en el código del programa. Otra razón importante para hacer esto es que evita que se usen variables mal escritas dentro del programa. Si no hubiese requisitos de declaración, se podría crear un nuevo identificador sin saberlo. Cuanto más se programa, y cuanto más complejos y prácticos son los programas, más se agradece que las variables deban ser declaradas antes de usarlas.

Repaso de la Sección 1.7

1. ¿Cómo se puede pensar en una variable en términos de un programa C?
2. Explique qué significa declarar una variable.
3. Nombre tres especificadores de tipo fundamentales en C.
4. Explique qué significa iniciar una variable.
5. ¿Cuál es una buena razón para declarar las variables?

1.8. INTRODUCCIÓN A LOS OPERADORES DE C

En esta sección aprenderá cómo hacer que un programa C actúe sobre variables. Esto significa que aprenderá a hacer operaciones aritméticas así como otras operaciones que son específicas de C.

¿Qué son los operadores de C?

Un operador de C permite que el programa haga algo a las variables. Específicamente, un **operador aritmético** permite hacer operaciones aritméticas (como una suma, +) sobre variables. Esta sección presentará los operadores más frecuentemente usados en C.

Operadores aritméticos

Los operadores aritméticos frecuentemente usados en C pueden verse en la Tabla 1.6.

Tabla 1.6. Operadores aritméticos frecuentemente usados

Símbolo	Significado	Ejemplo
+	Suma	respuesta = 3 + 5 (respuesta → 8)
-	Resta	respuesta = 5 - 3 (respuesta → 2)
*	Multiplicación	respuesta = 5 * 3 (respuesta → 15)
/	División	respuesta = 10 / 2 (respuesta → 5)
%	Módulo	respuesta = 3 % 2 (respuesta → 1)

El Programa 1.8 ilustra el uso de los cuatro primeros operadores aritméticos en C.

```

Programa 1.8 #include <stdio.h>

main()
{
    float numero_1 = 15.0;    /* Primer operador aritmético.    */
    float numero_2 = 3.0;    /* Segundo operador aritmético.    */
    float respuesta_suma;    /* Respuesta de la suma.    */
    float respuesta_resta;    /* Respuesta de la resta.    */
    float respuesta_multi;    /* Respuesta de la multiplicación    */
    float respuesta_division; /* Respuesta de la división.    */

    respuesta_suma = numero_1 + numero_2;
    respuesta_resta = numero_1 - numero_2;
    respuesta_multi = numero_1 * numero_2;
    respuesta_division = numero_1 / numero_2;

    printf("15 + 3 = %f\n", respuesta_suma);
    printf("15 - 3 = %f\n", respuesta_resta);
    printf("15 * 3 = %f\n", respuesta_multi);
    printf("15 / 3 = %f\n", respuesta_division);
}

```

La ejecución del Programa 1.8 produce:

```

15 + 3 = 18.000000
15 - 3 = 12.000000
15 * 3 = 45.000000
15 / 3 = 5.000000

```

Observe que todos los números usados en el Programa 1.8 son de tipo `float`. Es importante darse cuenta de que en C, la división de variables de tipo `int` truncará la respuesta. Esto significa que $5/2 = 2$ y que $2/3 = 0$. El operador resto (%) exige el uso de números enteros. Este operador es también llamado operador *módulo*.

Tabla 1.7. Precedencia de operadores

Prioridad	Operación
Primera	()
Segunda	Negación (asignar un número negativo)
Tercera	Multiplicación *, División /
Cuarta	Suma +, Resta -

Consideraciones importantes

C no permite una estructura que haga una asignación inversa. Esto significa que no se puede hacer una asignación a una expresión:

```
6 + 3 = respuesta; ← No permitido;
```

Tampoco se puede hacer una asignación a una constante:

```
3 = respuesta; ← No permitido;
```

Ambos intentos producirán errores de compilación.

Prioridad de operaciones

La **prioridad** de operaciones es simplemente el orden en el que se ejecutan las operaciones aritméticas. Por ejemplo, considere la expresión:

```
x = 5 + 4/2;
```

La prioridad de operaciones exige que la división se haga antes que la suma. Si no se hiciera así, la operación anterior podría interpretarse de dos formas distintas. Si la división se hiciera antes daría $(5 + 2 = 7)$. Si la suma se hiciera primero, se sumaría 5 a 4 y la suma sería dividida por 2 $(9/2 = 4,5)$. La interpretación de cualquier expresión debe ser consistente para obtener resultados fiables y predicables de los programas. Por ejemplo, si se quiere indicar que 5 se sumará a 4 primero y el resultado será luego dividido por 2, se deben usar paréntesis:

```
x = (5 + 4) / 2;
```

La Tabla 1.7 indica la precedencia (prioridad) de las operaciones en C.

En todos los casos, las operaciones se ejecutan de izquierda a derecha. El Ejemplo 1.3 lo ilustra.

Ejemplo 1.3 Determine los resultados de las siguientes operaciones:

A. $Y = 6 + 12/6 * 2 - 1$	C. $Y = 6 + 12/6 * (2 - 1)$
B. $Y = (6 + 12)/6 * 2 - 1$	D. $Y = 6 + 12/(6 * 2) - 1$

Solución

Usando la Prioridad de operadores, los resultados son:

- A. $Y = 6 + 12/6 * 2 - 1$
 $Y = 6 + 2 * 2 - 1$ (Operaciones de izquierda a derecha. Prioridad de división).
 $Y = 6 + 4 - 1$ (Prioridad de la multiplicación).
 $Y = 9$ (Operaciones de izquierda a derecha).
- B. $Y = (6 + 12)/6 * 2 - 1$
 $Y = 18/6 * 2 - 1$ (Las operaciones dentro de los paréntesis se hacen primero).
 $Y = 3 * 2 - 1$ (Operaciones de izquierda a derecha).
 $Y = 6 - 1$ (Prioridad de la multiplicación).
 $Y = 5$
- C. $Y = 6 + 12/6 * (2 - 1)$
 $Y = 6 + 12/6 * 1$ (Las operaciones dentro de los paréntesis se hacen primero).
 $Y = 6 + 2 * 1$ (Operaciones de izquierda a derecha).
 $Y = 6 + 2$ (Prioridad de la multiplicación).
 $Y = 8$
- D. $Y = 6 + 12/(6 * 2) - 1$
 $Y = 6 + 12/12 - 1$ (Las operaciones dentro de los paréntesis se hacen primero).
 $Y = 6 + 1 - 1$ (Prioridad de la división).
 $Y = 6$ (Operaciones de izquierda a derecha).
-

Operadores de asignación compuestos

En C, los operadores de asignación compuestos combinan el operador de asignación simple con otro operador. Por ejemplo, considere la instrucción:

```
respuesta = respuesta + 5;
```

Lo que esta instrucción significa es que a la posición de memoria denominada «respuesta» se le asignará un nuevo valor igual a su valor anterior más 5. No significa «respuesta es igual a respuesta más 5». El signo = no significa igualdad en C; significa *asignado a* y se llama operador de asignación. Así, en el ejemplo anterior, si respuesta tenía un valor de 10, entonces:

```
respuesta = respuesta + 5;
```

hará que el nuevo valor de respuesta sea 15.

La expresión anterior puede acortarse en C usando la asignación compuesta:

```
respuesta += 5;
```

La Tabla 1.8 muestra los operadores de asignación compuestos presentados al principio de esta sección.

Tabla 1.8. Asignaciones compuestas frecuentes en C

Símbolo	Ejemplo	Significado
<code>+=</code>	<code>X += Y;</code>	<code>X = X + Y;</code>
<code>-=</code>	<code>X -= Y;</code>	<code>X = X - Y;</code>
<code>*=</code>	<code>X *= Y;</code>	<code>X = X * Y;</code>
<code>/=</code>	<code>X /= Y;</code>	<code>X = X / Y;</code>
<code>%=</code>	<code>X %= Y;</code>	<code>X = X % Y;</code>

El uso de los operadores de asignación compuestos se ilustra en el Programa 1.9.

Programa 1.9

```
#include <stdio.h>

main()
{
    int numero = 10; /* Valor del numero para el ejemplo. */

    numero += 5;
    printf("Valor del numero += 5 es %d\n", numero);
    numero -= 3;
    printf("Valor del numero -= 3 es %d\n", numero);
    numero *= 3;
    printf("Valor del numero *= 3 es %d\n", numero);
    numero /= 5;
    printf("Valor del numero /= 5 es %d\n", numero);
    numero %= 3;
    printf("Valor del numero %= 3 es %d\n", numero);
}
```

La ejecución del Programa 1.9 produce:

```
Valor del numero += 5 es 15
Valor del numero -= 3 es 12
Valor del numero *= 3 es 36
Valor del numero /= 5 es 7
Valor del numero %= 3 es 1
```

Observe la última función `printf()` del programa. Para obtener una impresión del signo %, se ha usado un doble %.

Conclusión

En esta sección se han presentado los operadores aritméticos básicos de C. Aquí habrá aprendido el significado de un operador de asignación, así como los operadores de

asignación compuestos y el orden de prioridad de las operaciones. Compruebe sus conocimientos de esta sección haciendo la siguiente sección de repaso.

Repaso de la Sección 1.8

1. Indique los operadores aritméticos más frecuentemente usados en C.
2. En C, ¿qué hace la división entera con el resto? ¿Cuál es la importancia de esto?
3. En C, ¿se permite lo siguiente: `3 - 2 = resultado`; ? Explíquelo.
4. ¿Qué significa la prioridad de una operación?
5. Dé un ejemplo de un operador compuesto de asignación usado en C.

1.9. MÁS `printf()`

Esta sección presenta algunos detalles más acerca de la función `printf()`. Ya se ha usado `\n` como parte de esta función y se ha indicado brevemente que produce un retorno de carro y una nueva línea. En esta sección aprenderá más acerca de la potencia de la función `printf()`.

Secuencias de escape

`\n` es un ejemplo de una **secuencia de escape** que puede ser usada por la función `printf()`. El símbolo barra invertida o barra inclinada inversa *backslash* (`\`) se denomina habitualmente carácter de escape. Se puede pensar en una secuencia de escape usada en la función `printf()` como un escape de la interpretación normal de una cadena de caracteres. Esto significa que el siguiente carácter después del `\` tiene un significado especial, como se puede ver en la Tabla 1.9.

Los ejercicios interactivos de este capítulo le permitirán practicar con el uso de las secuencias de escape de `printf()` en su sistema.

Tabla 1.9. Secuencias de escape

Secuencia	Significado
<code>\n</code>	Nueva línea (salto de línea y principio de la siguiente)
<code>\t</code>	Tabulador
<code>\b</code>	Retroceder un carácter
<code>\r</code>	Retorno de carro
<code>\f</code>	Salto de página
<code>\'</code>	Comilla simple
<code>\"</code>	Comilla doble
<code>\\</code>	Barra inclinada inversa (invertida) <i>Backslash</i>
<code>\xdd</code>	Código ASCII en hexadecimal
<code>\ddd</code>	Código ASCII en octal

Nota: La comilla doble y la barra inclinada inversa *backslash* pueden ser impresos poniendo delante una barra invertida *backslash*.

Especificadores de ancho de campo

La función `printf()` le permite formatear a su salida. Recuerde de los programas anteriores que cuando se imprimía la salida de un tipo `float` aparecía como: 16.000000; se imprimían los seis ceros de la parte decimal aunque no se necesitaran. La función `printf()` proporciona **especificadores de ancho de campo** de forma que se pueda controlar como aparecerán en el monitor los valores que se imprimen. La sintaxis es:

`%<ancho>.<dígitos>F`

Donde

`%` = Indicador de formato.

`<ancho>` = Anchura total del campo.

`<dígitos>` = Número de dígitos a la derecha del decimal.

`F` = Especificador de formato.

Por ejemplo, la sentencia:

```
printf ("El numero %5.2f lo usa.", 6.0);
```

imprimiría:

```
El numero      6.00 lo usa.
```

Observe que el ancho inicial indica cómo se justifica el número (cinco espacios) y luego se especifica el número de dígitos que siguen al punto decimal. Para ver ejemplos adicionales, repase el Programa 1.5.

Conclusión

En esta sección se han presentado más detalles importantes de la función `printf()`. Ha visto otras secuencias de escape que pueden usarse con la función `printf()` y también algunos especificadores de formato. Compruebe que ha comprendido lo expuesto en esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 1.9

1. Explique el uso de una secuencia de escape en la función `printf()`.
2. ¿Cómo se llama a veces al carácter *backslash* (`\`) en la función `printf()`?
3. Indique tres secuencias de escape usadas en la función `printf()`.
4. ¿Qué es un especificador de ancho de campo tal y como se usa en la función `printf()`?

1.10. LEYENDO LA ENTRADA DEL USUARIO

Presentación

La potencia real de un programa técnico escrito en C es su habilidad para interactuar con el usuario. Esto significa que el usuario del programa puede introducir valores de

variables. Como podrá adivinar, hay una función C predefinida que permite que esto ocurra.

La función `scanf()`

La función `scanf()` es una función predefinida que permite a los programas leer la entrada del usuario desde el teclado. Se puede pensar en ella como en la opuesta a la función `printf()`. Su uso se ilustra en el Programa 1.10.

```

Programa 1.10 #include <stdio.h>
               /*           Recogiendo la entrada de usuario.           */
               main()
               {
                   float valor;      /* Un número introducido por el usuario. */

                   printf("Introduzca un numero => ");
                   scanf("%f", &valor);
                   printf("El valor es => %f", valor);
               }

```

Cuando se ejecute el Programa 1.10, la salida aparecerá de la siguiente manera (asumiendo que el usuario del programa introduce el valor 23) :

```

Introduzca un numero => 23
El valor es => 23.000000

```

Observe que la función `scanf()` tiene un formato similar al de la función `printf()`. Primero, contiene `%f`, entre comillas. Esto le dice al programa que se espera recibir un valor de tipo real. A continuación, se indica el identificador de la variable donde se almacenará el valor. Esto se hace usando una coma fuera de las comillas y luego un `&` (signo *ampersand*) inmediatamente seguido del nombre del identificador de la variable (`&valor`). Ahora, el valor que introduzca el usuario será el valor de la variable. Observe también que como resultado de usar la función `scanf()` se produce un retorno de carro.

Especificadores de formato

Los especificadores de formato de la función `scanf()` son similares a los usados para la función `printf()`. Esto se ilustra en la Tabla 1.10.

Debería observarse que tanto el especificador `%f` como el `%e` pueden ser usados para aceptar notación exponencial o decimal.

Tabla 1.10. Especificadores de formato de `scanf()`

Especificador	Significado
%c	Un carácter individual.
%d	Entero decimal con signo.
%e	Notación exponencial.
%f	Notación en coma flotante.
%o	Entero sin signo en base octal.
%u	Entero sin signo en base decimal.
%x	Entero sin signo en base hexadecimal.

La función `scanf()` puede aceptar más de una entrada con una sola sentencia, como se muestra debajo:

```
scanf ("%f%d%c", &numero_1, &numero_2, &character);
```

En el caso precedente, la variable `numero_1` aceptará un tipo `float`, la variable `numero_2` un tipo `int` y `character` un tipo `char`. En este caso el usuario del programa debería teclear tres valores separados por blancos. Por ejemplo:

```
52.7 18 t
```

Debido a que es muy fácil que el usuario del programa cometa errores introduciendo los datos de esta manera, la petición de múltiples entradas con un único `scanf()` no se usará en este texto.

Un programa de aplicación

Ya sabe como introducir datos, hacer cálculos básicos y mostrar los resultados. El Programa 1.11 calcula el voltaje que cae en una resistencia cuando los valores de la intensidad y la resistencia son conocidos. La relación matemática es:

$$\text{Voltaje} = \text{Intensidad} \times \text{Resistencia}$$

Programa 1.11 `#include <stdio.h>1`

```
/*                                Ley de Ohm                                */
main()
{
    float voltaje;                /* Valor del voltaje.          */
    float intensidad;            /* Valor de la intensidad.     */
    float resistencia;           /* Valor de la resistencia.    */

    printf("Introduzca la intensidad en amperios => ");
    scanf("%f", &intensidad);
    printf("Introduzca la resistencia en ohmios => ");
    scanf("%f", &resistencia);
```

```

    voltaje = intensidad * resistencia;    /* Calcula el voltaje. */
    printf("El valor del voltaje es %f voltios", voltaje);
}

```

Asumiendo que el usuario introducirá los valores de 3 para la intensidad y 4 para la resistencia, la ejecución del Programa 1.11 daría:

```

Introduzca la intensidad en amperios => 3
Introduzca la resistencia en ohmios => 4
El valor del voltaje es 12.000000 voltios

```

Hay varios puntos clave a observar en lo que concierne a este programa:

- Todas las variables han sido declaradas y se ha comentado algo de cada una de ellas.
- Cada función `scanf()` ha usado `%f` para indicar que la entrada sería de tipo `float` y `&variable` para indicar en qué variable se almacenaría el valor de entrada.
- El cálculo real ha sido comentado.
- La función `printf()` ha usado `%f` para indicar que la salida numérica sería de tipo `float` y el identificador de variable cuyo valor debía ser impreso ha sido indicado al final de las comillas.

Lo que ha visto en el programa anterior es un problema fundamental en tecnología que ha sido resuelto usando el lenguaje C. Es un problema muy sencillo que puede ser fácilmente resuelto en una calculadora de bolsillo. Pero, por ahora, la cuestión es mantener los problemas sencillos para no entorpecer la comprensión del lenguaje C. A medida que progrese con el texto, los programas serán mucho más potentes.

Uso de la notación E

Alguna mención debería hacerse acerca del uso de **notación E (exponencial)** en C. Como se dijo antes, C acepta la notación E para los tipos reales. Esto es válido tanto para la entrada como para la salida. Por ejemplo, en el último programa, el usuario del programa podría haber introducido valores más prácticos como 0,003 amperios para la intensidad y 2000 ohmios para la resistencia. Esto podría haberse hecho usando la notación E:

```

Introduzca la intensidad en amperios => 3E-3
Introduzca la resistencia en ohmios => 2E3
El valor del voltaje es 6.000000 voltios

```

Es importante señalar que C acepta tanto la E mayúscula como la e minúscula para este tipo de representación de los datos. Hay otras funciones C, tales como `gets()`, `atoi()` y `atof()`, que pueden ser usadas para realizar conversiones. Examinaremos estas funciones cuando aprendamos algo sobre las operaciones con cadenas de caracteres.

Conclusión

Esta sección le ha situado en el punto en que puede empezar a desarrollar algunos programas tecnológicos básicos usando el lenguaje C. En la próxima sección aprenderá a realizar más funciones aritméticas usando el lenguaje C. Este será un paso importante que le permitirá manejar casi cualquier tipo de fórmula tecnológica. Compruebe que ha comprendido esta sección haciendo los siguientes ejercicios de repaso.

Repaso de la Sección 1.10

1. Defina el propósito de la función `scanf()`.
2. ¿Cómo sabe la función `scanf()` qué identificador de variable debe usar para introducir los datos?
3. ¿Se produce un retorno de carro a una nueva línea cuando se usa la función `scanf()`?
4. ¿Cuántos valores reales por petición deberían ser introducidos por el usuario en un programa C?
5. Explique qué debe hacerse para tener valores impresos en la pantalla con notación E.

1.11. IMPLEMENTACIÓN Y DEPURACIÓN DE PROGRAMAS: ERRORES DE PROGRAMACIÓN FRECUENTES

Presentación

El material de esta sección está diseñado para ayudarle a minimizar algunos de los más frecuentes errores que cometen los programadores de C novatos.

Tipos de mensajes de error

Dependiendo del tipo de compilador usado, se recibirán distintos tipos de mensajes de error. La mayoría de los compiladores producen tres tipos de mensajes de error:

1. Mensajes de errores fatales
2. Mensajes de errores de compilación
3. Mensajes de aviso

Un *mensaje de error fatal* termina inmediatamente el proceso de compilación. No comprueba ningún error más y provoca una parada con un mensaje que indica cuál puede ser la causa más frecuente de este problema. La mayoría de los sistemas C moverán el cursor a la línea de código fuente donde se «piensa» que ha ocurrido el error. Puede ocurrir, dependiendo de la naturaleza del error, que donde se haya puesto el cursor no sea la localización real del error. Debe recordar que ningún esquema de localización de errores es perfecto y, además, que no todos los posibles errores son predecibles. El compilador le estará dando su suposición más probable de lo que está causando el error. Es asunto suyo determinar exactamente dónde se ha producido el error y cómo corregirlo.

Un *mensaje de error de compilación* es el resultado de un error menos severo. En estos casos, el compilador tratará de continuar la compilación y producirá otros mensajes de compilación si encuentra más errores de este tipo. En algunos casos, puede no ser capaz de continuar el proceso de compilación y, como resultado, se producirá un mensaje de error fatal. En cualquier caso, no se produce código objeto y el proceso termina con la obtención de una lista de mensajes de error. Se puede usar esta lista como ayuda para determinar la situación de los errores en el código fuente.

Los *mensajes de aviso* son el resultado de errores de programación y permiten que el programa se compile y enlace, pero, en algunos sistemas C, no permitirán que su programa se ejecute. Estos mensajes de aviso se muestran al programador como una referencia que le permita determinar los problemas en su código fuente.

Sensibilidad a mayúsculas y minúsculas

Recuerde que un aspecto importante de la programación en C es que los identificadores son sensibles a su tipografía. Esto significa que C distingue entre letras mayúsculas y minúsculas. Esta sensibilidad se aplica a todos los identificadores. Por ejemplo, el Programa 1.12 no se ejecutará porque el programador usa mayúsculas para la primera letra de las variables cuando las declara, pero las olvida cuando las usa en el programa.

```
Programa 1.12 /*      Ejemplo de mayúsculas y minúsculas      */
main()
{
    float Uno = 2; /* Variable del programa inicializada. */
    float Otro;    /* Variable del programa. */

    otro = 2 * uno;
    /* Este programa no ejecutará! */
}
```

Hay dos errores en el Programa 1.12, ambos del mismo tipo. Las variables declaradas `Uno` y `Otro` usan letras mayúsculas al principio. Sin embargo, en el cuerpo del programa, el programador las ha olvidado.

```
otro = 2 * uno;
```

Además, el compilador pensará que éstos son nuevos identificadores que no han sido declarados.

El punto y coma

Otro error muy frecuente de los programadores de C novatos es la omisión del punto y coma. En C, el punto y coma identifica el fin de una sentencia o instrucción. El punto y coma es realmente parte de las sentencias de C y debe ser incluido en ellas. Un punto y coma perdido siempre causa que el programa no se ejecute. Sin embargo, un

punto y coma perdido puede confundir al compilador acerca de la causa del problema. Incluso se pueden obtener mensajes de error producidos por esta causa que no mencionan para nada al punto y coma. Una buena regla a tener en cuenta es que si el mensaje de aviso no tiene sentido hay que comprobar los puntos y coma en primer lugar.

Por ejemplo, al Programa 1.13 le falta un punto y coma. Sin embargo, el mensaje de error (el tipo que se obtiene depende de su sistema) no será capaz de darse cuenta de que el problema viene de un punto y coma perdido.

```

Programa 1.13 /*      Punto y coma perdido.      */
main()
{
    float uno = 2    /* Variable del programa inicializada. */
    float otro;      /* Variable del programa. */

    otro = 2 * uno;
    /* Este programa no ejecutará! */
}

```

Observe que al Programa 1.13 le falta un punto y coma en la declaración de la variable uno.

La mejor regla a seguir es siempre llevar a cabo una buena inspección visual de su programa C antes de intentar compilarlo. Mire siempre si faltan puntos y comas.

Comentarios incompletos o anidados

Otra fuente común de errores en el código fuente es el uso de comentarios. Estos errores son habitualmente de dos tipos. En el primero, el usuario escribe un delimitador de inicio de comentario /* pero se olvida de escribir el delimitador de fin de comentario. Este error se puede ver en el Programa 1.14.

```

Programa 1.14 /*      Fin de un comentario olvidado. Comentarios Anidados      */
main()
{
    float constante_1 = 2;    /* Variable del programa inicializada.
    /*
    float variable_1;        /* Variable del programa.

    /* Esto es un comentario ....
    variable_1 = constante_1 + constante_1; /* Otro comentario */
        fin del comentario anidado */

    /* Este programa no ejecutará! */
}

```

El Programa 1.14 no ejecutará porque se ha olvidado escribir un delimitador de fin de comentario `*/` para el comentario: `/* variable del programa.` Sin embargo, esto confunde tanto al compilador que cuando muestra un mensaje de aviso, éste no indica que el problema está relacionado con el comentario. Esto introduce otra buena regla. Cuando se obtengan mensajes de error que no se comprenden, después de comprobar que todas las sentencias tienen su punto y coma, compruebe que todos los delimitadores de final de comentario están presentes.

El Programa 1.14 muestra también un ejemplo de comentarios anidados, la segunda fuente potencial de errores en los comentarios. Normalmente, este programa no ejecutará porque uno de los comentarios está contenido dentro de otro. Algunos compiladores le permiten configurar el entorno de ejecución de forma que se pueda anidar comentarios. Sin embargo, esta práctica no es recomendable y no se usará en este texto.

El Ejemplo 1.4 le permite practicar para desarrollar su habilidad visual para encontrar errores relacionados con los comentarios.

Ejemplo 1.4 Determine si hay algún error en el siguiente programa que no le permitirá ejecutar. Si es así, indique lo que haría para corregirlo(s).

Programa 1.15

```

/*      ¿Hay aquí un error?      */
main()
{
    float Este_valor = 15;        /* Constante del programa. */
    float Ese_valor;              /* Variable del programa. */

    Ese_valor = este_valor + este_valor
}

```

Solución

Realice siempre una buena inspección visual de su código fuente antes de intentar compilarlo. Hacer esto le entrenará para localizar sus propios errores rápidamente. La habilidad para detectar errores se adquiere con gran cantidad de práctica.

Programa 1.15. Errores:

Se han cambiado las mayúsculas y minúsculas en la variable inicializada:

```
Ese_valor = este_valor + este_valor
```

Se ha perdido un punto y coma al final de la sentencia de C:

```
Ese_valor = este_valor + este_valor
```

El estilo de colocación de las llaves no es erróneo, pero no se aconseja poner las llaves en las mismas líneas que otras sentencias porque no son fáciles de ver. Una buena práctica de programación en C aconseja dedicar una línea completa para cada llave.

Conclusión

En esta sección se han presentado algunos de los errores de programación más frecuentes cometidos por los programadores de C novatos. Hemos visto que una buena práctica de la programación requiere que se comprueben cuidadosamente los programas para ver si faltan puntos y comas, si las mayúsculas y minúsculas están bien usadas y si faltan o sobran delimitadores de comentarios. Compruebe que ha comprendido lo aprendido en esta sección haciendo los siguientes ejercicios de repaso.

Repaso de la Sección 1.11

1. Indique los tres tipos de mensajes de error que se encuentran en C.
2. ¿Qué tipo de mensajes de error terminará el proceso de compilación?
3. Explique cómo la sensibilidad a mayúsculas y minúsculas puede hacer que un programa no se ejecute.
4. ¿Por qué es conveniente mirar si faltan puntos y comas si no se entiende la causa de un mensaje de error?
5. ¿Qué es un comentario anidado? ¿Es legal?

1.12. PROGRAMA DE APLICACIÓN: CONVERSIÓN DE TEMPERATURA

Presentación

El tema de esta sección es el diseño de un programa que convierta una lectura de temperatura de grados Fahrenheit a grados Celsius. Ésta es una conversión muy fácil de realizar en muchas calculadoras de bolsillo. Se usa aquí para ilustrar algunos elementos fundamentales en el diseño de un programa. El problema se mantiene sencillo intencionadamente de forma que no domine el proceso de desarrollo del programa, pero, aun así, el problema tiene el rigor suficiente como para necesitar la mayoría del material nuevo que se ha presentado en este capítulo.

Primer paso: definiendo el problema.

El primer paso, y el más importante, en el diseño de un programa es definir el problema por escrito. Esta definición debe incluir algunas especificaciones acerca del programa en sí. Estas especificaciones son:

- Propósito del programa.
- Entrada que se necesita (fuente).
- Proceso sobre la entrada
- Salida que se necesita (destino)

El problema puede definirse como sigue:

- Propósito del programa: convertir una temperatura leída de grados Fahrenheit a grados Celsius.

- Entrada que se necesita: temperatura en grados Fahrenheit (fuente: el teclado).
- Proceso sobre la entrada: temperatura Celsius = $5/9 \times (\text{temperatura Fahrenheit} - 32)$.
- Salida que se necesita: temperatura en grados Celsius (destino: la pantalla).

Después de leer esta definición, usted (y otros que puedan trabajar con usted) conocerá claramente el propósito del programa. Es evidente que la información para el programa será recibida desde el teclado y que consistirá en una lectura de temperatura en grados Fahrenheit. El programa procesará esta entrada de acuerdo a una fórmula muy específica. La salida será impresa en la pantalla del computador. Será la temperatura expresada en grados Celsius.

Para este sencillo problema, esta información puede parecer obvia. Sin embargo, para programas más complejos donde alguna información se recibirá desde el teclado mientras que otra se recibirá del disco, el proceso puede no ser tan obvio. Definirlo por escrito de la forma anterior asegura que usted y las personas para las que diseña el programa están de acuerdo en lo que debe hacerse —y lo que es igual de importante— lo que no debe hacerse.

Estableciendo los bloques del programa

Una vez completada la definición del problema, el siguiente paso es establecer los bloques («comentarlos») del programa C. Esto se ilustra en el Programa 1.16.

Programa 1.16 /*

Programa: Conversión de grados Fahrenheit a grados centígrados
Desarrollado por: Cualquier programador

Descripción: Este programa convierte una temperatura leída en grados Fahrenheit a su equivalente en grados centígrados.

Variables: ninguna

Constantes: ninguna
*/

```
main()
{
    /* Explicar el programa al usuario. */

    /* Obtener el valor Fahrenheit del usuario. */

    /* Realizar los cálculos. */

    /* Mostrar la respuesta. */
}
```

El Programa 1.16 es un esbozo que le da al programa un título, proporciona el nombre del programador y describe lo que hace el programa. El resto es un esbozo de las principales secciones del programa. Actuará como una guía para desarrollar el código. Lo más importante es que el esbozo será realmente parte del programa —los comentarios para cada una de las partes principales del programa. Una vez que el esbozo esté completo, se almacena en disco y se saca una copia impresa del mismo. Esta copia impresa puede servir para verificar que el esbozo se ajusta a la intención original del programa y puede servir como una fuente de documentación.

Observe que las partes esenciales `main()`, `{` y `}` están incluidas, de forma que el programa compilará. El programa debería ya compilarse en este punto para asegurarse de que las secciones comentadas no contienen errores (tales como comentarios anidados, según se explicó en la sección anterior).

El paso siguiente

Una vez que se ha esbozado el programa, el siguiente paso es desarrollar cada sección, de una en una. Este proceso reduce mucho el tiempo dedicado a la depuración del programa. Para este programa, el primer paso es codificar la explicación del programa, compilarla y observar la salida. Este paso se muestra en el Programa 1.17. En este programa se presenta una nueva función de C llamada `puts()`. Esta función genera un salto de línea automático. Es una función muy práctica cuando sólo se quiere imprimir texto. Esta función necesita la instrucción `#include <stdio.h>` al principio del programa.

Cuando se ejecute el Programa 1.17, el bloque de explicación produce la siguiente salida:

```
Este programa convertira una temperatura leida en
grados Fahrenheit a su equivalente en
grados centigrados.
```

```
Introduzca la temperatura en grados Fahrenheit
y el programa hara el resto.
```

La cuestión es desarrollar el programa paso a paso, lo que reduce la frustración potencial de tratar de descubrir errores en un programa grande. Si hubiese un error en una etapa temprana, el programador sabría que está confinado a esta parte del programa. Cuando se han eliminado los errores de esta primera parte se puede continuar con el siguiente paso hasta completar el programa. En todo caso, si hay errores deberán estar confinados al código más reciente que se ha introducido en el programa (la excepción a esto se produce cuando el programa usa código antiguo —pero, aun así, este método será el preferido).

Terminando el programa

El siguiente paso es codificar y compilar la siguiente parte del esbozo, en la cual se lee introduce el valor de la temperatura. Cuando se hace esto en el Programa 1.17, la ejecución dará lo siguiente:

```
Este programa convertira una temperatura leida en
grados Fahrenheit a su equivalente en
grados centigrados.
```

```
Introduzca la temperatura en grados Fahrenheit
y el programa hara el resto.
Introduzca la temperatura en grados Fahrenheit =>
```

Si se introduce un valor en este punto del programa se puede comprobar, de nuevo, que no hay errores de compilación ni de lectura de datos.

El siguiente paso es codificar y compilar la siguiente parte del esbozo, en la cuál se procesa el valor de la temperatura y se convierte a grados Fahrenheit. Para completar este punto hacen falta variables, que deben ser añadidas al bloque de declaración. Como antes, estas variables serán de tipo `float`. Cada variable está comentada, como se requiere en una buena práctica de la programación. Además, una buena práctica de programación exige que las variables se incluyan en el bloque del programador. Observe que se usa el signo en la fórmula para indicar que está realizando una multiplicación.

El último paso de programa se hace cuando se muestran los resultados de los cálculos anteriores. Cuando se realiza esta parte se ha completado el programa, una versión de cuál puede verse en el Programa 1.17.

```
Programa 1.17 #include <stdio.h>
/*
Programa: Conversión de grados Fahrenheit a grados centigrados
Desarrollado por: Cualquier programador

Descripción: Este programa convierte una temperatura leida en grados
Fahrenheit a su equivalente en grados centigrados.

Variables:
    Fahrenheit = Temperatura en grados Fahrenheit
    centígrados = Temperatura en grados Celsius

Constantes: ninguna
*/

main()
{
```

```

float temp_fahrenheit;
float temp_centigrados;

/* Explicar el programa al usuario. */
puts("");
puts("Este programa convertira una temperatura leida en");
puts("grados Fahrenheit a su equivalente en");
puts("grados centigrados.");
puts("");
puts("Introduzca la temperatura en grados Fahrenheit");
puts("y el programa hara el resto.");

/* Obtener el valor Fahrenheit del usuario. */
puts("");
printf("Introduzca la temperatura en grados Fahrenheit => ");
scanf("%f", &temp_fahrenheit);

/* Realizar los cálculos. */
temp_centigrados = 5/9 * (temp_fahrenheit - 32);

/* Mostrar la respuesta. */
puts("");
printf("Una temperatura de %f grados Fahrenheit \n",
      temp_fahrenheit);
printf("es igual a %f grados centigrados. \n", temp_centigrados);
}

```

La última parte del programa, la sección «Mostrar la respuesta», ha sido codificada y el programa debe compilarse de nuevo para comprobar si existen errores de programación. Observe que se usa el `%f` para imprimir los tipos reales y que el `\n` se usa para indicar que hay que imprimir en una nueva línea. Una vez que se ha hecho todo esto, queda todavía un paso final muy importante por hacer.

Comprobación de la salida

Hasta este punto del desarrollo del programa, se ha comprobado que no hay errores de compilación. Sin embargo, para estar seguros de que no hay errores de ejecución, hay que comprobar el programa para distintos valores de entrada. Los valores de salida obtenidos se comprueban después para ver su exactitud. Para ello se introducen varios valores (negativos, positivos y cero) y los resultados se comprueban por varias personas usando una calculadora. Hacer esto asegura que el programa fue codificado correctamente para producir los resultados pretendidos por los diseñadores del programa.

Una ejecución de prueba del programa da:

```

Este programa convertira una temperatura leida en
grados Fahrenheit a su equivalente en
grados centigrados.

```

```

Introduzca la temperatura en grados Fahrenheit
y el programa hara el resto.
Introduzca la temperatura en grados Fahrenheit => 212

```

```

Una temperatura de 212.000000 grados Fahrenheit
es igual a 0.000000 grados centigrados.

```

¡Como se demuestra en esta prueba hay un error de ejecución en el programa! Por eso es importante comprobar el programa con distintos valores y comprobar los resultados. El hecho de que no haya errores de compilación no significa que el programa haga lo que se pretende de él. Que no haya errores de compilación sólo significa que el código está pasable, pero no indica nada sobre el diseño del programa. La razón por la que se obtiene 0.000000 es porque se están usando números enteros en una división (5/9). Recuerde que cuando se discutieron los enteros en este capítulo se dijo que la división de enteros no deja resto (no puede, porque son números enteros). Por tanto, 5/9 devolverá un valor 0. Este 0 se multiplica por el término (Fahrenheit - 32), que otra vez vuelve a dar cero (un número multiplicado por 0 da 0). Por tanto, independientemente del valor en grados Fahrenheit que se introduzca, el resultado será siempre cero. Para corregir este error, los enteros de la división 5/9 deben convertirse a valores decimales mediante la simple adición a cada número de un punto decimal seguido por un cero:

```
centigrados = 5.0/9.0 * (Fahrenheit - 32);
```

Conclusión

Es esta sección se le ha presentado su primera oportunidad real de desarrollar un programa técnico en C. Se le ha presentado el concepto de hacer una descripción previa por escrito de los requisitos del programa. A continuación se ha visto como hacer un esbozo del programa real usando comentarios. A partir de este esbozo, se ha desarrollado y probado cada sección por separado. Una vez que el programa ha sido completamente codificado, se ha probado su exactitud para distintos valores de entrada.

Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 1.12

1. ¿Cuál es el primer paso en el desarrollo de un programa?
2. Indique qué elementos deberían ser incluidos en la definición del problema.
3. Indique el primer paso en la codificación real del programa.
4. Explique el proceso usado para desarrollar el programa final.

1.13. PROGRAMAS DE APLICACIÓN ADICIONALES

A continuación veremos tres programas adicionales diseñados para mostrar cómo se pueden realizar cálculos prácticos en C. Muchas veces se necesita un programa sencillo que realice una tarea por nosotros. Por ejemplo, nosotros no solemos ser muy buenos haciendo conversiones o problemas matemáticos complejos (tal como «¿cuántas horas hay en 25 años?») de memoria.

El Programa 1.18 es un ejemplo de cómo implementar en C conversiones o cálculos. Calcula la resistencia equivalente de dos resistencias colocadas en paralelo.

Programa 1.18 `#include <stdio.h>`

```
main()
{
    float R1;      /* Resistencia 1.          */
    float R2;      /* Resistencia 2.          */
    float REQ;     /* Resistencia equivalente. */

    printf("Introduzca el valor de la resistencia 1 => ");
    scanf("%f",&R1);
    printf("Introduzca el valor de la resistencia 2 => ");
    scanf("%f",&R2);
    printf("\n");
    REQ = (R1 * R2) / (R1 + R2);
    printf("La resistencia equivalente es %f\n",REQ);
}
```

Si se ejecuta el Programa 1.18 con R1 y R2 igual a 1000 y 4000 ohmios respectivamente, se obtiene:

```
Introduzca el valor de la resistencia 1 => 1000
Introduzca el valor de la resistencia 2 => 4000
```

```
La resistencia equivalente es 800.000000
```

¿Puede pensar en una forma de modificar el programa anterior para resolver problemas de tipo similar, tal como convertir pulgadas cuadradas a metros cuadrados o encontrar la resistencia equivalente de tres resistencias colocadas en paralelo?

Otros tipos de cálculos pueden necesitar el uso de funciones matemáticas especiales además de la suma, resta, multiplicación y división. Por ejemplo, la pregunta «¿cuánto es 200 elevado a la séptima potencia?» requiere habilidades matemáticas que poca gente puede llevar a cabo sin un lápiz y un papel. Afortunadamente, C viene equipado con muchas funciones matemáticas adicionales. El Programa 1.19 utiliza una de estas funciones para calcular el monto total de una inversión en un periodo de años usando la siguiente fórmula:

$$\text{Valor acumulado} = \text{Cantidad invertida} * \text{Interés}^{\text{Años}}$$

Puesto que se necesita elevar un número a una potencia, es necesario usar una función diseñada para este propósito. La función que se necesita es `pow()` y se encuentra en el archivo de definiciones `math.h`. Examine el Programa 1.19 para ver cómo se usa la función `pow()` en los cálculos.

Programa 1.19

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

main()
{
    float cantidad;    /* Dinero a invertir.      */
    float interes;     /* Tasa de interes anual.    */
    float annos;       /* Numero de años.          */
    float total;       /* Cantidad total acumulada. */

    printf("Introduzca la cantidad a invertir => ");
    scanf("%f",&cantidad);
    printf("Introduzca la tasa de interes anual => ");
    scanf("%f",&interes);
    printf("Introduzca el numero de annos => ");
    scanf("%f",&annos);
    printf("\n");
    interes /= 100.0;
    total = cantidad * pow((1.0 + interes), annos);
    printf("La cantidad total acumulada es %f\n", total);
}
```

Algunas otras funciones útiles que se pueden encontrarse en `math.h` son:

```
abs()   atof()  cos()  exp()  log()
log10() pow()   sin()  sqrt() tan()
```

El Programa 1.20 usa la función `sqrt()` de `math.h` para calcular la longitud de la hipotenusa de un triángulo rectángulo.

Programa 1.20

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

main()
{
    float ladoa, ladob, ladoc;

    printf("Introduzca el lado A => ");
    scanf("%f",&ladoa);
    printf("Introduzca el lado B => ");
    scanf("%f",&ladob);
```

```

        ladoc = sqrt(ladoa*ladoa + ladob*ladob);
        printf("\nLa longitud de la hipotenusa es %5.1f",ladoc);
    }

```

Podría merecer la pena modificar el Programa 1.20 para que calculara también todos los ángulos interiores del triángulo. Esto requeriría el uso de las funciones `sin()`, `cos()` y `tan()` de `math.h`. Observe que estas funciones necesitan que el ángulo de entrada esté en radianes.

1.14. PROGRAMACIÓN EN ANSI C

En 1989, el Instituto Americano de Estándares Nacionales (ANSI) definió un estándar que describía el lenguaje C. Este estándar garantiza que si se escribe el código C de acuerdo al estándar ANSI C, ese código debería compilar y ejecutar en cualquier máquina que proporcione soporte para el ANSI C. Incluso antes de que el lenguaje C fuese estandarizado, ya era considerado *transportable*. Esto significa que un programa C escrito en un tipo de computador puede ser compilado y ejecutado en un tipo distinto de computador. Por ejemplo, un programa C escrito en un computador personal debería poderse recompilar y ejecutar adecuadamente en un VAX, una estación de trabajo SUN, un computador grande (*mainframe*) IBM o cualquier otro tipo de computador que use ANSI C. Desde que se adoptó el ANSI C por la comunidad de programadores, el nivel de transportabilidad se ha elevado. El estándar ANSI C cubre las funciones más básicas del lenguaje C, tales como entrada desde teclado y salida a pantalla, y funciones complejas tales como entrada y salida a archivos.

Cada tipo de computador (también llamado *plataforma*) utiliza un hardware diferente como base. Por ejemplo, algunos computadores personales usan la arquitectura 80x86 de Intel (8086 hasta el Pentium), mientras que otros usan la 680x0 de Motorola (68000 hasta 68040). Cada función de ANSI C necesita conocimiento acerca del hardware subyacente en el computador. El estándar ANSI C garantiza que las diferencias en el hardware no afectarán a la forma en que se escriben y ejecutan los programas. Por ejemplo, el sistema operativo Windows NT ha sido transportado a varias plataformas hardware sin tener que reescribir ningún programa escrito en el lenguaje C.

Los programas de este libro seguirán el estándar ANSI C.

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido aquí para permitirte adquirir una valiosa experiencia y, lo que es más importante, para tener una realimentación inmediata de lo que hacen los conceptos y órdenes presentados en este capítulo. Además son divertidos.

Ejercicios

1. Prediga cuál será la salida del Programa 1.21 y luego pruébelo.

Programa 1.21

```
#include <stdio.h>
main()
{
    printf("El numero es %d\n", 15);
    printf("El numero es %i\n", 15);
    printf("El numero es %x\n", 15);
    printf("El numero es %o\n", 15);
    printf("El numero es %e\n", 15.0);
    printf("El numero es %E\n", 15.0);
}
```

2. El Programa 1.22 es muy divertido. Intente predecir su salida y, luego, pruébelo.

Programa 1.22

```
#include <stdio.h>
main()
{
    char letra;

    letra = 'b';
    printf("Esta es %d or %c or %x.", letra, letra, letra);
}
```

3. Calcule el Programa 1.23 con lápiz y papel. Después, pruébelo ¿Coinciden sus resultados con los del computador?

Programa 1.23

```
#include <stdio.h>
main()
{
    float resultado = 10;

    resultado = 2 * (3 + 5)/8 - 3;
    printf("El resultado es %f.\n", resultado);
}
```

4. Intente predecir la salida para cada caso previsto en el Programa 1.24. Después ejecute el programa. Asegúrese de escribir lo que le ha devuelto el programa en sus notas.

Programa 1.24

```
#include <stdio.h>
float numero_1 = 125.738;
main()
{
    printf("En notacion decimal 125.738 = %d\n", numero_1);
    printf("En notacion real 125.738 = %f\n", numero_1);
    printf("En notacion cientifica 125.738 = %e\n", numero_1);
}
```

Autoevaluación

DIRECTRICES

El Programa 1.25 ha sido programado en C por un estudiante sin experiencia. Puede contener algunos errores. Responda a las cuestiones siguientes referidas a este programa.

Programa 1.25 #include <stdio.h>

```

/*
Programa: Ley de Ohm
Desarrollado por: Un buen programador

Descripción: Este programa soluciona la intensidad en un circuito. El
              usuario debe introducir el valor del voltaje en el
              circuito y la resistencia del circuito.

Variables:
    intensidad = Intensidad del circuito en amperios.
    voltaje    = Voltaje del circuito en voltios.
    resistencia = Resistencia del circuito en ohmios.

Constantes: ninguna
*/

main()
{
    /* Bloque de declaración */

    float voltaje;      /* Valor del voltaje.          */
    float intensidad;   /* Valor de la intensidad.      */
    float resistencia;  /* Valor de la resistencia.     */

    /* Explicar el programa al usuario. */
    puts("Este programa calcula el valor de la");
    puts("intensidad de un circuito en amperios.");
    puts("");
    puts("Debe introducir el valor del voltaje del circuito");
    puts("en voltios y la resistencia del circuito en ohmios");
    puts("");
    puts("");

    /* Obtención de los valores de entrada del usuario. */
    printf("Valor del voltaje del circuito = ");
    scanf("%f", &voltaje);
    printf("Valor de la resistencia del circuito = ");
    scanf("%f", &resistencia);

    /* Realizar los cálculos. */
    intensidad = voltaje / resistencia;    /* Calcular el voltaje.
*/

    /* Mostrar la respuesta. */

```

```

puts("");
puts("La intensidad en un circuito con una resistencia total");
printf("de %e ohmios y un voltaje total de %e\n", resistencia,
        voltaje);
printf("voltios es %e amperios.\n", intensidad);
}

```

Preguntas

1. ¿Compilará y ejecutará el programa en su sistema? Si no, ¿por qué no?
2. Explique lo que hace el programa ¿Cómo ha podido averiguarlo?
3. ¿Cuántas variables hay en el programa? ¿De qué tipo son? ¿Cómo lo ha averiguado?
4. Explique por qué se ha usado la función `puts()` para explicar el programa al usuario.
5. Explique cómo puede el usuario introducir los valores del voltaje y la resistencia ¿Cómo los ha averiguado?
6. ¿De qué manera se mostrarán los valores de salida? ¿Cómo lo ha averiguado?

Problemas de fin de capítulo

Conceptos generales

Sección 1.1

1. ¿Qué tipo de programa se usa para escribir el código fuente en C?
2. Explique qué programa convierte su código fuente en algo que entiende el computador.

Sección 1.2

3. ¿Con qué deben comenzar todos los programas en C?
4. Explique el propósito de los `/* */` usados en C.
5. ¿Cómo se indica el principio y el final de las instrucciones de un programa en el lenguaje C?

Sección 1.3

6. ¿Qué hace que un programa sea fácil de comprender, modificar y depurar?
7. Explique el propósito del bloque del programador.
8. ¿Qué contiene el bloque del programador?
9. ¿Es necesario tener un bloque de programador para que compile un programa C?

Sección 1.4

10. ¿Qué es un componente léxico en C?
11. Indique qué otros símbolos aparte de las letras del alfabeto inglés y los diez dígitos decimales del sistema de numeración arábigo pueden usarse para escribir un programa C.
12. Explique qué significa un componente léxico predefinido.

Sección 1.5

13. ¿Cuál es la principal diferencia en la forma de representar caracteres y cadenas de caracteres en la función `printf()`?
14. Indique que elemento especifica en la función `printf()` cómo se va a convertir, imprimir y dar forma a sus argumentos.
15. ¿Cuál es el nombre dado a los valores reales entre los paréntesis de una función?
16. ¿Cuántos argumentos debe contener como mínimo una función `printf()`?

Sección 1.6

17. ¿Cuál es el nombre que se da a una colección de declaraciones o sentencias independientes en C?
18. Indique con qué deben empezar todos los identificadores en C.
19. ¿Cuántos caracteres de un identificador reconoce C?
20. ¿Qué es un identificador?

Sección 1.7

21. Nombre los tres tipos de especificadores de tipo fundamentales en C.
22. ¿Cómo se reserva una cantidad específica de memoria en C para recibir un valor posteriormente?
23. En C, ¿qué evita que una nueva variable aparezca en su programa como resultado de un error de tipografía?
24. ¿Cómo se llama a la combinación de una declaración de variable con operador de asignación?

Sección 1.8

25. Defina una característica única de la división entera en C.

26. ¿Hay un orden en el que se deban realizar las operaciones aritméticas? ¿Cómo se denomina?
27. ¿Qué significa la sentencia de C `resultado *= 5;`?
28. ¿Cómo se denomina una sentencia como la del problema 27?

Sección 1.9

29. ¿Qué causa, en una función `printf()`, una importante diferencia en la interpretación de una cadena de caracteres?
30. Indique el carácter de la función `printf()` que es denominado carácter de escape.
31. Indique qué determina el número de dígitos a la derecha del punto decimal en un valor impreso con la función `printf()`.

Sección 1.10

32. Ponga un ejemplo del uso de la función `scanf()` para leer el valor de una constante `constante_1` declarada de tipo real.
33. Explique el propósito de la función `scanf()` según se ha descrito en este capítulo.
34. Ponga un ejemplo de cómo imprimir el valor de una constante `constante_1` en notación E.
35. Para la sentencia C:

```
scanf("%f", &valor);
```

¿en qué formato numérico puede el usuario del programa introducir valores?

Sección 1.11

36. ¿Cuáles son los tres tipos de errores de compilación usados en C?
37. Explique el efecto de un mensaje de error fatal.
38. ¿Cuáles son los tres errores más frecuentes provocados por los programadores novatos de C que se han presentado en este capítulo?
39. ¿Identifican siempre los mensajes de error el problema de su programa? Explíquelo.

Sección 1.12

40. Liste las cuatro áreas específicas, presentadas en este capítulo, que deberían ser definidas por escrito cuando se diseña un programa.
41. Indique el primer paso de la codificación de un programa.

Diseño de programas

Ya tiene información para desarrollar el código fuente de los siguientes programas. Para todos los programas que diseñe, se espera que siga los pasos descritos en la Sección 1.12.

42. Escriba un programa C que determine cuántos bits son necesarios para representar un entero sin signo.

Por ejemplo, se necesitan tres bits para representar el valor 7, cuatro bits para representar números entre el 8 y el 15, etc. Use las funciones `log()` y `log10()` en su programa. El usuario debe introducir el entero sin signo.

43. Construya un programa C que evalúa las siguientes expresiones:

$$N^2 \quad N^3 \quad 2^N \quad 3^N$$

44. Construya un programa que calcule la reactancia inductiva para una frecuencia en particular. El usuario del programa debe introducir el valor del inductor y de la frecuencia. La fórmula para calcular la reactancia inductiva es:

$$X_L = 2\pi fL$$

Donde:

X_L = Resistencia inducida en ohmios

f = frecuencia en hercios

L = Valor de la inductancia en henrios

45. Construya un programa C que convierta grados Celsius a grados Fahrenheit. La entrada del usuario es la temperatura Fahrenheit. La relación es:

$$F = (9/5)C + 32$$

Donde:

C = Temperatura en Celsius

F = Temperatura en Fahrenheit

46. Construya un programa C que calcule el área de un círculo. El usuario debe introducir el radio del círculo. Use la siguiente fórmula para el área:

$$A = \pi R^2$$

47. Escriba un programa C que calcule la longitud de los lados opuesto y adyacente de un triángulo rectángulo, dada la longitud de la hipotenusa y un ángulo (θ), como se ve en la Figura 1.3. El usuario debe introducir la longitud de la hipotenusa y el ángulo (en grados).

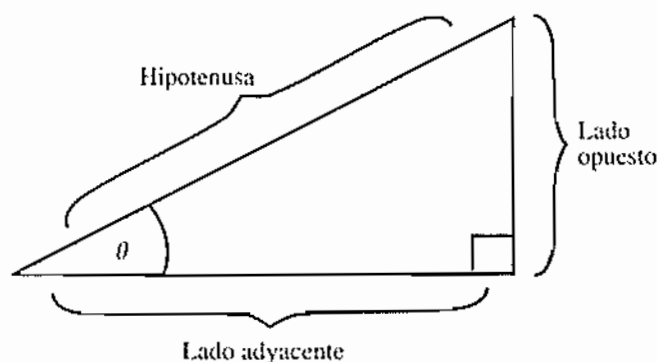


Figura 1.3. Diagrama para la pregunta 47.

Programación estructurada

Objetivos

Este capítulo le da la oportunidad de aprender lo siguiente:

1. Cómo reconocer un programa estructurado en bloques.
2. Cómo desarrollar un programa con estructura de bloques utilizando C.
3. Un teorema muy importante sobre programación.
4. El uso de las funciones de C en el desarrollo de un programa estructurado en bloques.
5. Cómo pasar valores entre funciones.
6. El significado y uso de un parámetro formal.
7. El significado y uso de un parámetro real.
8. El significado de preprocesamiento y las directivas del preprocesador.
9. El uso de la directiva `#define`.
10. Cómo desarrollar y grabar sus propios archivos de cabecera.
11. Cómo utilizar el diseño descendente en el desarrollo de un programa que resuelve un problema tecnológico.

Palabras clave

Bloque de programa
Estructura de bloques
Separador de bloques
Bloque secuencial
Bloque repetitivo
Bloque de selección
Función
Recursividad
Macro
Archivo de cabecera

Prototipo de una función
Lista de parámetros formales
Paso de valores
Parámetro formal
Parámetro real
Múltiples argumentos
Llamadas a funciones
Concatenación de componentes léxicos
Operaciones sobre cadenas de caracteres
Prólogo

Contenido

2.1. Concepto de bloque de programa	2.6. Implementación y depuración de programas: haciendo sus propios archivos de cabecera
2.2. Uso de funciones	
2.3. Dentro de una función C	
2.4. Uso de funciones	2.7. Programa de aplicación: Circuito RL en serie
2.5. Uso de la directiva #define	
-

Introducción

En el Capítulo 1 comenzó a desarrollar sus propios programas en C. Aprendió cómo utilizar C para leer valores del usuario de un programa, realizar operaciones sobre esos valores y mostrar los resultados por pantalla.

A partir de ahora, obtendrá información que le ayudará a desarrollar programas en C relacionados con la tecnología que no son fáciles de resolver con una simple calculadora científica de bolsillo. Esto significa que sus programas serán más grandes y realizarán muchos tipos de operaciones útiles. Ello hace necesario la introducción del concepto de programación estructurada, lo que le permitirá adquirir buenas técnicas de programación desde el principio del aprendizaje del lenguaje C. El resto de los capítulos de este libro harán uso de estos buenos hábitos de programación.

El propósito de este capítulo es ayudar a diseñar programas que sean fáciles de leer, comprender, depurar y modificar. Por todo ello, este capítulo es muy importante.

2.1. CONCEPTO DE BLOQUE DE PROGRAMA

Presentación

En el Capítulo 1 se presentó el uso de la estructura de bloques en programación. Se vio que dividir un programa en bloques distintos lo hacía más fácil de leer y modificar. Recuerde la analogía con la estructura de una carta. La estructura hacía más fácil la lectura de la misma.

Esta sección presenta información más detallada sobre el desarrollo de programas con estructura de bloques empleando C.

Un ejemplo de descomposición en bloques

Considérese el Programa 2.1. Este programa simplemente imprime una información en la pantalla. Sin embargo, ilustra una de las características de una estructura de bloques.

Programa 2.1

```
#include <stdio.h>
/*
Programa: Típico programa en C.
Desarrollado por: Un buen programador

Descripción: Este programa ilustra un típico ejemplo de
              programa en C no estructurado.

Variables:   ninguna

Constantes:  ninguna
*/
main()
{
    /* Explicación del programa al usuario. */

    puts("Este es un programa en C que ilustra el típico");
    puts("ejemplo de programa no estructurado");
    puts("Cuando el programa se ejecuta, el usuario no");
    puts("puede decir si el programa esta o no estructurado,");
    puts("solo puede hacerlo el programador.");
    puts("Por tanto, un programa estructurado en C solo es");
    puts("util al programador, al jefe del programador,");
    puts("al profesor del programador, a aquellos que ");
    puts("necesiten modificar el programa y a aquellos");
    puts("que convivan con el programador mientras este ");
    puts("intenta encontrar errores en el programa.");
}
```

Descomponiendo la estructura en bloques

Probablemente, no tenga ningún problema en comprender lo que hace el Programa 2.1. Simplemente imprime un conjunto de cadenas de caracteres por la pantalla. Entonces, ¿por qué estructurarlo?

Deberían considerarse varios aspectos. Primero, el programa parece aburrido. Cada línea del programa comienza en el mismo lugar —una función `puts` después de otra en la misma columna de la izquierda. El uso de párrafos haría más legible y más interesante la estructura del programa. Esto permitiría distinguir una parte del programa del resto. Cada párrafo puede considerarse como un **bloque de programa**, donde cada bloque de programa representa una idea principal dentro del mismo. Este concepto, ya visto en la Sección 1.12, se utilizará de nuevo aquí.

El Programa 2.2 muestra cómo hacer el Programa 2.1 un poco más interesante.

Programa 2.2

```
#include <stdio.h>
/*
Programa: Típico programa en C.
Desarrollado por: Un buen programador
```

Descripción: Este programa ilustra la forma más simple de programa estructurado en bloques.

Variables: ninguna

Constantes: ninguna

*/

main()

{

/* Primer párrafo de la explicación. */

puts("Este es un programa en C que ilustra el tipico");
puts("ejemplo de programa no estructurado");

/* Fin del párrafo de la explicación. */

/*-----*/

/* Segundo párrafo de la explicación. */

puts("Cuando el programa se ejecuta, el usuario no");
puts("puede decir si el programa esta o no estructurado,");
puts("solo puede hacerlo el programador.");

/* Fin del segundo párrafo de la explicación. */

/*-----*/

/* Tercer párrafo de la explicación.. */

puts("Por tanto, un programa estructurado en C solo es");
puts("util al programador, al jefe del programador,");
puts("al profesor del programador, a aquellos que ");
puts("necesiten modificar el programa y a aquellos");
puts("que convivan con el programador mientras este ");
puts("intenta encontrar errores en el programa");

/* Fin del tercer párrafo de la explicación. */

}

Observe el uso de las líneas discontinuas en el Programa 2.2 para separar los diferentes bloques del mismo. Nótese también los comentarios al comienzo y al final de cada bloque. Estos comentarios indican el propósito de cada uno de los bloques del programa. En este programa tan simple, la estructura añadida no es muy importante para comprender lo que hace el programa, pero sirve como un modelo a seguir en lo que resta.

Definición de estructura de bloques

Tener una **estructura de bloques** significa que los programas contruidos de esta manera estarán formados por un conjunto de grupos de instrucciones, en vez de una lista continua de instrucciones, una después de otra. Los programas más grandes del capítulo anterior se presentaron de esta forma.

Cada grupo o bloque de instrucciones comienza con un comentario que explica lo que hace el bloque. Un ejemplo del Programa 2.2 es el siguiente:

```
/* Segundo párrafo de explicación */
```

Cada bloque de programa se encuentra definido por unos espacios de separación denominados **separadores de bloques**. Un ejemplo de separador de bloques[†] es el siguiente:

```
/*-----*/
```

aunque también se pueden utilizar una o más líneas en blanco para separar los diferentes bloques.

El cuerpo de cada bloque de programa se resalta mediante la sangría del mismo. No es crítico el número de espacios dejado a la izquierda, lo que sí es importante es distinguir el cuerpo del bloque de sus comentarios de comienzo y fin. Un ejemplo del Programa 2.2 es el siguiente:

```
/* Segundo párrafo de la explicación. */

puts("Cuando el programa se ejecuta, el usuario no");
puts("puede decir si el programa esta o no estructurado,");
puts("solo puede hacerlo el programador.");

/* Fin del segundo párrafo de la explicación. */
```

Algunas reglas importantes

Lo que es importante en un programa con estructura de bloques es que no existan instrucciones de salto. Si no conoce este concepto (si nunca ha utilizado una sentencia `goto`) se puede considerar agraciado. La gente comprende más fácilmente las cosas si puede seguir las de forma lógica desde un punto a otro. Por tanto, siempre deberían seguirse las siguientes reglas cuando se realiza programación con estructura de bloques.

1. Todos los bloques tienen un único punto de entrada al comienzo de los mismos.
2. Todos los bloques tienen un único punto de salida al final de los mismos.
3. Cuando el computador finaliza la ejecución de un bloque, continúa con otro o finaliza.

[†] El uso de separadores de bloques es opcional. Con el objetivo de ahorrar espacio no siempre se utilizan en los programas de ejemplo.

¿Puede diseñar todos los programas de esta forma? Sí, puede —sin ninguna excepción. No hay ninguna excusa para escribir un programa en C que no tenga estructura de bloques.

Tipos de bloques

Los tres tipos de bloques necesarios, sin importar el lenguaje de programación que se emplee, son los siguientes:

1. Bloque secuencial.
2. Bloque repetitivo.
3. Bloque de selección (o bifurcación).

Un **bloque secuencial** es el tipo más simple de bloque de programación. Consta de un conjunto de sentencias, una seguida de otra. Un **bloque repetitivo** permite la ejecución repetida de una parte del programa. Un **bloque de selección** ofrece la posibilidad de realizar una secuencia diferente de instrucciones. Los conceptos de estos tres tipos diferentes de bloques se ilustran en la Figura 2.1.

Conclusión

Esta sección ha puesto de relieve la importancia de la estructura de bloques en programación y ha ofrecido un ejemplo concreto. Se ha presentado la forma de romper un programa C en bloques. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

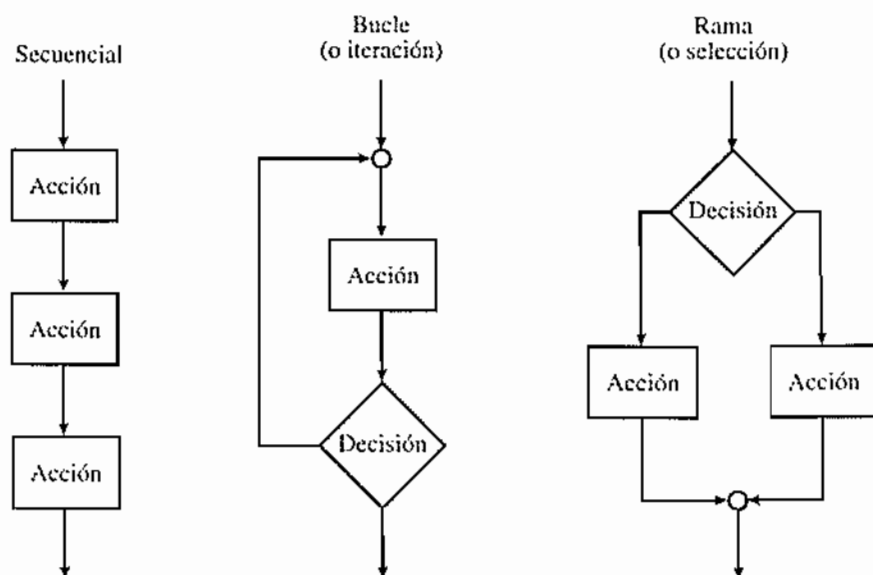


Figura 2.1. Conceptos de las tres clases de bloques de programación.

Repaso de la Sección 2.1

1. Explique razonadamente si un compilador requiere que un programa esté estructurado.
2. Defina el término «estructura de bloques».
3. Indique una forma de comenzar un bloque de programa.
4. Explique cómo se pueden separar los bloques de un programa.
5. Explique cómo se puede resaltar los bloques de un programa.
6. Indique los tres tipos diferentes de bloques en un programa.

2.2. USO DE FUNCIONES

Presentación

En la última sección se presentó el concepto de programación estructurada en bloques. En esta sección se verá como diseñar un programa en C con estructura de bloques. Esta sección servirá como una introducción a este concepto importante de C. A medida que avance en la lectura de este libro tendrá la oportunidad de incrementar su comprensión acerca de este concepto.

¿Qué es una función en C?

Una función en C es un segmento independiente de código fuente diseñado para realizar una tarea específica. Todos los programas escritos en C tienen al menos una función llamada `main()`. Usted ya ha hecho uso de algunas funciones de biblioteca, tales como `puts()`, `printf()` y `scanf()`.

Haciendo sus propias funciones

Usted puede crear sus propias funciones en C. Esto le permitirá escribir una función y expresar en C lo que hace. Luego podrá utilizarla de forma repetida al igual que las funciones de biblioteca de C. Esto significa que usted podría crear una función que resuelva un circuito eléctrico en serie, otra que resuelva un circuito paralelo y otra que resuelva las características eléctricas de un amplificador —por nombrar algunas. Las define una única vez, las da un nombre y luego las llama tantas veces como desee —al igual que llama a las funciones `puts()` y `scanf()` siempre que quiera. Como puede deducirse de esto, para llamar a una función, simplemente se utiliza su nombre. Usted podría crear su propia biblioteca de funciones, grabarlas en su disco y luego invocarlas en sus programas de la misma forma que hace con `puts()`, `printf()` y `scanf()`.

Este concepto se ilustra en la Figura 2.2

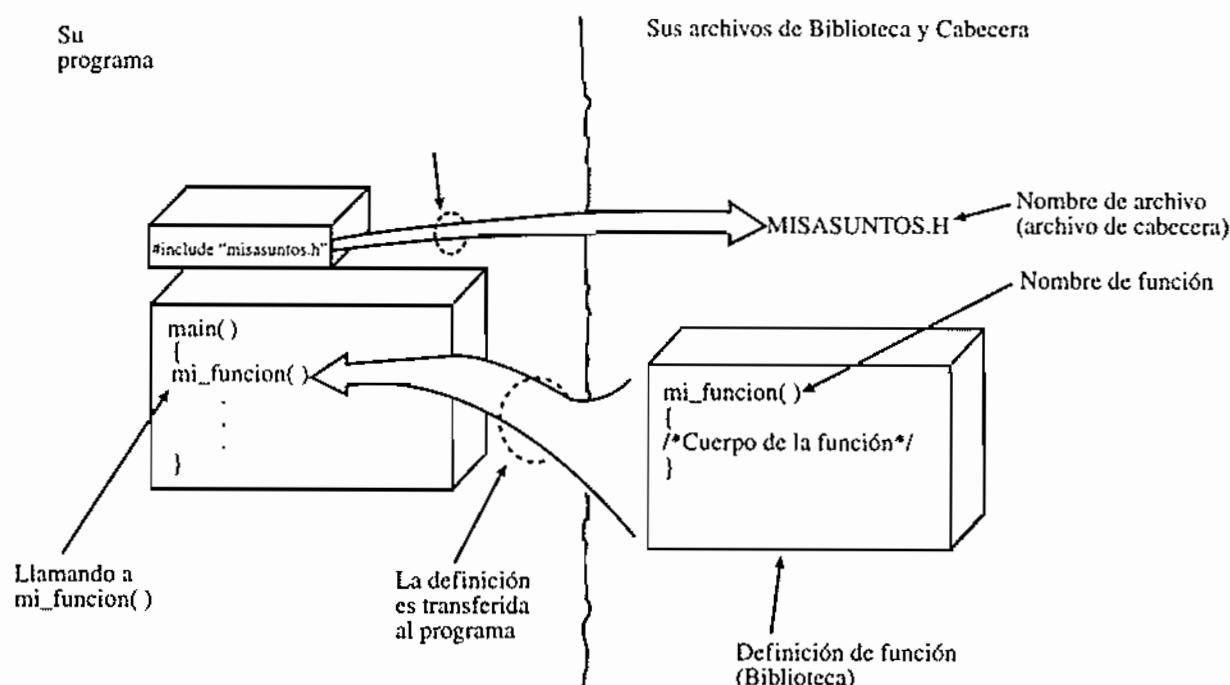


Figura 2.2. Concepto de llamada a función en C.

¿Qué hace una función?

En C, cuando se crea una función diferente a `main()`, primero debe declararse y después definirse. La declaración de una función se realiza mediante lo que se denomina **prototipo de una función**. Un prototipo de función consta del nombre de la función y de información importante relativa a la misma. Los prototipos de las funciones aparecen al comienzo del programa y antes de la función `main()`.

Cuando se define una función, se indica de nuevo el nombre de la función, la información relativa a la misma (al igual que en el prototipo de la función) y el cuerpo de la función, que contiene todo el código fuente utilizado por la función. Todo esto se muestra en el siguiente ejemplo.

Ejemplo. El Programa 2.3 es el programa de la sección anterior pero reescrito con funciones. Su salida es exactamente la misma que la del programa anterior. Usted conoce bien lo que hace el programa, concéntrese en cómo lo hace.

Programa 2.3

```
#include <stdio.h>
/*
Programa: Típico programa en C.
Desarrollado por: Un buen programador
```

Descripción: Este programa ilustra la forma más simple de programa estructurado en bloques.

Variables: ninguna

Constantes: ninguna

Prototipos de las funciones:

```

*/

void primer_parrafo(void);
/* Esta función realiza el primer párrafo para este */
/* programa */
/*----- */

void segundo_parrafo(void);
/* Esta función realiza el segundo párrafo para este */
/* programa */
/*----- */

void tercer_parrafo(void);
/* Esta función realiza el tercer párrafo para este */
/* programa */
/*----- */

main()
{
    primer_parrafo();    /* Primer párrafo de la explicación. */
    segundo_parrafo();   /* Segundo párrafo de la explicación. */
    tercer_parrafo();    /* Tercer párrafo de la explicación. */
    exit(0);
}

void primer_parrafo()
{
    /* Primer párrafo de la explicación. */

    puts("Este es un programa en C que ilustra el tipico");
    puts("ejemplo de programa no estructurado");
}

void segundo_parrafo()
{
    /* Segundo párrafo de la explicación. */

    puts("Cuando el programa se ejecuta, el usuario no");
    puts("puede decir si el programa esta o no estructurado,");
    puts("solo puede hacerlo el programador.");

    /* Fin del segundo párrafo de explicación. */
}

```

```

void tercer_parrafo()
{
    /* Tercer párrafo de explicación.. */

    puts("Por tanto, un programa en C estructurado solo es");
    puts("util al programador, al jefe del programador,");
    puts("necesiten modificar el programa y a aquellos");
    puts("que convivan con el programador mientras este");
    puts("intenta encontrar errores en el programa");

    /* Fin del tercer párrafo de explicación. */
}

```

Análisis del programa

La clave para analizar el Programa 2.3 es observar lo que contiene la función `main()`:

```

main()
{
    primer_parrafo();      /* Primer párrafo de la explicación. */
    segundo_parrafo();     /* Segundo párrafo de la explicación. */
    tercer_parrafo();      /* Tercer párrafo de la explicación. */
    exit(0);
}

```

En este programa la función `main()` consta de una serie de llamadas a otras funciones. La primera de todas ellas es `primer_parrafo()`. La definición de esta función es la siguiente:

```

void primer_parrafo()
{
    /* Primer párrafo de la explicación. */

    puts("Este es un programa en C que ilustra el típico");
    puts("ejemplo de programa no estructurado");

    /* Fin del primer párrafo de la explicación. */
}

```

Esta definición consta de un código de programa encerrado entre llaves (`{` y `}`). Esto es exactamente lo mismo que se hace en la función reservada de C, `main()`. `main()` lleva a cabo el proceso de llamada a cada una de las funciones en el orden necesario. Este proceso se ilustra en la Figura 2.3


```
void tercer_parrafo(void);
/* Esta función realiza el tercer párrafo para este */
/* programa */
/*-----*/
```

Observe que las tres funciones definidas en el programa (`primer_parrafo`, `segundo_parrafo` y `tercer_parrafo`) no se encuentran comentadas (no están encerradas entre `/*` y `*/`). Esto significa que representan algún tipo de información para el compilador de C. Esta información indica al compilador que existe una función con el mismo nombre definida más tarde en el programa. Esto es el prototipo de la función. Cada función tiene un tipo (al igual que las variables y las constantes) y en este programa su tipo es `void`, lo que significa que la función no devuelve ningún valor. Esto se debe a que la única tarea que realiza la función es imprimir algunas palabras en la pantalla; no realiza ningún cálculo. La segunda cosa que muestra, es que no hay nada en su lista de parámetros formales (encerrados entre `( y )`). Este aspecto es muy importante ya que indica al compilador el tipo y la cantidad de memoria a reservar para estas funciones. La **lista de parámetros formales** se declara en este caso como `void`, debido a que se encuentra vacía.

El concepto de prototipo de una función se ilustra en la Figura 2.4.

Nótese que cada prototipo finaliza con un punto y coma, de forma semejante a cuando se realiza la llamada a la misma en la función `main()`. Sin embargo, observe —y esto es importante— que cuando se define una función, ésta no finaliza con un punto y coma. Esto es lo mismo que ocurre con `main()` cuando se define no se finaliza con un punto y coma.

Observe la estructura de la definición de la siguiente función:

```
void primer_parrafo()
{
    /* Primer párrafo de la explicación. */

    puts("Este es un programa en C que ilustra el tipico");
    puts("ejemplo de programa no estructurado");

    /* Fin del primer párrafo de la explicación. */
}
```

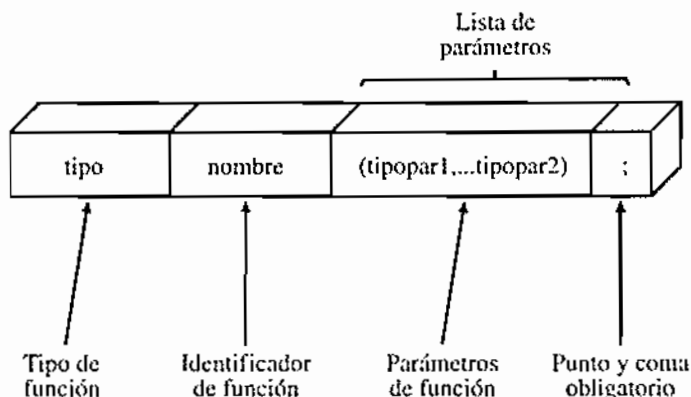


Figura 2.4. Concepto de prototipo de función.

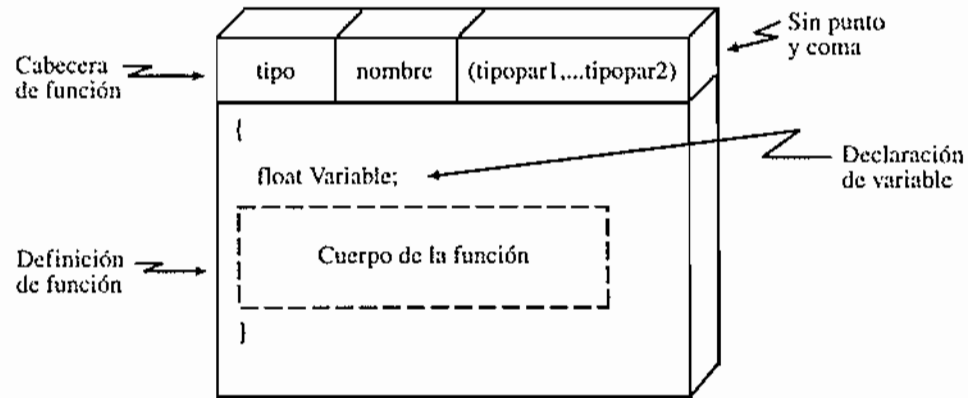


Figura 2.5. Estructura de una definición de función .

Esta definición indica el tipo de la función (`void` —puesto que no devuelve ningún valor—) y su nombre. Este nombre es seguido por los paréntesis `()` obligatorios, los cuales no contienen entre ellos nada como se indicó en la declaración del prototipo (éste mostraba `void()`), y no hay punto y coma (como se dijo anteriormente). El cuerpo de la función se encuentra definido por la apertura de llave `{` y el cierre de llave `}`.

La estructura de una definición de función se ilustra en la Figura 2.5.

Hay un último punto importante. Considérese de nuevo la función `main()` :

```
main()
{
    primer_parrafo();      /* Primer párrafo de la explicación. */
    segundo_parrafo();     /* Segundo párrafo de la explicación. */
    tercer_parrafo();      /* Tercer párrafo de la explicación. */
    exit(0);
}
```

La última función a la que se llama es `exit(0)`; `exit` es una función C que provoca la finalización normal de un programa en C.

Conclusión

Esta sección ha presentado mucha información nueva. La importancia de esta sección no radica tanto en los detalles exactos (aunque estos son muy importantes cuando se programa) sino más bien en los conceptos —el concepto de cómo dividir un programa en partes discretas denominadas funciones, y como una función, la función `main()`, puede llamar a esas funciones en un orden que determina la ejecución del programa. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 2.2

1. ¿Qué es una función en C?
2. Explique el significado del tipo `void`. Dé un ejemplo.

3. Indique el significado de prototipo de una función.
4. ¿Cuál es el propósito del prototipo de una función?
5. Explique el significado de una llamada a función.
6. ¿Por qué `exit()` es la última función llamada en `main()`?

2.3. DENTRO DE UNA FUNCION C

Presentación

En esta sección descubrirá cómo se pueden utilizar las funciones para devolver un valor. Esto significa que se pueden desarrollar funciones para realizar cálculos complejos y devolver los resultados de los mismos a la función que realizó la llamada.

Idea básica

El Programa 2.4 lee un valor del usuario del programa y lo eleva al cuadrado.

Programa 2.4 `#include <stdio.h>`

```
main()
{
    flota numero;          /* Número a elevar al cuadrado. */
    float respuesta;       /* El cuadrado del número.      */
    printf("Deme un numero y lo elevare al cuadrado => ");
    scanf("%f", &numero);
    respuesta = numero * numero;
    printf("El cuadrado de %f es %f", numero, respuesta);
    exit(0);
}
```

Cuando se ejecuta el Programa 2.4 y el usuario introduce el valor 3, el resultado del programa es el siguiente:

```
Deme un numero y lo elevare al cuadrado => 3
El cuadrado de 3.000000 es 9.000000
```

El Programa 2.5 ilustra cómo hacer el Programa 2.4 empleando una función C.

Programa 2.5 `#include <stdio.h>`

```
/* Prototipo de función. */

float cuadrado(float numero);
/* Esta es la función que eleva un número al cuadrado. */
```

```

main()
{
    float valor;          /* Número a elevar al cuadrado.      */
    float respuesta;      /* El cuadrado del número. */

    printf("Dame un número y lo elevare al cuadrado => ");
    scanf("%f", &valor);
    respuesta = cuadrado(valor);      /* Llamada a la función. */
    printf("El cuadrado de %f es %f", valor, respuesta);
    exit(0);
}

float cuadrado(float numero)
{
    float respuesta;      /* El cuadrado del número. */
    respuesta = numero * numero;
    return(respuesta);    /* Devuelve el valor del cuadrado. */
}

```

Análisis del programa.

El Programa 2.5 consta de una función separada denominada `cuadrado()`. Esta nueva función calcula el cuadrado de un número ¿de dónde obtiene ese número? Este número se obtiene de la función `main()`, ¿cómo lo obtiene?, desde la función `main()`. ¿Cómo se pasa? Se pasa a través de su argumento (un número de tipo `float`). La Figura 2.6 ilustra cómo se realiza el **paso de valores**.

En la Figura 2.6 se observa que la lista de parámetros formales contiene las declaraciones de los parámetros de la función. En este caso, la función sólo tiene uno, llamado `parametro_1`. Este parámetro formal aparece tanto en el prototipo de la función como en la cabecera de la definición de la función. Sin embargo, cuando se llama a la función (en este caso desde `main()`) su parámetro real no necesita tener el mismo nombre que su parámetro formal, sólo debe tener el mismo tipo. Por ejemplo, en el Programa 2.5, el parámetro real es `valor` y el parámetro formal `numero`. De este modo los **parámetros formales** definen los tipos y el número de los parámetros de la función, mientras que los **parámetros reales** se utilizan cuando se llama a la función.

Como se puede observar en la figura, el parámetro formal de la función `cuadrado` no es `void`, esto es, no es `cuadrado(void)` sino `cuadrado(float numero)`. Esta declaración se realizó por primera vez en la parte de prototipos del Programa 2.5:

```

/* Prototipo de función. */

float cuadrado(float numero);
/* Esta es la función que eleva un número al cuadrado. */

```

Observe que en este caso la función devuelve un valor, es decir, no es de tipo `void`. Por lo tanto, es necesario indicar al compilador su tipo, en este caso el tipo de la

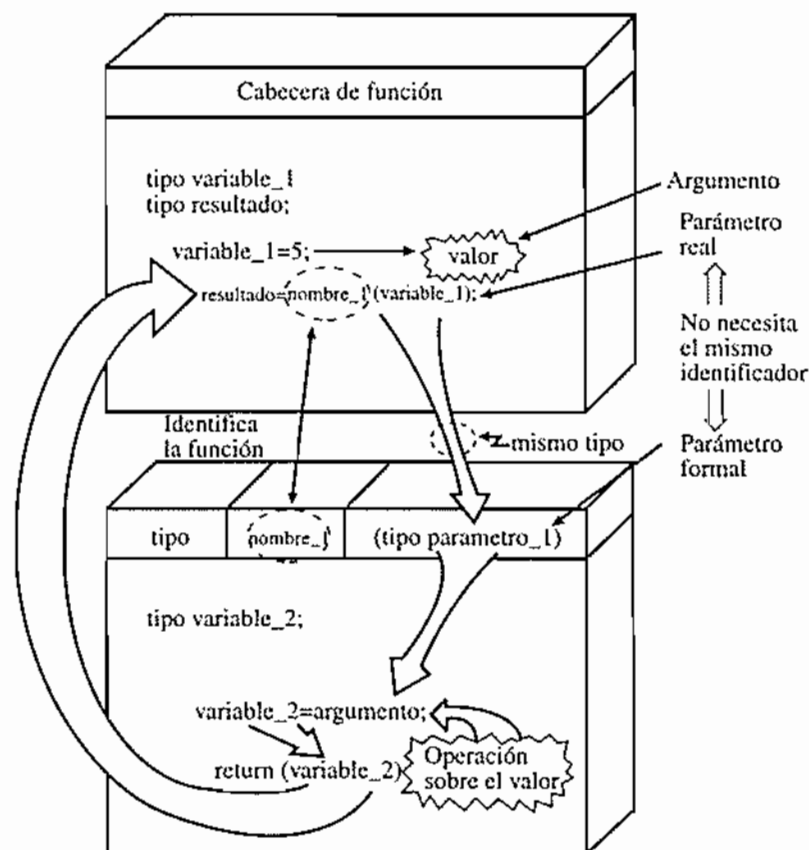


Figura 2.6. Pasando un valor de una función a otra.

función es `float`. Esto significa que el valor devuelto por la función después de elevar el número al cuadrado será de tipo `float`. En esta función el número que se eleva al cuadrado también debe ser del mismo tipo. Este hecho se indica al compilador en la declaración del prototipo:

```
(float numero);
```

La declaración del prototipo para esta función informa al compilador de la existencia de la definición de una función en el programa identificada como `cuadrado` que devolverá un valor de tipo `float` y que realizará su operación (en este caso elevar al cuadrado) sobre otro número identificado como `numero`, que es también de tipo `float`. Toda esta información es necesaria y se indica en la declaración del prototipo de la función:

```
float cuadrado(float numero);
```

Observe que la declaración finaliza con un punto y coma obligatorio. A continuación se presenta la definición de la función:

```
float cuadrado(float numero)
{
    float respuesta;          /* El cuadrado del número. */
    respuesta = numero * numero;
    return respuesta;
}
```

```

    respuesta = numero * numero;
    return(respuesta);    /* Devuelve el valor del cuadrado. */
}

```

Esta definición de función comienza exactamente igual que su prototipo; la única diferencia importante es que *no finaliza con un punto y coma*. Este hecho indica al compilador que lo que sigue a continuación en el programa es la definición de una función. También define su tipo y el tipo de su parámetro, de la misma forma que en el prototipo. Recuérdese, que el prototipo de la función indica al compilador el tipo de memoria que tiene que reservar; la definición de la función describe exactamente la acción que lleva a cabo la misma.

Hay un nuevo término de C que se ha utilizado en esta definición de función. Este es:

```
return();
```

Este término indica al computador el valor a ser devuelto a la función que realizó la llamada. En este caso se devuelve el valor de la variable `respuesta`. Para ello, el identificador de la variable `respuesta` se sitúa como argumento dentro de `return`:

```
return(respuesta);
```

que finaliza con un punto y coma obligatorio. Así, esta función tomará un valor de la función que realizó la llamada. Este valor es de tipo `float` y se almacena en el identificador de variable `numero`. A continuación se realiza el siguiente cálculo:

```
respuesta = numero * numero;
```

cuyo resultado se almacena en el identificador denominado `respuesta`. El valor almacenado en `respuesta` se devuelve a continuación a la función que realizó la llamada mediante la siguiente sentencia:

```
return(respuesta);
```

Véase a continuación la función `main()`:

```

main()
{
    float valor;          /* Número a elevar al cuadrado. */
    float respuesta;      /* El cuadrado del número.      */

    printf("Dame un número y lo elevare al cuadrado => ");
    scanf("%f", &valor);
    respuesta = cuadrado(valor); /* Llamada a la función. */
    printf("El cuadrado de %f es %f", valor, respuesta);
    exit(0);
}

```

La función `main()` utiliza dos variables. Una es el valor del número a elevar al cuadrado (`valor`) y la otra se emplea para almacenar el resultado del cálculo (`res-`

puesta). El programa simplemente pide al usuario un número a elevar al cuadrado y a continuación lee ese número del teclado. Estas acciones se realizan mediante las funciones `printf` y `scanf`. Pero véase que ahora el cálculo se lleva a cabo de forma diferente, puesto que se realiza una llamada a la función que define el cuadrado. El valor de la variable `valor` se utiliza como parámetro de la función y esta devuelve el resultado en la variable `respuesta`. La función `exit(0);` finaliza la ejecución del programa.

La Tabla 2.1 ilustra algunas de las definiciones importantes acerca de las funciones.

Conclusión

Esta sección ha sido muy importante. De nuevo los detalles han sido menos importantes que los conceptos. Se ha presentado el concepto de paso de valores de una función a otra. Se ha visto que esto se puede realizar utilizando parámetros que representan un lugar donde almacenar un valor que se pasa de una función a otra. La segunda idea importante es el empleo de la sentencia de C `return()` que se utiliza para devolver un valor a la función que realizó la llamada.

En la siguiente sección se verá una aplicación más útil del paso de valores entre funciones. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 2.3

1. Indique qué entiende por parámetro de una función.
2. ¿Cuál es la diferencia que existe entre un parámetro formal y un parámetro real? ¿Deben ser los mismos? ¿Puede ser diferentes?
3. Explique el significado del paso de valores entre funciones.
4. ¿Qué diferencia existe entre el prototipo de una función y la cabecera de la definición de una función?
5. ¿Cómo se devuelve un valor a una función que llama a otra?

Tabla 2.1. Definiciones importantes acerca de funciones

Término	Ejemplo	Comentarios
Parámetros formales (usados cuando se define la función).	<code>cuadrado_it (float numero)</code>	<code>numero</code> es un parámetro formal. Indica que un valor de tipo real será pasado a la función llamada <code>cuadrado_it (valor)</code> .
Parámetros reales (usados cuando se llama a la función definida).	<code>cuadrado_it (valor)</code>	<code>valor</code> es un parámetro real. El identificador usado como parámetro real puede ser diferente del indicado como parámetro formal, pero debe ser del mismo tipo.
Argumentos (valores reales que se pasan a la función llamada).	<code>valor = 5;</code> <code>cuadrado_it (valor);</code>	El número 5 es el argumento porque es el valor real que se pasa a la función <code>cuadrado_it (valor)</code> .

2.4. USO DE FUNCIONES

Presentación

En la última sección se presentó la potencia del paso de valores entre funciones. En esta sección se verá cómo pasar más argumentos, así como más aspectos útiles relacionados con las funciones. Encontrará que la columna vertebral del lenguaje C radica en las funciones. Se dedicará mucho tiempo a aprender sobre las mismas.

Más de un argumento

El Programa 2.6 ilustra el paso de múltiples argumentos entre funciones. Este programa calcula la reactancia capacitiva de un condensador dado el valor del condensador y la frecuencia. La fórmula matemática que permite calcular la reactancia capacitiva de un condensador es la siguiente:

$$X_C = 1/(2\pi fC)$$

donde:

X_C = La reactancia del condensador en ohmios.
 f = La frecuencia en hertzios.
 C = El valor del condensador en faradios.

Programa 2.6 `#include <stdio.h>`

```

/* Prototipo de la función. */

float reactancia_capacitiva(float capacidad, float frecuencia);

/* Esta función calcula la reactancia capacitiva para un valor
   dado de condensador y una frecuencia. */

main()
{
    float faradios;    /* Valor del condensador. */
    float hertzios;    /* Valor de la frecuencia. */
    float reactancia;  /* Valor de la reactancia.*/

    printf("Introduzca el valor del condensador en faradios => ");
    scanf("%f",&faradios);
    printf("Introduzca el valor de la frecuencia en hertzios => ");
    scanf("%f",&hertzios);

    reactancia = reactancia_capacitiva(faradios, hertzios);

```

```

        printf("La reactancia de un condensador de %e\n", faradios);
        printf("a la frecuencia de %e hertzios es %e ohmios\n", hertzios,
            reactancia);
        exit(0);
    }

float reactancia_capacitiva(float capacidad, float frecuencia)
{
    float reactancia;          /* La reactancia capacitiva. */
    float pi = 3.14159;

    reactancia = 1/(2 * pi * frecuencia * capacidad);
    return(reactancia);
}

```

Análisis del programa

El Programa 2.6 hace uso de dos funciones: `main()` y `reactancia_capacitiva()`. La característica principal de este programa es que se pasan dos valores entre las funciones `main()` y `reactancia_capacitiva()` (el valor del condensador y el valor de la frecuencia). Esto se consigue utilizando dos parámetros formales para la función `reactancia_capacitiva()`:

```
float reactancia_capacitiva(float capacidad, float frecuencia);
```

Como se puede observar en este prototipo, los parámetros formales se identifican como `capacidad` y `frecuencia`, ambos declarados de tipo `float`. La función devuelve un valor, también de tipo `float`. Vease la definición de la función:

```

float reactancia_capacitiva(float capacidad, float frecuencia)
{
    float reactancia;          /* La reactancia capacitiva. */
    float pi = 3.14159;

    reactancia = 1/(2 * pi * frecuencia * capacidad);
    return(reactancia);
}

```

De nuevo, la cabecera de la función es exactamente la misma que la del prototipo —con la única excepción de que no finaliza con un punto y coma. El cuerpo de la función comienza con la llave obligatoria `{` y declara dos identificadores. El primero, `reactancia`, representa el valor de la reactancia capacitiva calculada y el otro el valor de `pi`. A continuación se realiza el cálculo de la reactancia capacitiva y este valor es devuelto a la función que realiza la llamada.

Véase ahora la función `main()`:


```

main()
{
    float faradios;      /* Valor del condensador. */
    float hertzios;     /* Valor de la frecuencia. */
    float reactancia;   /* Valor de la reactancia. */

    printf("Introduzca el valor del condensador en faradios => ");
    scanf("%f",&faradios);
    printf("Introduzca el valor de la frecuencia en hertzios => ");
    scanf("%f",&hertzios);

    reactancia = reactancia_capacitiva(faradios, hertzios);

    printf("La reactancia de un condensador de %e faradios\n",faradios);
    printf("a la frecuencia de %e hertzios es %e ohmios\n",hertzios,
           reactancia);
    exit(0);
}

```

main() contiene la declaración de tipos de tres variables. Las dos primeras, faradios y hertzios, son introducidas por el usuario por medio de dos llamadas diferentes a la función scanf(). Cada una de estas variables se encuentra precedida por el operador obligatorio &. La tercera, reactancia, se utiliza para mostrar el valor resultante. Una vez introducidos los valores del condensador y la frecuencia, se llama a la función definida anteriormente pasándole los dos argumentos:

```
reactancia = reactancia_capacitiva(faradios, hertzios);
```

El valor devuelto por la función se almacena en la variable reactancia. Este resultado se visualiza a continuación en la pantalla del computador para que el usuario del programa pueda observar dicho resultado.

La ejecución del Programa 2.6 produce los resultados que se muestran a continuación (considérese que el usuario del programa introduce los siguientes valores: un condensador de 1 μ f y una frecuencia de 5 KHz):

```

Introduzca el valor del condensador en faradios => 1e-6
Introduzca el valor de la frecuencia en hertzios => 5e3
La reactancia capacitiva de un condensador de 1e-6 faradios
a la frecuencia de 5e3 hertzios es 3.183101e+001 ohmios.

```

Llamando a más de una función

Recuerde que en C una función puede llamar a más de una función. En el Programa 2.7 se realizan llamadas a más de una función. Este programa calcula la reactancia

capacitiva de un condensador y la reactancia inductiva de una inductancia, donde la reactancia inductiva viene dada por:

$$X_L = 2\pi fL$$

donde

X_L = La reactancia inductiva en ohmios.
 f = La frecuencia en hertzios.
 L = La inductancia en henrios.

Programa 2.7 #include <stdio.h>

```
/* Prototipos de funciones. */

float reactancia_capacitiva(float capacidad, float frecuencia);

/* Esta función calcula la reactancia capacitiva para una valor
   dado de un condensador y una frecuencia */

float reactancia_inductiva(float inductancia, float frecuencia);

/* Esta función calcula la reactancia inductiva para un valor
   dado de una inductancia y una frecuencia */

main()
{
    float faradios;          /* Valor del condensador.  */
    float henrios;          /* Valor de la inductancia. */
    float hertzios;         /* Valor de la frecuencia. */
    float reactancia;       /* Valor de la reactancia. */

    printf("Introduzca el valor del condensador en faradios => ");
    scanf("%f",&faradios);
    printf("Introduzca el valor de la inductancia en henrios => ");
    scanf("%f",&henrios);
    printf("Introduzca el valor de la frecuencia en hertzios => ");
    scanf("%f",&hertzios);

    reactancia = reactancia_capacitiva(faradios, hertzios);
    printf("La reactancia de un condensador de %e faradios\n",
           faradios);
    printf("a una frecuencia de %e hertzios es %e ohmios\n",hertzios,
           reactancia);

    reactancia = reactancia_inductiva(henrios, hertzios);
    printf("La reactancia de una inductancia de %e henrios\n",henrios);
```

```

        printf("a una frecuencia de %e hertzios es %e ohmios\n", hertzios,
               reactancia);
        exit(0);
    }

float reactancia_capacitiva(float capacidad, float frecuencia)
{
    float reactancia;          /* reactancia capacitiva. */
    float pi = 3.14159;

    reactancia = 1/(2 * pi * frecuencia * capacidad);

    return(reactancia);
}

float reactancia_inductiva(float inductancia, float frecuencia)
{
    float reactancia;          /* reactancia inductiva. */
    float pi = 3.14159;

    reactancia = 2 * pi * frecuencia * inductancia;
    return(reactancia);
}

```

Análisis del programa

La principal diferencia en el Programa 2.7 es que hay dos funciones aparte de `main()`. Éstas son:

```
float reactancia_capacitiva(float capacidad, float frecuencia);
```

y

```
float reactancia_inductiva(float inductancia, float frecuencia);
```

Ambas funciones tienen su propia definición. Al igual que antes, cada definición de función comienza con la misma cabecera que su prototipo, con la excepción requerida del punto y coma:

```

float reactancia_capacitiva(float capacidad, float frecuencia)
{
    float reactancia;          /* reactancia capacitiva. */
    float pi = 3.14159;

    reactancia = 1/(2 * pi * frecuencia * capacidad);
    return(reactancia);
}

```

```

float reactancia_inductiva(float inductancia, float frecuencia)
{
    float reactancia;          /* reactancia inductiva. */
    float pi = 3.14159;

    reactancia = 2 * pi * frecuencia * inductancia;
    return(reactancia);
}

```

No hay ninguna diferencia en el orden en que aparezcan estas funciones en el programa. La llamada a cada una de estas funciones se realiza en la función `main()`:

```

reactancia = reactancia_capacitiva(faradios, hertzios);

y

reactancia = reactancia_inductiva(henrios, hertzios);

```

Observe que se utiliza la misma variable, `reactancia`, en ambos casos. Esto se debe a que el valor almacenado en ella se muestra por pantalla justo después de su asignación y puede por tanto ser reutilizada.

```

reactancia = reactancia_capacitiva(faradios, hertzios);
printf("La reactancia de un condensador de %e faradios\n", faradios);
printf("a una frecuencia de %e hertzios es %e ohmios\n", hertzios,
       reactancia);

reactancia = reactancia_inductiva(henrios, hertzios);
printf("La reactancia de una inductancia de %e henrios\n", henrios);
       reactancia);

```

Llamadas a funciones desde dentro de una función

Una función puede a su vez llamar a otras funciones. Este concepto se ilustra en la Figura 2.7.

El Programa 2.8 muestra este proceso. Este programa calcula la impedancia de un circuito LC en serie. La relación matemática es la siguiente:

$$Z = X_L - X_C$$

donde

Z = La impedancia del circuito en ohmios.
 X_C = La reactancia capacitiva en ohmios.
 X_L = La reactancia inductiva en ohmios.

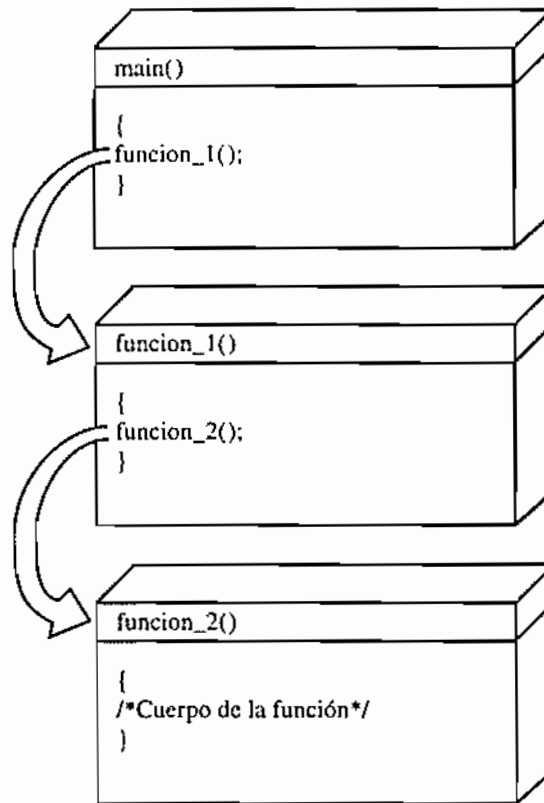


Figura 2.7. Una función llamada 'Llamando a otra función'.

Ya se conoce la forma de calcular cada una de las reactancias de forma individual. En el Programa 2.8 se combinan estas dos reactancias en un único valor que representa la impedancia total del circuito en serie. El punto importante a observar en este programa es cómo una función que ha sido llamada desde `main()` llama a su vez a otras dos funciones.

Programa 2.8 `#include <stdio.h>`

`/* Prototipos de funciones. */`

`float reactancia_capacitiva(float capacidad, float frecuencia);`

`/* Esta función calcula la reactancia capacitiva para un valor
dado de un condensador y una frecuencia */`

`float reactancia_inductiva(float inductancia, float frecuencia);`

`/* Esta función calcula la reactancia inductiva para un valor
dado de una inductancia y una frecuencia */`

`float impedancia_en_serie(float capacidad, float inductancia,`

```

        float frecuencia);
/* Esta función calcula la impedancia en una inductancia y un
   condensador en serie */

main()
{
    float faradios;      /* Valor del condensador. */
    float henrios;       /* Valor de la inductancia. */
    float hertzios;      /* Valor de la frecuencia. */
    float impedancia;    /* Valor de la impedancia. */

    printf("Introduzca el valor del condensador en faradios => ");
    scanf("%f",&faradios);
    printf("Introduzca el valor de la inductancia en henrios => ");
    scanf("%f",&henrios);
    printf("Introduzca el valor de la frecuencia en hertzios => ");
    scanf("%f",&hertzios);

    impedancia = impedancia_en_serie(faradios, henrios, hertzios);

    printf("La impedancia del circuito es %e ohmios\n", impedancia);
    exit(0);
}

float reactancia_capacitiva(float capacidad, float frecuencia)
{
    float reactancia;    /* La reactancia capacitiva. */
    float pi = 3.14159;

    reactancia = 1/(2 * pi * frecuencia * capacidad);

    return(reactancia);
}

float reactancia_inductiva(float inductancia, float frecuencia)
{
    float reactancia;    /* La reactancia inductiva. */
    float pi = 3.141559;

    reactancia = 2 * pi * frecuencia * inductancia;

    return(reactancia);
}

float impedancia_en_serie(float capacidad, float inductancia,
                          float frecuencia)
{
    float react_cap;     /* Reactancia capacitiva resultante. */
    float react_ind;     /* Reactancia inductiva resultante. */
    float impedancia;    /* Impedancia resultante. */

```

```

    react_cap = reactancia_capacitiva(capacidad, frecuencia);
    react_ind = reactancia_inductiva(inductancia, frecuencia);
    printf("Reactancia capacitiva => %e ohmios\n", react_cap);
    printf("Reactancia inductiva => %e ohmios\n", react_ind);

    impedancia = react_ind - react_cap;

    return(impedancia);
}

```

Análisis del programa

El Programa 2.8 muestra como una función puede a su vez llamar a otras funciones. Nótese que no existe ninguna diferencia en el orden en el que se definan las funciones dentro del programa. La función `impedancia_en_serie` es llamada desde `main()`. La definición de esta función se encuentra al final del programa. Esta función llama a su vez a otras dos funciones:

```

react_cap = reactancia_capacitiva(capacidad, frecuencia);
react_ind = reactancia_inductiva(inductancia, frecuencia);

```

Observe que cada una de estas dos funciones devuelve el valor calculado a la función que realizó la llamada (`impedancia_en_serie`). Estos valores se utilizan en el cálculo de la impedancia. El resultado se obtiene restando los valores de la reactancia inductiva y la reactancia capacitiva. Este resultado será negativo cuando el valor de la reactancia capacitiva sea mayor que el de la reactancia inductiva.

La función `impedancia_en_serie()` devuelve el valor calculado a la función que la llamó, `main()`. En ésta, se visualiza el valor final al usuario del programa. Asumiendo que el usuario del programa introduce los siguientes valores: un condensador de 1 μF , una inductancia de 1 mH y una frecuencia de 5 KHz, la ejecución del programa anterior produciría la siguiente salida:

```

Introduzca el valor del condensador en faradios => 1e-6
Introduzca el valor de la inductancia en henrios => 1e-3
Introduzca el valor de la frecuencia en hertzios => 5e3
Reactancia capacitiva => 3.183101e+001 ohmios.
Reactancia inductiva => 3.141590e+001 ohmios.
La impedancia del circuito es -4.151115e-001 ohmios.

```

Cómo se puede llamar a las funciones

La Figura 2.8 ilustra las diferentes formas en las que una función puede llamar a otras funciones. Nótese que una función puede llamarse a sí misma. Esto se conoce como **recursividad** o **recursión**. La única restricción es que no se puede definir una función dentro del cuerpo de otra función.

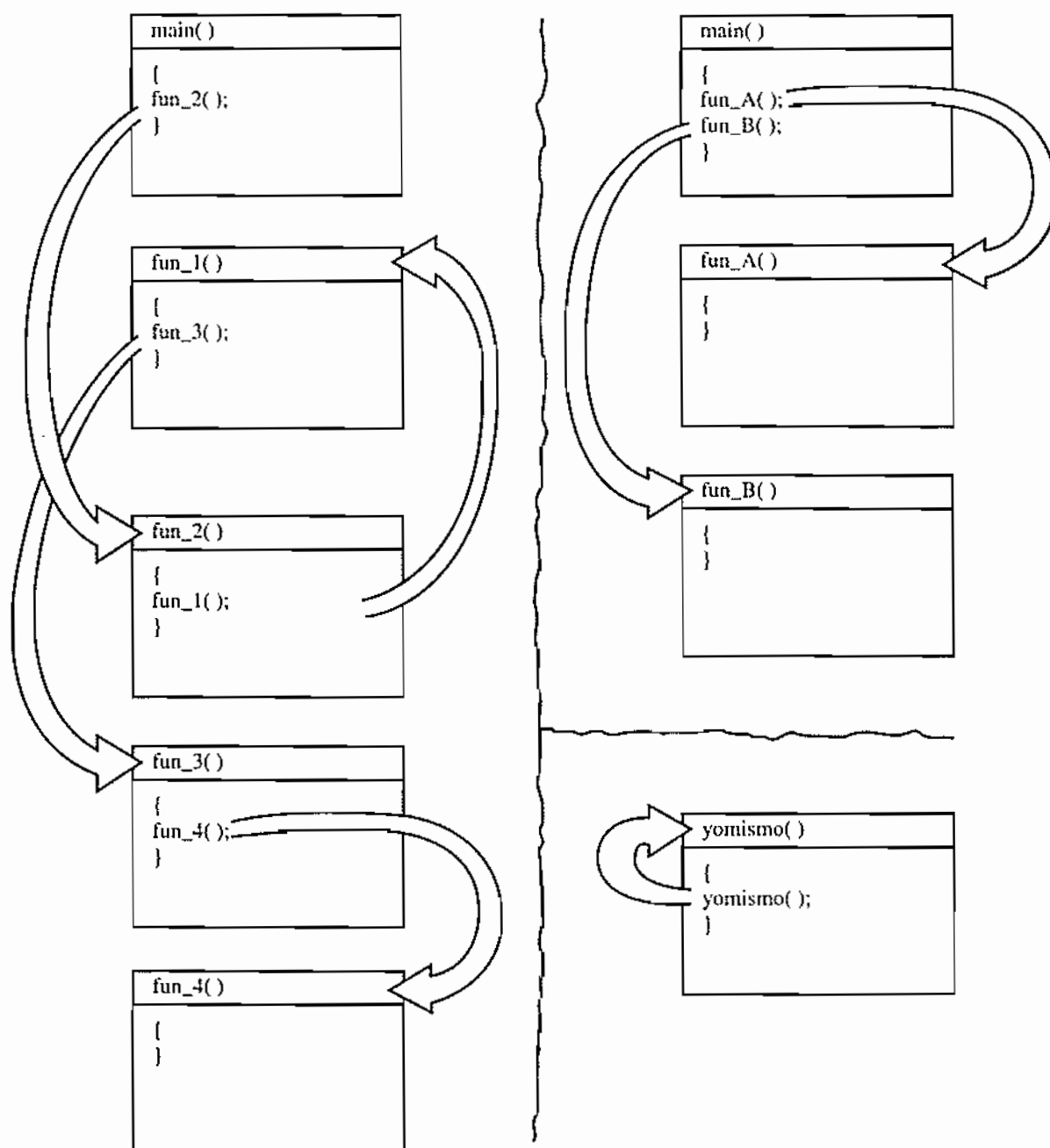


Figura 2.8. Cómo pueden las funciones llamar a otras funciones.

Conclusión

En esta sección se ha aprendido más acerca de la estructura de un programa C. Se ha visto que se puede pasar más de un valor en la llamada a una función y se ha visto además, que una función puede a su vez llamar a más de una función. El orden en el que se definen las funciones en un programa no es importante. Por último se ha visto que una función puede a su vez llamar a otras funciones e incluso a sí misma.

En la siguiente sección aprenderá a definir elementos en sus programas C. Como se verá, puede almacenar esta información en su propio disco y utilizarla más tarde en todos sus programas. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 2.4

1. Explique razonadamente si una función puede pasar más de un valor en la llamada a otra función.
2. Explique razonadamente si es posible que una función llame a su vez a más de una función.
3. ¿Existe alguna diferencia en el orden en el que se definen las funciones en el cuerpo de un programa C?
4. Explique el significado de una función llamada que llama a su vez a otra función.
5. ¿Qué es la recursividad?

2.5. USO DE LA DIRECTIVA #define

Presentación

En la sección anterior se vio más acerca de la potencia de las funciones. En esta sección se muestra como definir sus propias constantes en sus programas. Esto hace más transportables sus programas, más fáciles de modificar y más fáciles de comprender. En esta sección se verá como utilizar esta nueva y potente característica de C.

Idea básica

Observe el Programa 2.9. Este programa calcula el área de un círculo.

Programa 2.9

```
#include <stdio.h>
#define cuadrado(x) x*x
#define PI 3.141592
#define area_circulo(r) PI * cuadrado(r)

main()
{
    float area;      /* Area del círculo. */
    float radio;     /* Radio del círculo. */

    printf("Dame el radio => ");
    scanf("%f",&radio);

    area = area_circulo(radio);

    printf("El area de un circulo de radio %f\n",radio);
    printf("es %f unidades cuadradas\n.", area);
}
```

El Programa 2.9 utiliza tres directivas `#define` del preprocesador. La primera

```
#define cuadrado(x) x*x
```

define una macro. Una macro, en este sentido, es simplemente una cadena de elementos léxicos que serán sustituidos por otra cadena de elementos léxicos. Cuando la sentencia

```
cuadrado(x)
```

aparezca en el programa, el preprocesador la sustituirá por

```
x*x
```

Como puede observar, los blancos situados a cada lado de la directiva del preprocesador sirven para separar los elementos léxicos a ser sustituidos. Esto se ilustra en la Figura 2.9.

Una macro puede utilizarse en cualquier función de un programa.

La segunda directiva `#define` se utiliza para definir la constante `PI` (en C conviene escribir las constantes en mayúsculas). Cualquier aparición de `PI` en el programa será sustituida por el valor 3.141592.

La tercera directiva

```
#define area_circulo(r) PI*cuadrado(r)
```

define una operación matemática que incluye la macro `cuadrado(r)`. La macro `area_circulo` calcula el área de un círculo.

La ejecución del Programa 2.9 produce el siguiente resultado:

```
Dame el radio => 3
El area de un circulo de radio 3.000000
es 28.274334 unidades cuadradas.
```

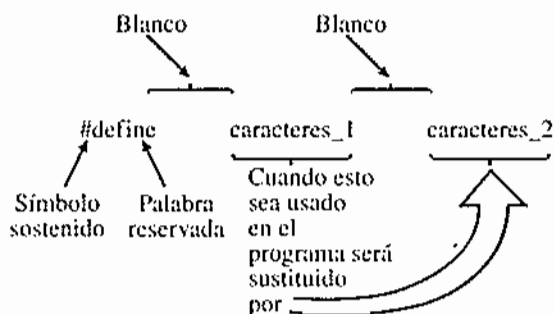


Figura 2.9. Construcción de la sentencia `#define`.

Revisando un programa anterior

Recuerde el Programa 2.8 de la sección anterior, que calculaba la impedancia de un circuito LC en serie. Este programa puede reescribirse de nuevo utilizando directivas del preprocesador. El Programa 2.10 muestra cómo se puede usar una macro con parámetros para definir otras macros con parámetros. En este programa, la impedancia del circuito LC en serie, se define por medio de fórmulas predefinidas para la reactancia capacitiva (denominada ahora x_c) y la reactancia inductiva (x_l).

Programa 2.10

```
#include <stdio.h>
#define PI 3.141592
#define X_c(c,f) 1/(2*PI*f*c)
#define X_l(l,f) 2*PI*f*l
#define impedancia_serie(c,l,f) X_l(l,f) - X_c(c,f)

main()
{
    float capacidad; /* Valor del condensador en faradios. */
    float inductancia; /* Valor de la inductancia en henrios. */
    float frecuencia; /* Valor de la frecuencia en hertzios. */
    float imped; /* Valor de la impedancia del circuito en ohmios. */

    printf("Dame la capacidad => ");
    scanf("%f",&capacidad);
    printf("Dame la inductancia => ");
    scanf("%f",&inductancia);
    printf("Dame la frecuencia => ");
    scanf("%f",&frecuencia);

    imped = impedancia_serie(capacidad, inductancia, frecuencia);

    printf("La impedancia de un circuito LC en serie con una\n");
    printf("capacidad de %e faradios, ", capacidad);
    printf("y una inductancia de %e henrios\n", inductancia);
    printf("es %e ohmios.\n", imped)
}
```

Análisis del programa

El Programa 2.10 calcula, al igual que el programa de la sección anterior, el valor de la impedancia de un circuito LC en serie. La diferencia es que en este caso no se utilizan funciones para definir las fórmulas sino directivas del preprocesador (macros):

```
#define PI 3.141592
#define X_c(c, f) 1/(2*PI*f*c)
#define X_l(l, f) 2*PI*f*l
#define impedancia_serie(c, l, f) X_l(l, f) - X_c(c, f)
```

El orden en este caso es importante —`x_c` no puede definirse antes que `impedancia_serie` ya que ésta hace uso de `x_c`. Todo lo que se necesita ahora es utilizar la sentencia definida

```
impedancia_serie(c, l, f)
```

en cualquier momento que se quiera calcular la impedancia para este tipo de circuito.

Asumiendo que el usuario del programa introduce un valor de 2 μF como valor del condensador, 1 mH como valor de la inductancia y una frecuencia de 5 KHz, este programa producirá la siguiente salida:

```
Dame la capacidad => 2e-6
Dame la inductancia => 1e-3
Dame la frecuencia => 5e3
La impedancia de un circuito LC en serie con una
capacidad de 2.000000e-006 faradios
y una inductancia de 1.000000e-003 henrios
es 1.550042e+001 ohmios.
```

Incluyendo la directiva `#define`

Como se verá en la próxima sección sobre implementación y depuración de programas, usted puede crear su propio archivo con sus macros y directivas de preprocesador y grabarlas en disco. También puede dar un nombre al archivo (como, por ejemplo, `miscosas.h`). Este archivo podría contener todas las fórmulas electrónicas que necesita usted así como cualquier otra fórmula de su área de interés técnico. Más tarde, cualquier programa C que usted desee escribir podrá hacer uso de estas fórmulas. Todo lo que se necesita es utilizar la directiva de preprocesador `#include`.

Conclusión

Esta ha sido una sección emocionante. Se ha visto un nuevo aspecto del lenguaje C —una herramienta de programación de gran valor para crear programas técnicos grandes y complejos. Encontrará que todas estas directivas del preprocesador se convertirán en muy útiles y familiares. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 2.5

1. ¿Qué es una directiva del preprocesador?
2. Explique el significado de la directiva `#include`.

3. ¿Qué es una macro?
4. ¿Cómo se definen normalmente las constantes en C?
5. ¿Puede utilizarse un parámetro con la directiva `#define`? Dé un ejemplo.

2.6. IMPLEMENTACIÓN Y DEPURACIÓN DE PROGRAMAS: CREANDO SUS PROPIOS ARCHIVOS DE CABECERA

Presentación

En esta sección descubrirá como hacer en C sus propios **archivos de cabecera** (.h). En ellos puede almacenar todas sus macros de preprocesador. Esto significa que usted puede crear su propia biblioteca de información para su área de tecnología y usarla en cualquiera de sus programas.

Un ejemplo

Considere el Programa 2.11

Programa 2.11

```
#include <stdio.h>
#define cuadrado(x) x*x
#define cubo(x) x*x*x
main()
{
    float total; /* Valor del cuadrado. */
    total = cuadrado(5);
    printf("El cuadrado de 5 es %f \n",total);
}
```

Análisis del programa

El Programa 2.11 es similar al presentado en la sección anterior. Este programa utiliza dos directivas `#define`:

```
#define cuadrado(x) x*x
#define cubo(x) x*x*x
```

La primera, define `cuadrado(x)` como `x*x` y la segunda define `cubo(x)` como `x*x*x`. La función `main()` utiliza la definición `cuadrado(x)` con el propósito de demostración. La ejecución del programa anterior produce la siguiente salida:

El cuadrado de 5 es 25.000000

Grabando sus propias directivas #define

La Figura 2.10 ilustra cómo recoger las directivas #define del Programa 2.11 y guardarlas en un archivo de disco.

Como se muestra en la Figura 2.10, ahora podría utilizar este archivo mediante:

```
#include "miscosas.h"
```

Nótese que se emplean las dobles comillas en vez de los símbolos <> para encerrar el nombre del archivo de cabecera. Esto indica al preprocesador que busque el archivo de cabecera en la unidad o directorio activo. Una vez hecho esto el programa anterior se convierte en el Programa 2.12.

Programa 2.12

```
#include <stdio.h>
#include "miscosas.h"
main()
{
    float total    /* Valor del cuadrado. */
    total = cuadrado(5);
    printf("El cuadrado de 5 es %f\n",total);
}
```

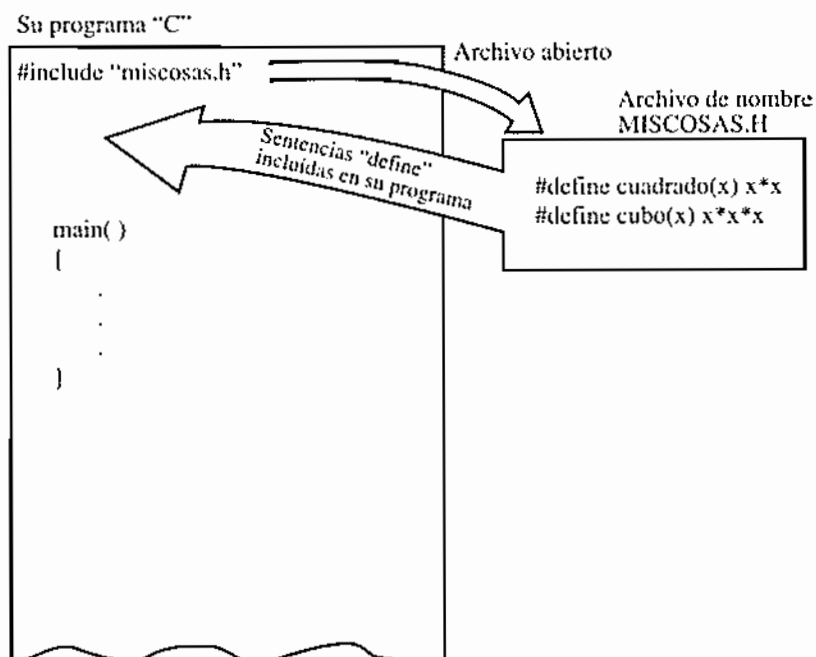


Figura 2.10. Concepto de salvar las sentencias #define.

Un programa de muestra

Considere el Programa 2.13. Este programa podría ser utilizado por un fabricante de circuitos integrados. Las sentencias `#define` se refieren a información acerca de algunos elementos empleados en el proceso de fabricación de circuitos integrados.

Programa 2.13

```
#include <stdio.h>
#define cabecera "Elemento Simbolo Numero atomico Peso atomico\n"
#define Ge "Germanio Ge 32 72.60\n"
#define Si "Silicio Si 14 28.09\n"
#define Au "Oro Au 79 197.2\n"
#define Ag "Plata Ag 47 107.88\n"
#define As "Arsenico As 33 74.91\n"
#define Ge_Wt 72.60
#define Si_Wt 28.09
#define Au_Wt 197.2
#define Ag_Wt 107.88
#define As_Wt 74.91

main()
{
    float total;

    printf(cabecera);
    printf(Ge);
    total = Ge_Wt + Ag_Wt;
    puts("");
    printf("El peso atomico combinado del germanio y la plata
        es %f\n",total);
}
```

Análisis del programa

El Programa 2.13 emplea directivas `#define` para definir las siguientes cadenas de caracteres:

```
#define cabecera "Elemento Simbolo Numero atomico Peso atomico\n"
#define Ge "Germanio Ge 32 72.60\n"
#define Si "Silicio Si 14 28.09\n"
#define Au "Oro Au 79 197.2\n"
#define Ag "Plata Ag 47 107.88\n"
#define As "Arsenico As 33 74.91\n"
```

Como ejemplo, la primera directiva `#define` define el componente léxico `cabecera` como la siguiente cadena de caracteres (nótese el uso de las dobles comillas):

```
"Elemento Simbolo Numero atomico Peso atomico\n"
```

Observe que la cadena de caracteres finaliza con el carácter de nueva línea `\n`. Esto significa que cuando la función `printf()` utilice el componente léxico `cabecera`, el compilador sustituirá esta cadena de caracteres. Así

```
printf(cabecera);
```

producirá

```
Elemento      Simbolo   Numero atomico   Peso atomico
```

cuando se ejecute el programa. Esto mismo ocurrirá con todos los demás componentes léxicos que definan cadenas de caracteres. Por ejemplo,

```
printf(Ge);
```

producirá

```
Germanio  GE   32   72.60
```

cuando se ejecute el programa. De esta manera el código fuente de

```
printf(cabecera);
```

```
printf(Ge);
```

producirá

```
Elemento      Simbolo   Numero atomico   Peso atomico
Germanio      Ge         32              72.60
```

cuando se ejecute el programa.

Ahora el técnico puede fácilmente mostrar la información específica acerca de los elementos atómicos en el programa introduciendo simplemente el símbolo del elemento en una sentencia de salida.

El resto de las directivas `#define` definen valores numéricos para los componentes léxicos. Estos valores representan los pesos atómicos para el correspondiente símbolo del elemento.

```
#define Ge_Wt  72.60
#define Si_Wt  28.09
#define Au_Wt  197.2
#define Ag_Wt  107.88
#define As_Wt  74.91
```

De esta manera el componente léxico `Ge_Wt` será sustituido por el valor 72.60. Asumiendo que la variable total es de tipo `float`

```
total = Ge_Wt + Si_Wt;
```

el preprocesador lo sustituirá por

```
total = 72.60 + 28.09;
```


y dará como resultado un valor de 100.69 para la suma de los pesos atómicos de estos dos elementos.

La ejecución del Programa 2.13 produce la siguiente salida:

Elemento	Simbolo	Numero atomico	Peso atomico
Germanio	Ge	32	72.60
Plata	Ag	47	107.88

El peso atomico combinado del germanio y la plata es 180.480000

Creando el archivo de cabecera.

Para producir un archivo de cabecera que contenga todas las directivas `#define` del Programa 2.13, debe crearse un programa similar al Programa 2.13 y comprobar que todas las directivas `#define` producen los resultados esperados en su programa. A continuación, se eliminan del programa todas las líneas del mismo a excepción de aquellas con la directiva `#define`. Su programa ahora tendrá la apariencia del Programa 2.14

```

Programa 2.14 #define cabecera "Elemento Simbolo Numero atomico Peso atomico\n"
               #define Ge "Germanio Ge 32 72.60\n"
               #define Si "Silicio Si 14 28.09\n"
               #define Au "Oro Au 79 197.2\n"
               #define Ag "Plata Ag 47 107.88\n"
               #define As "Arsenico As 33 74.91\n"
               #define Ge_Wt 72.60
               #define Si_Wt 28.09
               #define Au_Wt 197.2
               #define Ag_Wt 107.88
               #define As_Wt 74.91

```

Las directivas `#define` del Programa 2.13 se grabarán en un archivo denominado `quimica.h`.

Usando su archivo de cabecera

El Programa 2.15 ilustra el uso de su nuevo archivo de cabecera. Observe que su salida es exactamente la misma que la del Programa 2.13.

```

Programa 2.15 #include <stdio.h>
               #include "quimica.h"

```

```

main()
{
    float total;

    printf(cabecera);
    printf(Ge);
    printf(Ag);
    total = Ge_Wt + Ag_Wt;
    puts("");
    printf("El peso combinado del germanio y la plata es %f\n",total);
}

```

La siguiente referencia invocará las definiciones de su archivo de cabecera:

```
#include "quimica.h"
```

Cuando su sistema C vea esto (con las dobles comillas en vez de <>), buscará en el disco un archivo con el mismo nombre, y a continuación traerá el contenido de ese archivo a su programa.

Operadores de concatenación de caracteres y de cadena de caracteres

El estándar ANSI C incluye una operación de preprocesador que permite la concatenación de cadenas de caracteres. El operador de concatenación de cadena de caracteres `##` permite juntar dos componentes léxicos y crear un tercer componente léxico. Por ejemplo, si utilizase las siguientes directivas `#define` en su programa:

```

#define PI_1      3.14159
#define PI_2      6.28318
#define Valor(x)  PI_##x

```

entonces `Valor(1)` sería sustituido por `PI_1`, que a su vez ha sido definido como 3.14159. De la misma manera, `Valor(2)` sería sustituido por `PI_2`, que ha sido definido como 6.28318.

El operador de cadena de caracteres es `#` y permite crear una cadena de caracteres de salida de un operando con un prefijo `#`. Esto se lleva a cabo colocando el operando entre comillas. Por ejemplo, si se define la siguiente directiva `#define` al comienzo de un programa:

```
#define valor_presente(x) printf(#x " = %d", x)
```

entonces cuando se declara `valor_presente(incremento)`, el preprocesador generará la siguiente sentencia:

```
printf("incremento" " = %d", incremento);
```

que se reduce a `printf("incremento = %d", incremento);`

Conclusión

Como puede ver, los archivos de cabecera son una potente extensión al lenguaje C. A medida que desarrolle más programas, su biblioteca de archivos de cabecera (creados por usted para su área particular de tecnología) se convertirán en una de sus herramientas de programación más valiosas. Otro aspecto importante acerca de los archivos de cabecera, es que ellos protegen su código fuente. Una persona que tenga una copia de su código fuente, sin acceso al código de sus archivos de cabecera, puede hacer muy poco para replicar su programa.

Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 2.6

1. Indique la ventaja de crear sus propios archivos de cabecera.
2. Como se ha presentado en esta sección, ¿qué información pueden contener sus archivos de cabecera?
3. ¿Cuál es la extensión que se da a un archivo de cabecera?
4. Explique cómo puede llamar a su archivo de cabecera desde un programa C.

2.7. PROGRAMA DE APLICACIÓN: CIRCUITO RL EN SERIE

Presentación

Esta sección le muestra paso a paso el desarrollo de un programa C que utiliza toda la información que se ha presentado en este capítulo. Se verán las decisiones que se pueden tomar durante el desarrollo de un programa muy especializado. A medida que incremente sus conocimientos sobre C y su versatilidad, se dará cuenta de que hay muchas opciones posibles a la hora de crear su código fuente. Uno de los aspectos más potentes de este lenguaje es esta versatilidad del código fuente.

Debido a esto, lo que se presenta en este programa de aplicación (así como en los otros) es un intento de incrementar la legibilidad y comprensión de lo que hace un programa mientras se conservan las características del lenguaje C.

El problema

Dado el circuito que se muestra en la Figura 2.11, desarrolle un programa C que calcule la caída de tensión en la resistencia y en la inductancia.

Primer paso —definiendo el problema

Recuerde, del programa de aplicación del Capítulo 1, que el primer paso en el diseño de un programa de computador es definir el problema por escrito. Como guía, se pueden considerar como requisitos mínimos los siguientes:

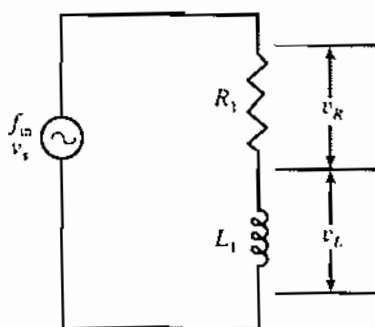


Figura 2.11. Circuito para el programa de aplicación.

- Propósito del programa.
- Entrada que se necesita (fuente).
- Proceso sobre la entrada.
- Salida que se necesita (destino).

El problema puede definirse como sigue:

- Propósito del programa: calcular la caída de tensión en voltios en una resistencia y una inductancia en un circuito RL en serie.
- Entrada que se necesita: valores para los siguientes:
 - Inductancia en henrios.
 - Resistencia en ohmios.
 - Tensión de la fuente en voltios.
 - Frecuencia aplicada en hertzios. (Fuente: el teclado.)
- Proceso sobre la entrada: calcular la caída de tensión en voltios en la inductancia y en la resistencia

$$X_L = 2\pi fL$$

$$Z = \sqrt{R^2 + X_L^2}$$

$$v_L = v_S(X_L/Z)$$

$$v_R = v_S(R/Z)$$

donde

- X_L = Reactancia inductiva en ohmios.
- f = Frecuencia aplicada en hertzios.
- L = Valor de la inductancia en henrios.
- Z = Impedancia del circuito en ohmios.
- R = Valor de la resistencia en ohmios.
- v_L = Caída de tensión en voltios en la inductancia.
- v_R = Caída de tensión en voltios en la resistencia.
- v_S = Voltaje de la fuente en voltios.

- Salida que se necesita: El valor de la caída de tensión en la resistencia, v_R , y el valor de la caída de tensión en la inductancia, v_S , (salida por la pantalla).

De esta descripción, debería quedar claro a cualquiera lo que se requiere que haga el programa (y, también muy importante, lo que no). El siguiente paso en el desarrollo de este programa es desarrollar un algoritmo. Puede considerarse a un algoritmo de forma similar a una receta de un pastel. Un algoritmo es una explicación paso a paso de lo que el programa hará exactamente, escrito de forma tan concisa como sea posible (al igual que debería ser escrita la receta de un pastel).

Desarrollando el algoritmo

Una vez definido el problema por escrito, incluyendo las cuatro áreas necesarias (propósito del programa, entrada que se necesita, proceso sobre la entrada y salida que se necesita), puede desarrollarse el algoritmo de forma sencilla. La mayoría de los programas relacionados con la tecnología seguirán el orden que se muestra a continuación.

Todo lo que sigue debe comentarse (entre `/*` y `*/`, excepto los prototipos de las funciones) y colocarse en el bloque del programador —algunas veces referido como prólogo.

- I. Información del programa
 - A. Nombre del programa
 - B. Nombre del programador
- II. Explicación del programa
 - A. Qué hace el programa
 - B.Cuál es la entrada que se necesita
 - C. Qué proceso se va a llevar a cabo
 - D. Cuáles serán los resultados
- III. Describir todas las funciones
 - A. Uso de prototipos de funciones
 - B. Explicar el propósito de cada prototipo
 - C. Definir todas las variables y constantes

Los siguientes son los pasos de programación que se utilizan en la mayoría de los programas técnicos:

- IV. Explicar el programa al usuario
 - A. Propósito del programa
 - B. Qué necesita hacer el usuario
 - C. Qué hace el programa
 - D. Cuáles serán los resultados finales
- V. Obtener la información del usuario
 - A. Pedir los datos de entrada al usuario
 - B. Confirmar los datos de entrada.
- VI. Llevar a cabo el proceso
 - A. Realizar el proceso.

- VII. Mostrar los resultados
 - A. Confirmar lo que el usuario ha introducido
 - B. Mostrar todas las unidades
- VIII. Preguntar si se quiere repetir el programa
 - A. Preguntar al usuario si desea repetir el programa
 - B. Ofrecer la opción de evitar instrucciones

Los contenidos anteriores servirán como guía para el desarrollo de este programa.

Primera codificación

La primera etapa en la codificación del programa consistirá en desarrollar el bloque del programador, compilarlo, grabarlo en un archivo y a continuación imprimir una copia. No se produce ninguna salida, pero el propósito es asegurarse de que la documentación del programa ya ha empezado. Esta primera etapa se muestra en el Programa 2.16

Programa 2.16 #include <stdio.h>

```

/*
Programa: Caída de tensión en circuito RL.
Desarrollado por: Un buen programador

Descripción: Este programa calcula la caída de tensión en
              voltios en una inductancia y una resistencia en
              serie en un circuito RL. El usuario debe introducir
              el valor de la inductancia en henrios, el valor
              la resistencia en ohmios, la frecuencia en
              hertzios y la tensión de la fuente en voltios.

Variables: ninguna

Constantes: ninguna

Prototipos de funciones:
*/

void explicacion_del_programa(void);
/* Esta función explica al usuario la operación que lleva */
/* a cabo el programa. */

void obtener_valores(void);
/* Esta función lee los valores de la resistencia, la */
/* inductancia, el voltaje de la fuente y la frecuencia */
/* aplicada, y calcula y muestra los resultados. */

main()
```

```

{
}

void explicacion_del_programa()
{
}

void obtener_valores()
{
}

```

Análisis del programa

El Programa 2.16 incluye el título del programa y el nombre del programador. A continuación sigue una breve descripción del mismo. Observe que la descripción incluye todas las unidades de las medidas. Observe también, que el programador ha decidido emplear un único bloque de programación para leer los valores del usuario del programa, realizar los cálculos y mostrar los valores por pantalla. A medida que se avance en el desarrollo del programa, puede que se considere este bloque demasiado grande, y se decida dividir éste en al menos dos bloques más con el objetivo de mejorar su legibilidad.

Introducción del primer código

El siguiente paso para este programa es realizar la codificación del bloque de explicación para el usuario. Una vez hecho éste, se compila el programa y se ejecuta. El bloque para este programa es el que sigue:

```

main()
{
    explicacion_del_programa(); /* explicar el programa al usuario. */
    exit(0);
}

void explicacion_del_programa() /* Explicar el programa al usuario. */
{
    printf("\n\n Este programa calcula la caida de tension\n");
    printf("en una inductancia y una resistencia en serie en\n");
    printf("un circuito RL. Debe introducir los valores de la\n");
    printf("resistencia en ohmios, la inductancia en henrios,\n");
    printf("la frecuencia aplicada en hertzios y el voltaje\n");
    printf("de la fuente en voltios.\n\n");
}

```

Observe que `main()` contiene ahora una llamada a la función `explicacion_del_programa()`. Ésta se incluye para que se pueda apreciar la salida. Es importante obser-

var el empleo de dos caracteres de nueva línea `\n` al comienzo de la primera función `printf`. Esto es una buena costumbre puesto que hace que el texto sea más fácil de leer.

Decisión de las fórmulas

El programador debe en este momento decidir si las fórmulas utilizadas para la solución del problema serán puestas en una función aparte o en un archivo de cabecera. Se ha decidido crear un archivo de cabecera para estas fórmulas debido a que probablemente serán empleadas de nuevo en futuros programas.

Las fórmulas se desarrollaron primero utilizando directivas `#define` como se muestra a continuación.

```
#include <stdio.h>
#include <math.h>
#define PI 3.141592
#define cuadrado(x) x*x
#define X_L(f,l) 2*PI*f*l
#define Z(f,l,r) sqrt(cuadrado(X_L(f,l)) + cuadrado(r))
#define V_L(f,l,r,v) v*(X_L(f,l)/Z(f,l,r))
#define V_R(f,l,r,v) v*(r/Z(f,l,r))
```

Estas sentencias se sitúan al comienzo del programa y antes del bloque del programador. Observe que los archivos `#include` se utilizan para la entrada/salida y para las funciones matemáticas. El programador ha seguido las fórmulas que se indicaron anteriormente.

Añadiendo el bloque de cálculo y de impresión

Una vez tomada la decisión de emplear archivos de cabecera para las fórmulas, se ha decidido emplear un bloque de programa diferente para realizar los cálculos y mostrar los valores.

Los archivos de cabecera no deberían incluirse hasta que se haya comprobado la corrección de la información a ser incluida en ellos.

Los resultados combinados de todas las etapas se muestran en el Programa 2.17.

Programa 2.17

```
#include <stdio.h>
#include <math.h>
#define PI 3.141592
#define cuadrado(x) x*x
#define X_L(f,l) 2*PI*f*l
#define Z(f,l,r) sqrt(cuadrado(X_L(f,l)) + cuadrado(r))
#define V_L(f,l,r,v) v*(X_L(f,l)/Z(f,l,r))
#define V_R(f,l,r,v) v*(r/Z(f,l,r))
```



```
/*
```

```
Programa: Caída de tensión en circuito RL.
```

```
Desarrollado por: Un buen programador
```

```
Descripción: Este programa calcula la caída de tensión en
               voltios en una inductancia y una resistencia en
               serie en un circuito RL. El usuario debe introducir
               el valor de la inductancia en henrios, el valor
               la resistencia en ohmios, la frecuencia en
               hertzios y la tensión de la fuente en voltios.
```

```
Variables:
```

```
    resistencia    = Valor de la resistencia en serie en ohmios.
    inductancia    = Valor de la inductancia en serie en henrios.
    frecuencia     = Valor de la frecuencia en hertzios.
    voltaje        = Valor del voltaje de la fuente en voltios.
    v_inductancia  = Valor del voltaje en la inductancia.
    v_resistencia  = Valor del voltaje en la resistencia.
```

```
Constantes: ninguna
```

```
Prototipos de funciones:
```

```
*/
```

```
float calcular_y_mostrar(float f, float l, float r, float v);
```

```
/*
```

```
    f = Frecuencia en hertzios.
    l = Inductancia en henrios.
    r = Resistencia en ohmios.
    v = Voltaje de la fuente en voltios.
    Esta función calcula y muestra la caída de tensión en
    la resistencia y en la inductancia.
```

```
*/
```

```
void explicacion_del_programa(void);
```

```
/*
```

```
    Esta función explica al usuario la operación que lleva
    a cabo el programa. */
```

```
void obtener_valores(void);
```

```
/*
```

```
    Esta función lee los valores de la resistencia, la
    inductancia, el voltaje de la fuente y la frecuencia
    aplicada, y calcula y muestra los resultados.
```

```
*/
```

```
main()
```

```
{
```

```
    explicacion_del_programa(); /* Explicar el programa al usuario. */
```

```

        obtener_valores();          /* Obtener los valores del circuito.*/
        exit(0);
    }

void explicacion_del_programa() /* Explicar el programa al usuario. */
{
    printf("\n\n Este programa calcula la caida de tension\n");
    printf("en una inductancia y una resistencia en serie en\n");
    printf("un circuito RL. Debe introducir los valores de la\n");
    printf("resistencia en ohmios, la inductancia en henrios,\n");
    printf("la frecuencia aplicada en hertzios y el voltaje\n");
    printf("de la fuente en voltios.\n\n");
}

void obtener_valores() /* Obtener los valores del circuito.*/
{
    float resistencia; /* Valor de la resistencia en serie en ohmios */
    float inductancia; /* Valor de la inductancia en serie en henrios*/
    float frecuencia; /* Valor de la frecuencia en hertzios */
    float voltaje; /* Valor del voltaje de la fuente en voltios */

    printf("Introduzca los siguientes valores:\n");
    printf("Resistencia en ohmios      => ");
    scanf("%f",&resistencia);
    printf("Inductancia en henrios      => ");
    scanf("%f",&inductancia);
    printf("Frecuencia en hertzios          => ");
    scanf("%f",&frecuencia);
    printf("Voltaje de la fuente en voltios  => ");
    scanf("%f",&voltaje);

    calcular_y_mostrar(frecuencia, inductancia, resistencia, voltaje);
}

float calcular_y_mostrar(float f, float l, float r, float v)
{
    float v_inductancia; /* Valor del voltaje en la inductancia */
    float v_resistencia; /* Valor del voltaje en la resistencia */

    v_inductancia = V_L(f,l,r,v);
    /* Calcula el voltaje en la inductancia */

    v_resistencia = V_R(f,l,r,v);
    /* Calcula el voltaje en la resistencia */

    printf("\n\nPara un circuito RL en serie que conste de una");
    printf("inductancia de %e henrios y una resistencia de %e\n",l,r);
    printf("ohmios\n",l,r);
    printf("con una frecuencia aplicada de %e hertzios\n",f);
    printf("y fuente de %e voltios,\n",v);
    printf("la caida de tension en los componentes es: \n\n");
}

```

```

printf("Voltaje en la inductancia => %e voltios\n",
      v_inductancia);
printf("Voltaje en la resistencia => %e voltios\n",
      v_resistencia);
}

```

Análisis del programa

Uno de los aspectos importantes que se han añadido al Programa 2.17 es la explicación de las variables utilizadas como argumentos formales en el prototipo de la función que realiza los cálculos y muestra los resultados:

```

float calcular_y_mostrar(float f, float l, float r, float v);
/*
    f = Frecuencia en hertzios.
    l = Inductancia en henrios.
    r = Resistencia en ohmios.
    v = Voltaje de la fuente en voltios.
    Esta función calcula y muestra la caída de tensión en
    la resistencia y en la inductancia.
*/

```

Observe que los comentarios definen exactamente cada uno de los identificadores utilizados como parámetros formales en la función. A continuación, como siempre, se muestra el propósito de la función. Es importante apreciar que el prototipo no se encuentra comentado entre `/*` y `*/`, puesto que éste sirve como una instrucción efectiva para el compilador de C.

La función `obtener_valores` llama a la función `calcular_y_mostrar`. Los parámetros reales que utiliza la función que lee los datos del usuario del programa son los siguientes:

```
calcular_y_mostrar(frecuencia, inductancia, resistencia, voltaje);
```

La definición de la función `calcular_y_mostrar` utiliza la misma cabecera que su prototipo (con la excepción de que no aparece el punto y coma). Observe de que forma tan fácil se codifican los cálculos para estas fórmulas complejas:

```

float calcular_y_mostrar(float f, float l, float r, float v)
{
    float v_inductancia;    /* Valor del voltaje en la inductancia */
    float v_resistencia;    /* Valor del voltaje en la resistencia */

    v_inductancia = V_L(f,l,r,v);
    /* Calcula el voltaje en la inductancia */

    v_resistencia = V_R(f,l,r,v);
    /* Calcula el voltaje en la resistencia */
}

```

```

printf("\n\nPara un circuito RL en serie que conste de una");
printf("inductancia de %e henrios y una resistencia de %e
      ohmios\n",l,r);
printf("con una frecuencia aplicada de %e hertzios\n",f);
printf("y fuente de %e voltios,\n",v);
printf("la caída de tensión en los componentes es: \n\n");
printf("Voltaje en la inductancia => %e voltios\n", v_inductancia);
printf("Voltaje en la resistencia => %e voltios\n", v_resistencia);
)

```

La salida de este programa muestra los valores introducidos por el usuario empleando notación exponencial junto con todas las unidades de medidas. Las soluciones también se muestran utilizando notación exponencial junto con sus unidades de medida.

Programa final

En este programa no se ha presentado al usuario la opción de repetir los cálculos puesto que no se ha explicado aun la forma de hacer esto en C. La última etapa de la codificación es crear un archivo de cabecera llamado `serie_rl.h` que contenga la siguiente información:

```

#include <stdio.h>
#include <math.h>
#define PI 3.141592
#define cuadrado(x) x*x
#define X_L(f,l) 2*PI*f*l
#define Z(f,l,r) sqrt(cuadrado(X_L(f,l)) + cuadrado(r))
#define V_L(f,l,r,v) v*(X_L(f,l)/Z(f,l,r))
#define V_R(f,l,r,v) v*(r/Z(f,l,r))

```

De esta forma el programa final incluirá al comienzo del mismo las siguientes sentencias `#include`:

```

#include <stdio.h>
#include <math.h>
#include "serie_rl.h"

```

Conclusión

Ha recorrido un gran camino en su observación del desarrollo de un programa C. Es importante que aplique los principios que se han presentado en esta sección a los programas que usted desarrolle. La planificación cuidadosa y la documentación de programas son aspectos muy importantes de una buena programación. Estas etapas son seguidas por los programadores profesionales que conocen el viejo dicho: «Cuanto más pronto se empiece a programar, más se tardará en depurar el programa final».

Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 2.7

1. Indicar uno de los principales objetivos en el desarrollo del programa de esta sección.
2. ¿Qué debería incluirse en el bloque del programador?
3. Explicar la razón del uso de las directivas `#define`.
4. ¿Cuál es la última cosa que se pregunta normalmente al usuario de un programa tecnológico típico?

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido para permitirle adquirir una valiosa experiencia y, lo que es más importante, para tener una realimentación inmediata de lo que los conceptos y mandatos presentados en este capítulo hacen. Además son divertidos.

Ejercicios

1. Prediga qué hará el Programa 2.18 y luego pruébelo.

Programa 2.18 `#include <stdio.h>`

```

void Primera(void);
void Segunda(void);

main()
{
    puts("Esto es main()");
    Primera();
}

void Primera()
{
    puts("Esta es Primera()");
    Segunda();
}

void Segunda()
{
    puts("Esta es Segunda()");
}

```

2. El Programa 2.19 se diferencia del anterior en que se necesita presionar `Ctrl-Break` para abortarlo. La diferencia se encuentra en que la función se llama a sí misma. Este es un ejemplo de recursividad.

Programa 2.19 #include <stdio.h>

```

void Recursiva(void);

main()
{
    Recursiva();
}

void Recursiva()
{
    puts("Este es un ejemplo de recursividad.");
    Recursiva();
}

```

3. C es un lenguaje tan versátil que puede hacer que se parezca a casi cualquier otro lenguaje de programación. El Programa 2.20 redefine órdenes de C. Estas podrían almacenarse en un archivo de cabecera denominado `nuevo_codigo.h`. Pruébalo para demostrarse que esto es así.

Programa 2.20

```

#include <stdio.h>
#define START main(){
#define END }
#define WRITE puts
START
WRITE ("Es esto un programa?");
END

```

Autoevaluación

DIRECTRICES

Utilice el Programa 2.16 para responder a las siguientes preguntas:

Preguntas:

1. ¿Cuántas funciones se han definido en el programa? Nómbralas.
2. ¿Cuál es el número total de funciones utilizadas en el programa? Indique aquellas que no se han nombrado en la pregunta 1.
3. Indique los identificadores empleados como parámetros formales en el programa.
4. Indique los identificadores empleados como parámetros reales en el programa. ¿Qué funciones pasan variables a otra función?
5. Explique como se pasan valores de una función a otra utilizando el Programa 2.16 como ejemplo.
6. ¿Cuántos identificadores de variables se utilizan en el programa? Nómbralos.
7. ¿Qué cambio mínimo debería hacerse en el programa para que mostrara el valor de la reactancia inductiva (X_L) .
8. Explique si el programa acepta datos introducidos por el usuario en notación exponencial.

Problemas de fin de capítulo

Conceptos generales

Sección 2.1

1. Explique qué se entiende por estructura de bloques.
2. ¿Por qué no es necesario utilizar la sentencia `goto` en un programa C?
3. ¿Qué tipo de bloque tiene la capacidad de repetir una parte del programa?
4. ¿Qué tipo de bloque tiene la capacidad de ejecutar una secuencia diferente de un código de programa.

Sección 2.2

5. En un programa C, ¿qué es lo que informa al compilador sobre las funciones que serán definidas en el programa?
6. Indique las diferencias entre el prototipo de una función y la cabecera empleada para definir la función.
7. ¿Cómo se realiza una llamada a una función?
8. ¿Qué función provoca la terminación normal de un programa? ¿Qué valor se utiliza para indicar una terminación normal?

Sección 2.3

9. ¿Qué tipo se asigna a una función cuando ésta no devuelve ningún valor?
10. Indique qué se utiliza en una función para definir el número y tipo de los datos que serán pasados a la función cuando está sea llamada.
11. ¿Qué se utiliza para devolver un valor de una función a aquella que realizó la llamada?

Sección 2.4

12. ¿A qué se llama cuando una función se llama a sí misma?
13. ¿Puede llamar una función a más de una función?
14. ¿Existe alguna diferencia en el orden en el que las funciones son llamadas?
15. ¿Puede una función a la que se ha llamado llamar a su vez a otra función?

Sección 2.5

16. ¿Qué es una directiva del preprocesador?
17. Indique las directivas del preprocesador que se han presentado en este capítulo.
18. Explique cómo se definen normalmente las constantes en C.
19. Dé un ejemplo de cómo se pueden utilizar parámetros con la directiva `#define`.

Sección 2.6

20. Explique cómo puede hacer un archivo con sus propias directivas `#define` para utilizarlas en otros programas.

21. ¿Cómo se invoca a un archivo de cabecera?
22. ¿Cuál es la extensión aceptada que se da a los archivos de cabecera en C?

Diseño de programas

En el desarrollo de los siguientes programas haga uso del método descrito en la Sección 2.7. Para cada programa, documente su diseño y manténgalo con su programa. Este proceso debería incluir el contenido del diseño que indique el propósito del programa, la entrada que se necesita, el proceso que se realiza sobre la entrada y la salida que se necesita, así como el algoritmo del programa. Asegúrese de que incluye toda esta documentación en su programa final. Esta debería contener, al menos, el bloque del programador, los prototipos de las funciones y una descripción de cada función así como de los parámetros formales que se utilicen.

23. Desarrolle un programa en C que calcule la potencia entregada por una fuente de alimentación. La relación matemática a utilizar es:

$$P = I \times E$$

donde

P = La potencia entregada en vatios.

I = La corriente de la fuente en amperios.

E = El voltaje de la fuente en voltios.

No utilice directivas `#define` para este programa. Haga uso de funciones para cada una de las partes importantes del mismo. Una función explicará al usuario la tarea que lleva a cabo el programa, otra leerá los valores del usuario y los pasará a otra función que definirá la fórmula de la potencia.

24. Modifique el programa desarrollado en el Problema 23 para que la fórmula de la potencia se incluya en un archivo de cabecera en vez de en una función.
25. Desarrolle un programa en C que determine el número de bytes que se necesitan para almacenar una matriz de N por M elementos. Haga el cálculo para elementos que ocupen un bit, un byte y dos bytes.
26. Escriba un programa en C que calcule el volumen de tres habitaciones diferentes. Asuma que cada habitación es un rectángulo perfecto. El usuario introducirá el alto, el ancho y la longitud de cada habitación. El programa devolverá el volumen para cada habitación así como el volumen total de las tres habitaciones.

27. Desarrolle un programa en C que calcule la factura de un paciente de un hospital. El usuario introducirá:

1. El número de días en el hospital.
2. El coste de la consulta.
3. El precio de las medicinas.
4. Costes diversos.
5. El precio por día.
6. La deducción del seguro.

El programa debe calcular lo siguiente:

1. El coste total
 2. El coste total menos la deducción del seguro
 3. El coste total menos el coste de las medicinas y la deducción.
28. Implemente la siguiente fórmula como una función:

$$X = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

Operaciones sobre datos y toma de decisiones

Objetivos

En este capítulo podrá aprender:

1. El significado y uso de los operadores de comparación.
2. La diferencia entre una bifurcación abierta y una cerrada.
3. La aplicación práctica de las bifurcaciones abiertas y cerradas.
4. El uso de operaciones lógicas y binarias en C.
5. La programación de bifurcaciones mediante el uso de operadores lógicos y de comparación.
6. Cómo seleccionar una entre varias alternativas.
7. Cómo trabajar con tipos de datos diferentes y realizar conversiones entre ellos.
8. El uso de la compilación condicional.
9. El desarrollo de programas a partir de sus correspondientes diagramas de flujo.

Palabras clave

Operadores de comparación
VERDADERO
FALSO
Asignación
Bifurcación abierta
Sentencia `if`
Sentencia condicional
Sentencia compuesta
Bifurcación cerrada

Sentencia `if...else`
Sentencia `if...else` compuesta
Operadores binarios
Operadores Booleanos
Complemento binario (bit a bit)
AND binario (bit a bit)
OR binario (bit a bit)
XOR binario (bit a bit)
Desplazamiento de bits

AND lógico
OR lógico
NOT lógico
Rango

Conversión de tipos
Valor-i
Compilación condicional
Suplentes del programa

Contenido

3.1. Operadores de comparación	3.8. Más sobre la sentencia <code>switch</code> y el operador condicional
3.2. La bifurcación abierta	3.9. Depuración y desarrollo de programas
3.3. La bifurcación cerrada	3.10. Programa de aplicación: La reparación de un robot
3.4. Operaciones Booleanas binarias	3.11. Programas de aplicación adicionales
3.5. Operaciones lógicas	
3.6. Conversión de tipos	
3.7. La sentencia <code>switch</code>	
-

Introducción

En este capítulo se mostrará cómo desarrollar programas en C que hagan que el computador parezca «inteligente». El lector aprenderá a crear programas que tomen decisiones de forma que su comportamiento pueda depender del valor de un dato que introduzca un usuario o del resultado de un cálculo previo.

El capítulo comienza explicando las distintas relaciones que pueden existir entre dos cantidades y cómo dichas relaciones se usan en el proceso de toma de decisiones de un programa. Se presentarán los operadores y el simbolismo utilizado en C para expresar estas relaciones lógicas.

El capítulo termina con una interesante aplicación práctica que consiste en el desarrollo de un programa para la reparación de un robot. Al finalizar el capítulo se podrán escribir programas que permitan analizar y resolver numerosos problemas dentro del campo de la tecnología.

3.1. OPERADORES RELACIONALES

Presentación

En esta sección se presentan los **operadores relacionales de comparación** que permiten conocer qué relaciones existen entre dos cantidades. Estos conocimientos servirán como fundamento para el material expuesto en la próxima sección donde se aprenderá a crear programas que sean capaces de tomar decisiones.

Operadores relacionales

Un operador relacional o de comparación es un símbolo que indica una relación entre dos cantidades. Estas cantidades pueden ser variables, constantes o funciones. Lo im-

Tabla 3.1. Operadores relacionales (de comparación) usados en C

Símbolo	Significado	Ejemplos VERDADEROS	Ejemplos FALSOS
>	Mayor que	$5 > 3$ $(3 + 8) > (5 - 2)$	$3 > 5$ $(12/6) > 18$
>=	Mayor o igual que	$10 \geq 10$ $(3 * 4) \geq 8/2$	$3 \geq 5$ $8 + 5 \geq 10 * 15$
<	Menor que	$3 < 5$ $12/3 < 12 * 3$	$5 < 3$ $9 - 2 < 3 + 1$
<=	Menor o igual que	$3 \leq 15$ $18/6 \leq 9/3$	$15 \leq 3$ $12 + 4 \leq 12/4$
==	Igual que	$5 == 5$ $2 + 7 == 18/2$	$10 == 5$ $8 - 5 == 2 + 4$
!=	Distinto que	$8 != 5$ $8 - 5 != 2 + 4$	$5 != 5$ $24/6 != 12/3$

portante es resaltar que dichas relaciones o bien son **VERDADERAS** o bien **FALSAS**, pero no hay ninguna otra posibilidad.

La Tabla 3.1 muestra las operaciones de comparación disponibles en C.

Los operadores de comparación devuelven un 1 para indicar una condición VERDADERA y un 0 para indicar una FALSA. El Programa 3.1 ilustra este comportamiento.

Programa 3.1 `#include <stdio.h>`

```
main()
{
    float valor_logico;    /* Valor numérico de una expresión de
                           comparación */
    printf("Valores logicos de las siguientes relaciones:\n\n");
    valor_logico = (3 > 5);
    printf("(3 > 5) es %f\n",valor_logico);
    valor_logico = (5 > 3);
    printf("(5 > 3) es %f\n",valor_logico);
    valor_logico = (3 >= 5);
    printf("(3 >= 5) es %f\n",valor_logico);
    valor_logico = (15 >= 3*5);
    printf("(15 >= 3*5) es %f\n",valor_logico);
    valor_logico = (8 < (10-2));
    printf("(8 < (10-2)) es %f\n",valor_logico);
    valor_logico = (2*3 < 24/3);
    printf("(2*3 < 24/3) es %f\n",valor_logico);
    valor_logico = (10 < 5);
    printf("(10 < 5) es %f\n",valor_logico);
    valor_logico = (24 <= 15);
    printf("(24 <= 15) es %f\n",valor_logico);
}
```

```

    valor_logico = (36/6 <= 2*3);
    printf("(36/6 <= 2*3) es %f\n", valor_logico);
    valor_logico = (8 == 8);
    printf("(8 == 8) es %f\n", valor_logico);
    valor_logico = (12+5 == 15);
    printf("(12+5 == 15) es %f\n", valor_logico);
    valor_logico = (8 != 5);
    printf("(8 != 5) es %f\n", valor_logico);
    valor_logico = (15 != 3*5);
    printf("(15 != 3*5) es %f\n", valor_logico);
}

```

Análisis del programa

La ejecución del programa da como resultado:

Valores logicos de las siguientes relaciones:

```

(3 > 5) es 0.000000
(5 > 3) es 1.000000
(3 >= 5) es 0.000000
(15 >= 3*5) es 1.000000
(8 < (10-2)) es 0.000000
(2*3 < 24/3) es 1.000000
(10 < 5) es 0.000000
(24 <= 15) es 0.000000
(36/6 <= 2*3) es 1.000000
(8 == 8) es 1.000000
(12+5 == 15) es 0.000000
(8 != 5) es 1.000000
(15 != 3*5) es 0.000000

```

En esta salida se puede observar que los operadores de comparación devuelven únicamente dos valores: un 1 cuando la operación es VERDADERA y un 0 cuando es FALSA. A continuación se comenta con más detalle el siguiente fragmento del programa:

```

valor_logico = (3 > 5);
printf("(3 > 5) es %f\n", valor_logico);
valor_logico = (5 > 3);
printf("(5 > 3) es %f\n", valor_logico);

```

La variable `valor_logico` se ha declarado de tipo `float`. El lector puede intentar usar otros tipos. En la primera línea se está asignando a dicha variable el valor de la operación `(3 > 5)`. Esta sentencia es FALSA y, por lo tanto, devolverá un 0, que se muestra en la salida usando un tipo `float` (`%f`) como 0.000000. En la segunda

parte, se asigna a la variable `valor_logico` el resultado de la operación $(5 > 3)$. Puesto que se trata de una sentencia VERDADERA, devolverá un 1 que, al imprimirlo como un tipo `float`, aparecerá en la salida como 1.000000.

El resto del programa realiza operaciones similares a las comentadas.

Igualdad

Puede parecer extraño que en C se use el símbolo `==` (dos iguales o «igual-igual») para expresar la relación de igualdad. Es comprensible que muchos de los que empiezan a trabajar con este lenguaje puedan pensar que el operador `=` (un único símbolo igual) sirve para expresar la relación de igualdad. Sin embargo, este símbolo, a diferencia de lo que significa en el campo de la matemática, representa una asignación. La Figura 3.1 ilustra el concepto de asignación.

Es importante distinguir entre el operador de asignación (`=`) y el operador de comparación de igualdad (`==`). El operador de asignación toma el valor correspondiente al lado derecho de la sentencia de asignación y lo almacena en la posición de memoria de la variable que aparece en el lado izquierdo. El operador de comparación de igualdad hace algo totalmente diferente: compara el valor almacenado en una posición de memoria con el almacenado en otra. No hay transferencia de datos de una posición a otra.

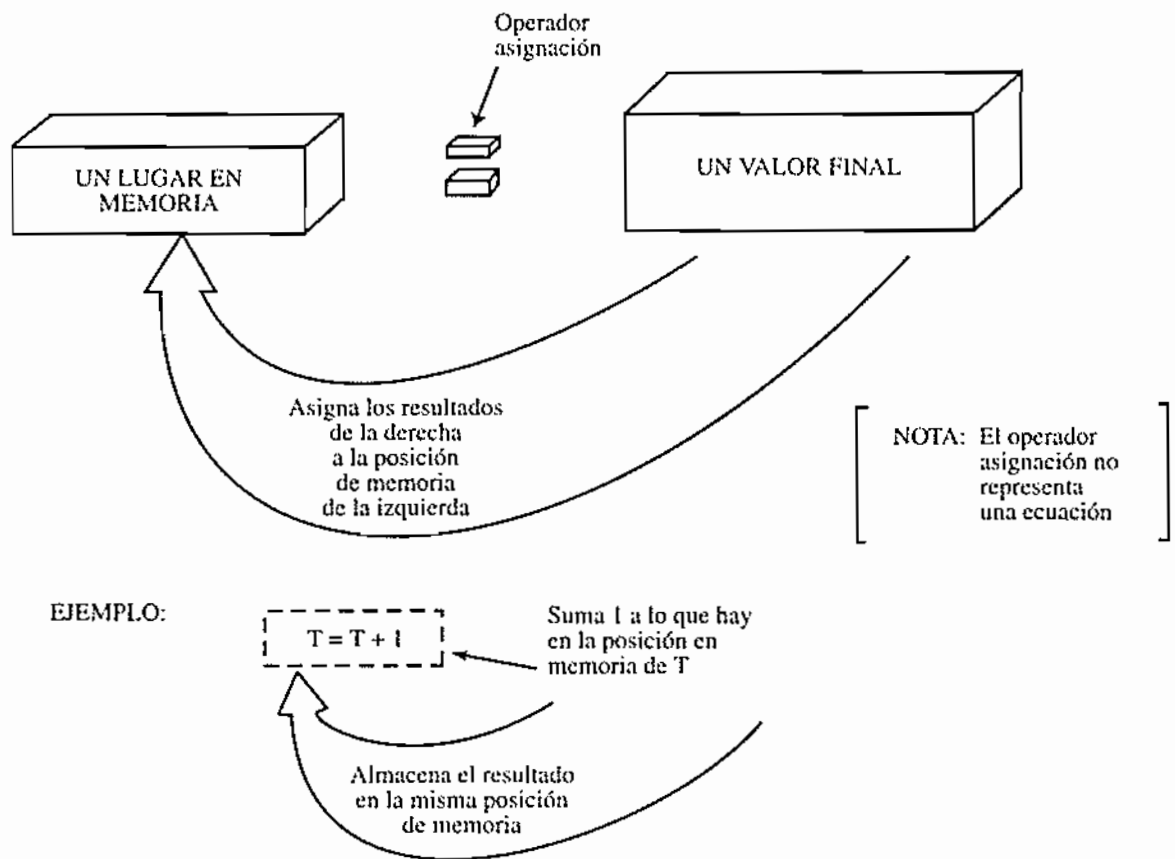


Figura 3.1. Concepto de asignación.

El siguiente ejemplo sirve de repaso del uso de los operadores de comparación.

Ejemplo 3.1 Suponiendo que se han realizado las siguientes asignaciones:

$a = 5, \quad b = 10, \quad c = 5$

Especifique cuáles serán los valores de las siguientes comparaciones:

A. $a == 5$ C. $a >= c$
B. $c == b$ D. $b != a$

Solución

- A. $a == 5$ es VERDADERA, el valor es 1.
B. $c == b$ es FALSA, el valor es 0.
C. $a >= c$ es VERDADERA, el valor es 1.
D. $b != a$ es VERDADERA, el valor es 1.

Conclusión

En esta sección se han presentado los operadores de comparación de C, mostrándose los símbolos que se usan en C para representar dichos operadores. Asimismo, se ha explicado que dichos operadores sólo pueden tomar dos valores: VERDADERO y FALSO, que realmente se corresponden con los valores numéricos 1 y 0 respectivamente. Por último, se ha mostrado la importante diferencia que existe entre el operador de asignación ($=$) y el de igualdad ($==$).

En la próxima sección se aplicará lo aprendido en esta sección al desarrollo de programas con capacidad de toma de decisiones. Antes de pasar a ella, compruebe su comprensión de los conceptos presentados en esta sección mediante el siguiente repaso.

Repaso de la Sección 3.1

1. ¿Qué son los operadores de comparación?
2. Enumere todos los operadores de comparación de C.
3. ¿Qué dos condiciones puede tomar un operador de comparación?
4. Explique qué significa que un operador de comparación devuelve un valor.
5. Explique la diferencia entre los símbolos $=$ y $==$.

3.2. LA BIFURCACIÓN ABIERTA

Presentación

Esta sección mostrará cómo se usan los operadores de comparación presentados en la sección anterior para poder tomar decisiones dentro del código de un programa.

La bifurcación abierta presentada en esta sección es quizá el mecanismo de toma de decisiones más sencillo que proporciona el computador. Aun así, este mecanismo supondrá un avance importante en los conocimientos de C del lector.

Idea básica

La Figura 3.2 ilustra la idea básica de la **bifurcación abierta** mostrando dos importantes características de este tipo de bifurcaciones. En primer lugar, el flujo del programa siempre continúa hacia adelante. En segundo lugar, independientemente de que se tome o no el camino alternativo, siempre se ejecutará el resto del programa. En el lenguaje C, este tipo de bifurcación se corresponde con la **sentencia if**.

La sentencia if

La sentencia **if** es de tipo **condicional** ya que su ejecución depende de una condición. La sintaxis de esta sentencia es la siguiente:

```
if (expresion) sentencia
```

Lo cual significa que si **expresion** es VERDADERA se ejecutará **sentencia**, mientras que si es FALSA no se ejecutará. El Programa 3.2 ilustra el uso de esta sentencia.

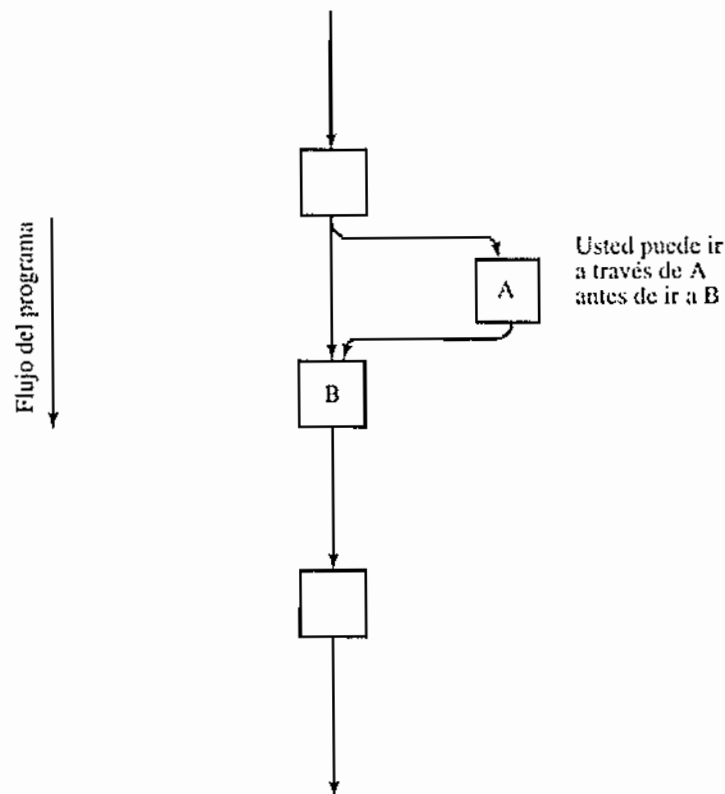


Figura 3.2. Concepto de rama abierta.

Programa 3.2

```
#include <stdio.h>
main()
{
    float numero; /* Valor proporcionado por el usuario. */

    printf("Deme un numero del 1 al 10 => ");
    scanf("%f",&numero);
    if (numero > 5)
        printf("El numero es mayor que 5.\n");
    printf("El numero leído fue %f.",numero);
}
```

Análisis del programa

El Programa 3.2 pide al usuario que introduzca un número del 1 al 10. Si éste introduce un número del 1 al 5 (por ejemplo, el 3), el programa imprimirá el mensaje:

El numero leído fue 3.

Si el usuario introduce un número mayor que 5 (como, por ejemplo, el 7), el programa mostrará los siguientes mensajes:

El numero es mayor que 5.

El numero leído fue 7

Se puede apreciar el uso de la sentencia `if` en el siguiente fragmento:

```
if (numero > 5)
    printf("El numero es mayor que 5.\n");
```

La expresión es una comparación que evalúa el valor introducido por el usuario para determinar si es mayor que 5. Si lo es, la expresión es VERDADERA y se ejecutará la siguiente sentencia:

```
printf("El numero es mayor que 5.\n");
```

En caso contrario, no se ejecutará esa sentencia.

Nótese como la sentencia dentro del `if` está sangrada con respecto al resto del programa. Asimismo, observe la ausencia de un punto y coma entre la expresión y la sentencia (no hay punto y coma detrás de `if (numero > 5)`).

Es importante resaltar que el programa siempre avanza hacia adelante y que la parte condicional podrá ejecutarse o no (dependiendo de si se cumple la condición). Sin embargo, con independencia del valor introducido por el usuario, siempre se ejecutará el resto del programa (siempre aparecerá la salida de la última llamada a la función `printf()`).

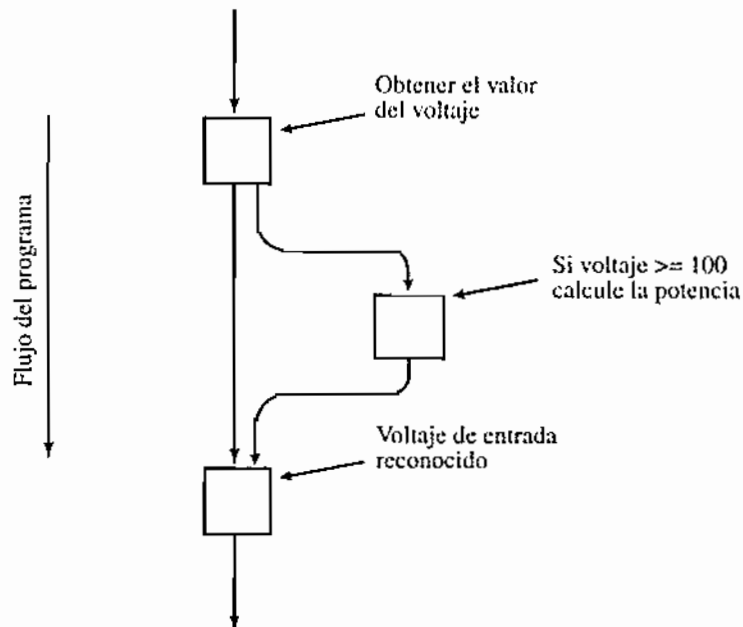


Figura 3.3. Cálculo opcional de disipación de potencia.

Un sentencia compuesta

El Programa 3.2 ilustró el uso de una condición `if` que incluía una sola sentencia. Existen situaciones, sin embargo, en las que se necesita desarrollar un programa que tenga más de una sentencia asociada a una condición. Por ejemplo, sea un programa que lee un valor introducido por el usuario que corresponde con la medición del voltaje en una resistencia. Como se observa en la Figura 3.3, la condición del programa es que sólo se calculará la potencia disipada en la resistencia si el voltaje es mayor o igual que 100 voltios.

Cuando se necesita incluir más de una sentencia en el espacio lógico reservado para una única sentencia, se usa la **sentencia compuesta**. Como muestra el Programa 3.3, se trata simplemente de una secuencia de sentencias encerradas entre llaves `{ }`.

Programa 3.3 `#include <stdio.h>`

```

main()
{
    float voltaje;      /* Voltaje medido en voltios.      */
    float resistencia;  /* Resistencia en ohmios.      */
    float potencia;     /* Potencia calculada en vatios. */

    printf("Introduzca el voltaje medido en voltios => ");
    scanf("%f",&voltaje);
    if (voltaje >= 100.0)
    {
        printf("Voltaje es mayor o igual que 100 V\n");
        printf("Introduzca el valor de la resistencia => ");
    }
}

```

```

scanf("%f",&resistencia);
potencia = voltaje * voltaje / resistencia;
printf("La disipacion de potencia es %f vatios.\n",potencia);
}
printf("Confirmacion del valor de entrada de %f voltios.",voltaje);
exit(0);
}

```

La Figura 3.4 ilustra el esquema lógico del Programa 3.3.

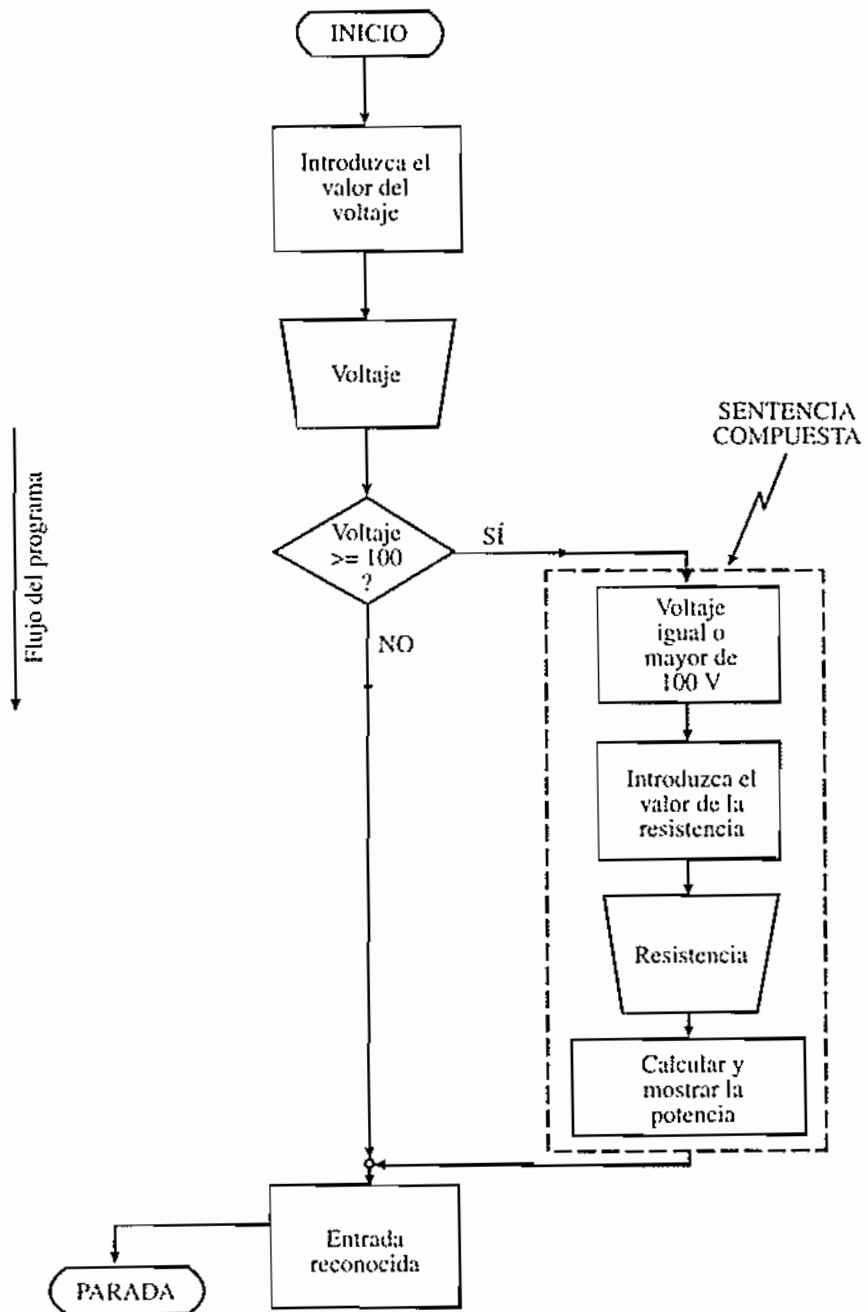


Figura 3.4. Construcción lógica del Programa 3.3.

Análisis del programa

Este programa ilustra el uso de la sentencia compuesta:

```
{
    printf("Voltaje es mayor o igual que 100 V\n");
    printf("Introduzca el valor de la resistencia => ");
    scanf("%f",&resistencia);
    potencia = voltaje * voltaje / resistencia;
    printf("La disipacion de potencia es %f vatios.\n",potencia);
}
```

Nótese que todo el fragmento está encerrado entre llaves. Lo que está dentro de estas llaves es otro bloque del programa (igual que las llaves usadas en la función `main()` o en cualquier otra función). Para la sentencia `if`, este nuevo bloque del programa es sólo otra sentencia. La Figura 3.5 ilustra este concepto.

Desde el punto de vista del programa, la operación de comparación siguiente:

```
if (voltaje >= 100.0)
```

causará la ejecución de una sola sentencia que, en este caso, es compuesta.

Así, si se ejecuta el Programa 3.3 y el usuario introduce un valor de 25 voltios para una resistencia de 500 ohmios, la salida será:

```
Introduzca el voltaje medido en voltios => 25
Confirmacion del valor de entrada de 25.000000 voltios.
```

En este caso el `if` no se ha activado ya que la comparación (relación) `voltaje >= 100.0` es FALSA.

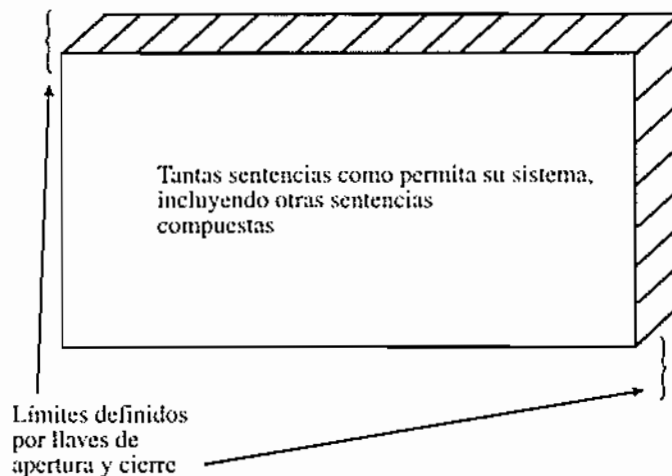


Figura 3.5. Concepto de visualización de una sentencia compuesta.

Sin embargo, si el usuario introduce un voltaje de 125 voltios para una resistencia de 500 ohmios, la salida será:

```
Introduzca el voltaje medido en voltios => 125
Voltaje es mayor o igual que 100 V
Introduzca el valor de la resistencia => 500
La disipacion de potencia es 31.250000 vatios.
Confirmacion del valor de entrada de 125.000000 voltios.
```

En esta salida puede observarse que sigue ejecutándose el último `printf`.

Uso de funciones

Puesto que la sentencia `if` puede utilizarse con una sentencia compuesta, parece lógico preguntarse si puede usarse para llamar a otra función (la cual a su vez podría contener sentencias `if` adicionales). Como puede verse en el Programa 3.4, la respuesta es afirmativa.

Programa 3.4 `#include <stdio.h>`

```
void calculo_de_potencia(float voltaje);

main()
{
    float voltaje;    /* Voltaje medido en voltios. */

    printf("Introduzca el voltaje medido en voltios => ");
    scanf("%f",&voltaje);
    if (voltaje >= 100.0)
        calculo_de_potencia(voltaje);
    printf("Confirmacion del valor de entrada de %f voltios.",voltaje);
    exit(0);
}

void calculo_de_potencia(float voltaje)
{
    float resistencia; /* Resistencia en ohmios. */
    float potencia;    /* Potencia calculada en vatios. */

    printf("Voltaje es mayor o igual que 100 V\n");
    printf("Introduzca el valor de la resistencia => ");
    scanf("%f",&resistencia);
    potencia = voltaje * voltaje / resistencia;
    printf("La disipacion de potencia es %f vatios.\n",potencia);
}
```

Análisis del programa

El Programa 3.4 ilustra una buena técnica de programación al estar dividido en dos partes. Esto no quiere decir que un programa sea mejor cuantas más partes tenga. Significa más bien que las diferentes tareas que realiza un programa deben quedar claramente distinguidas en su código. Este programa se ha dividido poniendo el cálculo opcional de la disipación de la potencia en una función separada, quedando la sentencia `if` simplemente como:

```
if (voltaje >= 100.0)
    calculo_de_potencia(voltaje);
```

Lo cual resulta mucho más legible. Esto indica que si el voltaje introducido por el usuario es mayor o igual que 100 voltios, se realizará el cálculo de la potencia utilizando para ello el valor del voltaje. Se puede observar que se pasa dicho valor como argumento de la función. El prototipo de la función que aparece al principio del programa

```
void calculo_de_potencia(float voltaje);
```

informa al compilador de que se trata de una función `void` (no devuelve ningún valor) con un argumento de tipo `float`. La cabecera de la definición de la función

```
void calculo_de_potencia(float voltaje)
```

es idéntica al prototipo (excepto por la ausencia del punto y coma final). Las declaraciones del valor de la resistencia y de la potencia están ahora incluidas sólo en la función que las usa.

```
float resistor;    /* Valor en ohmios de la resistencia */
float power;       /* Cálculo de la potencia en watios   */
```

La salida de este programa es la misma que la del 3.3. La diferencia está en que este programa se puede leer y depurar con más facilidad.

Conclusión

Esta sección ha presentado el uso de la sentencia `if` para construir bifurcaciones abiertas. Se ha mostrado que esta sentencia está compuesta de una condición y una sentencia, de tal forma que sólo se ejecutará la sentencia si la condición es VERDADERA.

Se ha presentado también la sentencia compuesta que consiste en una secuencia de sentencias encerradas entre llaves que permite incluir varias operaciones dentro de una sentencia condicional. Asimismo, se ha explicado la posibilidad de realizar llamadas a otras funciones desde una sentencia `if`.

En la próxima sección se presentará la bifurcación cerrada. Antes de pasar a ella, compruebe su comprensión de los conceptos explicados en esta sección mediante el siguiente repaso.

Repaso de la Sección 3.2

1. Describa una bifurcación cerrada.
2. ¿Cómo funciona una sentencia `if`?
3. ¿Qué es una sentencia compuesta?
4. ¿Se puede llamar a una función desde una sentencia `if`?

3.3. LA BIFURCACIÓN CERRADA

Presentación

Esta sección mostrará el concepto de **bifurcación cerrada** que extiende la capacidad de toma de decisiones del programa.

Idea básica

La Figura 3.6 ilustra la idea básica de la bifurcación cerrada mostrando dos importantes características de este tipo de bifurcaciones. En primer lugar, el flujo del programa siempre continúa hacia adelante. En segundo lugar, el programa ejecutará una de las dos alternativas e, independientemente de cuál de las dos se tome, siempre se ejecutará el resto del programa. En el lenguaje C, este tipo de bifurcación se corresponde con la **sentencia `if...else`**.

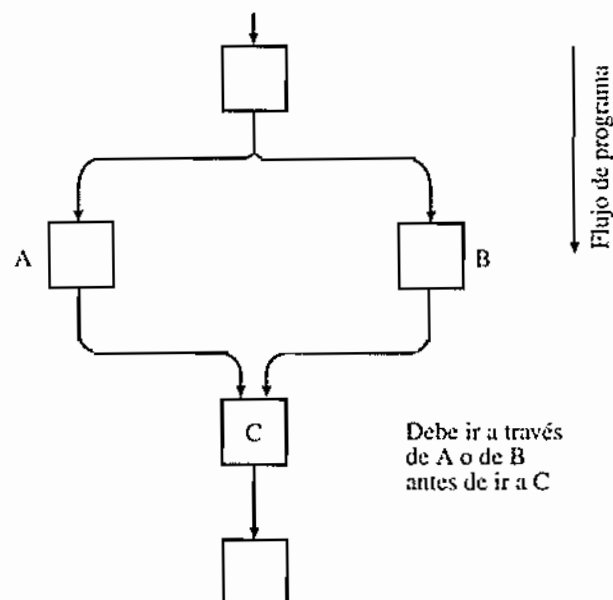


Figura 3.6. Concepto de rama cerrada.

La sentencia `if...else`

La sentencia `if...else` es una sentencia de tipo condicional que representa una bifurcación cerrada. La sintaxis de esta sentencia es la siguiente:

```
if (expresion) sentencia1 else sentencia2
```

Lo cual significa que si `expresion` es VERDADERA se ejecutará `sentencia1`, mientras que si es FALSA se ejecutará `sentencia2`. La Figura 3.7 muestra un ejemplo de bifurcación cerrada donde la condición es si un número es mayor que 5 o no.

Sentencia compuesta con `if...else`

El Programa 3.5 ilustra el uso de una sentencia compuesta dentro de una sentencia `if...else`. El programa calcula el área de un cuadrado o de un círculo dependiendo de lo que seleccione el usuario.

Programa 3.5

```
#include <stdio.h>
#define PI 3.141592

main()
{
    float eleccion; /* elección del usuario. */
    float longitud; /* longitud del lado o del radio. */
    float area;     /* área en unidades cuadradas. */

    printf("\n\nEste programa calcula el area de\n");
    printf("un cuadrado o de un circulo.\n");
    printf("\nElija un numero:\n");
    printf("1] Area de un circulo. 2] Area de un cuadrado.\n");
    printf("Su eleccion (1 o 2) => ");
    scanf("%f",&eleccion);
    if (eleccion == 1)
    {
        printf("Deme la longitud del radio del circulo => ");
        scanf("%f",&longitud);
        area = PI * longitud * longitud;
        printf("Un circulo de radio %f tiene un area de ",longitud);
        printf("%f unidades cuadradas.",area);
    }
    else
    if (eleccion == 2)
    {
        printf("Deme la longitud de un lado del cuadrado => ");
        scanf("%f",&longitud);
        area = longitud * longitud;
    }
}
```

```

        printf("Un cuadrado de longitud %f tiene un area de ",
               longitud);
        printf("%f unidades cuadradas.", area);
    }
    else
    {
        printf("Eleccion incorrecta.\n");
        printf("Debe ejecutar el programa de nuevo\n");
        printf("seleccionando un 1 o un 2.\n");
    }
    printf("\n\nAquí termina el programa que calcula\n");
    printf("el area de un circulo o un cuadrado.");
    exit(0);
}

```

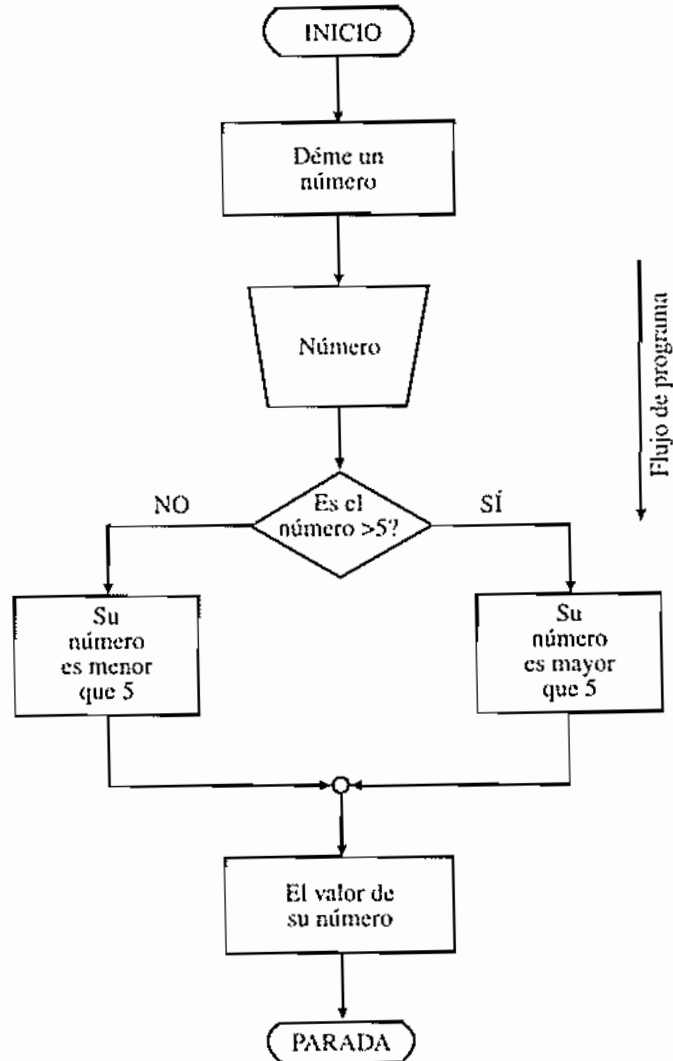


Figura 3.7. Ejemplo de bifurcación cerrada.

Análisis del programa

Como se puede observar en la Figura 3.8, este programa tiene tres opciones como queda reflejado en el uso de la construcción `if...else if...else`:

```
if (expresion1) sentencia1 else if (expresion2) sentencia2 else
sentencia3
```

Esta construcción indica que si `expresion1` es VERDADERA se ejecutará `sentencia1` y ninguna de las sentencias restantes. Si `expresion2` es VERDADERA se

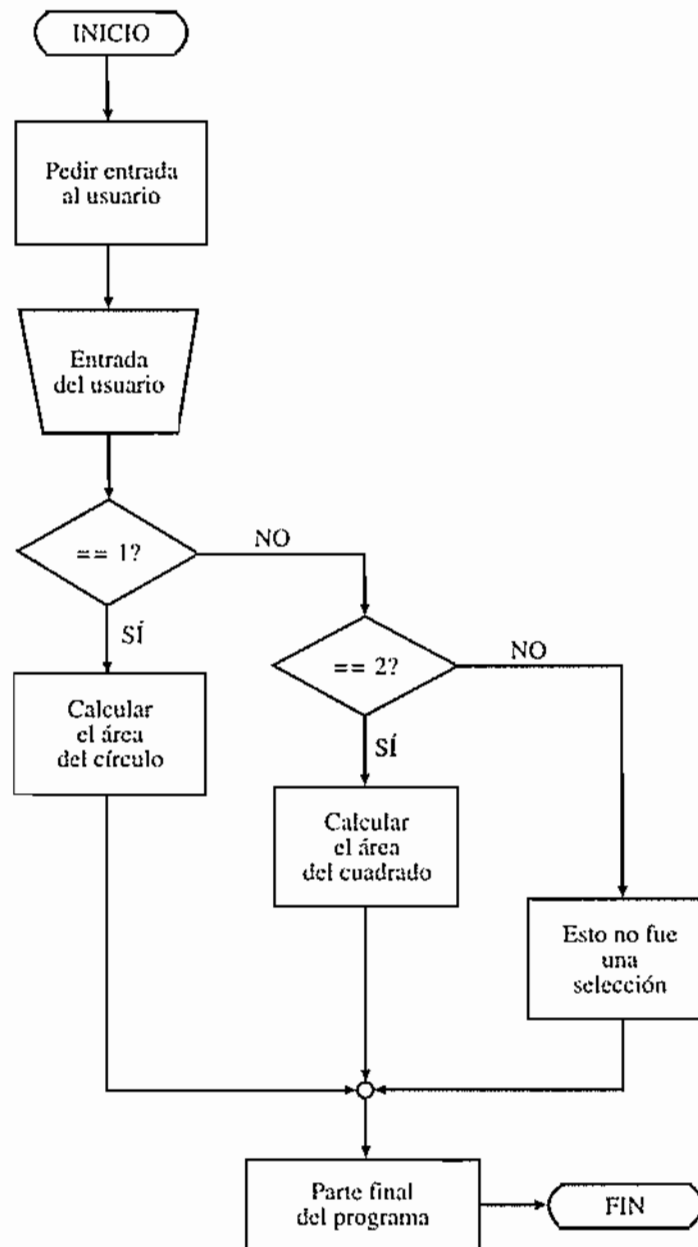


Figura 3.8. Construcción del Programa 3.5.

ejecutará sentencia_2 y ninguna más. Si ninguna de las dos expresiones es VERDADERA, se ejecutará sentencia_3 . Esto se codifica de la siguiente forma:

```
if (eleccion == 1)
{
    [Cuerpo de la sentencia compuesta para calcular el área de
    un círculo]
}
else
if (eleccion == 2)
{
    [Cuerpo de la sentencia compuesta para calcular el área de
    un cuadrado]
}
else
{
    [Cuerpo de la sentencia compuesta para imprimir un mensaje
    al usuario]
}
```

Si el usuario introduce un 1, se ejecutará la sentencia compuesta que calcula el área de un círculo. Si introduce un 2, se ejecutará la sentencia compuesta que calcula el área de un cuadrado. Para cualquier otro valor, se ejecuta la sentencia compuesta que sigue al último else.

Nótese la ausencia de un punto y coma detrás de las palabras reservadas `if` y `else`.

Uso de funciones

El Programa 3.6 resuelve el mismo problema pero de una forma más estructurada. En este programa todas las sentencias compuestas se han reemplazado por funciones y, además, las fórmulas para calcular las áreas se han definido al principio del programa mediante la directiva `#define`.

Programa 3.6

```
#include <stdio.h>
#define PI 3.141592      /* La constante pi.      */
#define cuadrado(x) x*x  /* área de un cuadrado. */
#define circulo(r) PI*r*r /* área de un círculo. */

void seleccion_del_usuario(void); /* Obtiene la elección del usuario. */
void datos_del_circulo(void);     /* Obtiene el radio del círculo y
                                calcula su área. */
void datos_del_cuadrado(void);    /* Obtiene el lado del cuadrado y
                                calcula su área. */
void seleccion_erronea(void);     /* Notifica una elección errónea */
```

```
main()
{
    printf("\n\nEste programa calcula el area de\n");
    printf("un cuadrado o de un circulo.\n");
    seleccion_del_usuario(); /* Obtiene la selección del usuario. */
    printf("\n\nAquí termina el programa que calcula\n");
    printf("el area de un circulo o un cuadrado.");
    exit(0);
}

void seleccion_del_usuario(void) /* Obtiene la elección del usuario. */
{
    float eleccion; /* elección del usuario. */

    printf("\nElija un numero:\n");
    printf("1] Area de un circulo. 2] Area de un cuadrado.\n");
    printf("Su eleccion (1 o 2) => ");
    scanf("%f",&eleccion);
    if (eleccion == 1)
        datos_del_circulo();
    else
        if (eleccion == 2)
            datos_del_cuadrado();
        else
            seleccion_erronea();
}

void datos_del_circulo(void) /* Obtiene el radio del círculo y
                             calcula su área. */
{
    float radio; /* radio del círculo. */
    float area; /* área del círculo en unidades cuadradas. */

    printf("Deme la longitud del radio del circulo => ");
    scanf("%f",&radio);
    area = circulo(radio);
    printf("Un circulo de radio %f tiene un area de ",radio);
    printf("%f unidades cuadradas.",area);
}

void datos_del_cuadrado(void) /* Obtiene el lado del cuadrado y
                              calcula su área. */
{
    float lado; /* lado del cuadrado. */
    float area; /* área del cuadrado en unidades cuadradas. */
    printf("Deme la longitud de un lado del cuadrado => ");
```

```

scanf("%f",&lado);
area = cuadrado(lado);
printf("Un cuadrado de longitud %f tiene un area de ",lado);
printf("%f unidades cuadradas.",area);
}

void seleccion_erronea(void)      /* Notifica una elección errónea */
{
    printf("Eleccion incorrecta.\n");
    printf("Debe ejecutar el programa de nuevo\n");
    printf("seleccionando un 1 o un 2.\n");
}

```

La Figura 3.9 ilustra la estructura de este programa. Como se observa en dicha figura, el programa ha sido dividido en bloques asociados a funciones.

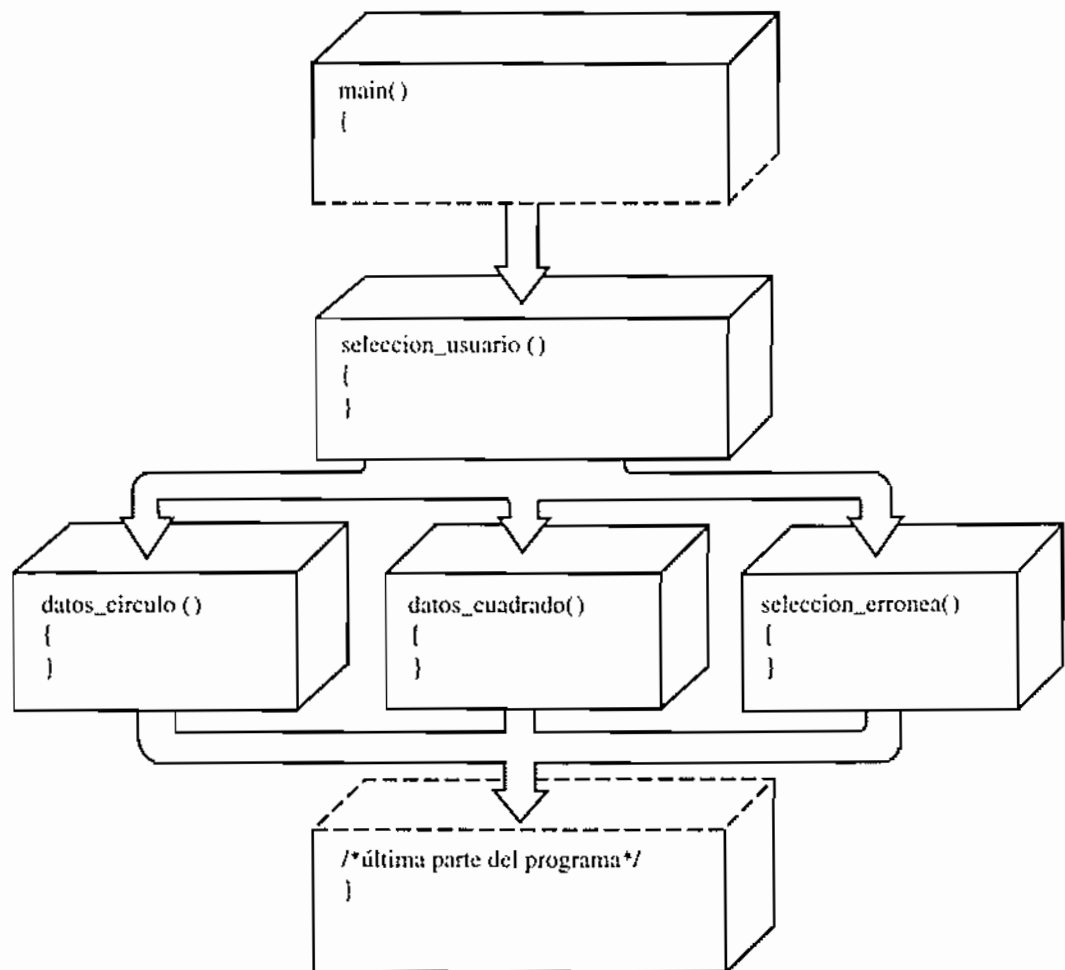


Figura 3.9. Estructura del Programa 3.7.

Análisis del programa

El programa comienza con las siguientes definiciones:

```
#define PI 3.141592      /* La constante pi.      */
#define cuadrado(x) x*x  /* área de un cuadrado. */
#define circulo(r) PI*r*r /* área de un círculo. */
```

Obsérvese que cada `#define` tiene asociado un comentario describiendo su propósito. A continuación, se presentan los prototipos:

```
void seleccion_del_usuario(void); /* Obtiene la elección del
                                   usuario. */
void datos_del_circulo(void);     /* Obtiene el radio del
                                   círculo y calcula su
                                   área. */
void datos_del_cuadrado(void);    /* Obtiene el lado del
                                   cuadrado y calcula su
                                   área. */
void seleccion_erronea(void);     /* Notifica una elección
                                   errónea */
```

Cada prototipo especifica su tipo y el tipo de sus argumentos y tiene asociado un comentario que será el mismo que aparecerá en la cabecera de la definición de la función. Seguidamente aparece la función `main()` que tiene una estructura muy simple:

```
main()
{
    printf("\n\nEste programa calcula el area de\n");
    printf("un cuadrado o de un circulo.\n");
    seleccion_del_usuario(); /* Obtiene la selección del
                               usuario. */
    printf("\n\nAquí termina el programa que calcula\n");
    printf("el area de un circulo o un cuadrado.");
    exit(0);
}
```

Con esta estructura es muy fácil tener una idea general de lo que hace el programa leyendo solamente la función `main()`. Esta función únicamente llama a `seleccion_del_usuario()`:

```
void seleccion_del_usuario(void) /* Obtiene la elección del
                                   usuario. */
{
    float eleccion; /* elección del usuario. */

    printf("\nElija un numero:\n");
    printf("1] Area de un circulo. 2] Area de un cuadrado.\n");
    printf("Su eleccion (1 o 2) => ");
```

```

scanf("%i",&eleccion);
if (eleccion == 1)
    datos_del_circulo();
else
    if (eleccion == 2)
        datos_del_cuadrado();
    else
        seleccion_erronea();
}

```

En esta función es fácil observar que primero se le pide al usuario que elija uno de los dos cálculos y, a continuación, se invoca a la función correspondiente (datos_del_circulo, datos_del_cuadrado o seleccion_erronea) que obtiene los datos requeridos, realiza los cálculos pertinentes y muestra el resultado.

Conclusión

En esta sección se ha presentado la construcción `if...else if...else`, que será utilizada con mucha frecuencia tanto en este capítulo como en posteriores. Compruebe si ha comprendido correctamente lo presentado en esta sección usando el siguiente repaso.

Repaso de la Sección 3.3

1. ¿Qué es una bifurcación cerrada?
2. ¿Qué diferencia hay entre las sentencias `if` e `if...else`?
3. ¿Se pueden usar sentencias compuestas dentro de la sentencia `if...else`?
4. ¿Se pueden realizar llamadas a funciones desde dentro de la sentencia `if...else`?
5. ¿Cómo funciona la construcción `if...else if...else`?

3.4. OPERACIONES BOOLEANAS BINARIAS (BIT A BIT)

Presentación

Esta sección presenta ejemplos que muestran el efecto de las operaciones Booleanas binarias (bit a bit). Como se podrá ver a continuación, este tipo de operaciones permiten manipular directamente los bits de los datos almacenados en la memoria del computador. Muchas aplicaciones pueden requerir el uso de este tipo de operaciones. Por ejemplo, los programas que acceden directamente a dispositivos hardware.

Manipulación de bits

Las operaciones binarias trabajan sobre los bits individuales de los datos almacenados en el computador. Por ello, es importante pensar en la representación binaria de los

Tabla 3.2. Operaciones Booleanas

Operación	Operador de Bit	Significado	Con Bits
COMPLEMENTO a uno	~	Cambia un bit a su opuesto	~1 = 0 ~0 = 1
AND binario (bit a bit)	&	El resultado es 1 si ambos bits son 1	0 & 0 = 0 0 & 1 = 0 1 & 0 = 0 1 & 1 = 1
OR binario (bit a bit)		El resultado es 0 si ambos bits son 0	0 0 = 0 0 1 = 1 1 0 = 1 1 1 = 1
OR EXCLUSIVO binario (bit a bit)	^	El resultado es 1 si ambos bits son diferentes	0 ^ 0 = 0 0 ^ 1 = 1 1 ^ 0 = 1 1 ^ 1 = 0

números para entender mejor este tipo de operaciones. Recuerde que un 1 se corresponde con el valor Booleano VERDADERO y un 0 con el valor FALSO. La Tabla 3.2 muestra el comportamiento de varias operaciones Booleanas.

El Programa 3.7 muestra el uso de las diversas operaciones Booleanas binarias. En los siguientes apartados se presentan ejemplos de las diversas operaciones analizando también el fragmento del programa donde se lleva a cabo dicha operación.

Programa 3.7 #include <stdio.h>

```
main()
{
    int valor1;
    int valor2;
    int desplazamiento_izq;
    printf("Introduzca un numero hexadecimal del 00 al FF =>");
    scanf("%X",&valor1);
    printf("El complemento de %X es => %X\n",valor1, ~valor1);
    printf("Introduzca otro numero hexadecimal del 00 al FF =>");
    scanf("%X",&valor2);
    printf("El AND binario de %X y %X produce => ", valor1, valor2);
    printf("%X\n", valor1 & valor2);
    printf("El OR binario de %X y %X produce => ", valor1, valor2);
    printf("%X\n", valor1 | valor2);
    printf("El OR-Exclusivo binario de %X y %X produce => ",
           valor1, valor2);
    printf("%X\n", valor1 ^ valor2);
    printf("Introduzca el numero de posiciones que se desplazara a ");
```

```

printf("la izquierda el primer numero => ");
scanf("%X",&desplazamiento_izq);
printf("Desplazar %X %d posiciones a la izquierda produce => "
      valor1,desplazamiento_izq);
printf("%X\n",valor1 << desplazamiento_izq);
}

```

Complemento binario (bit a bit)

El **complemento binario** (o complemento a 1) de un número se calcula cambiando la representación binaria del número los unos por ceros y viceversa. En el Ejemplo 3.2 se muestran algunos casos.

Ejemplo 3.2 Calcular el complemento binario de los siguientes números hexadecimales:

A. 00 B. FF C. A3

Solución

A.	Convertir a binario:	00000000
	Calcular complemento a 1:	11111111
	Convertir a hexadecimal:	FF
B.	Convertir a binario:	11111111
	Calcular complemento a 1:	00000000
	Convertir a hexadecimal:	00
C.	Convertir a binario:	10100011
	Calcular complemento a 1:	01011100
	Convertir a hexadecimal:	5C

Suponiendo que el usuario introduce el valor A3, el fragmento del Programa 3 correspondiente al complemento binario produciría la siguiente salida:

```

Introduzca un numero hexadecimal del 00 al FF => A3
El complemento de A3 es => 5C

```

AND binario (bit a bit)

El **AND binario** entre dos números equivale a aplicar la función AND sobre las correspondientes parejas de bits de las respectivas representaciones binarias de los números. En el Ejemplo 3.3 se muestran algunos casos.

Ejemplo 3.3 Calcular el AND binario de los siguientes números hexadecimales:

A. 00 & FF B. FF & A5 C. D3 & 8E

Solución

A.	Convertir a binario:	00000000
		11111111
	Calcular AND:	00000000
	Convertir a hexadecimal:	00
B.	Convertir a binario:	11111111
		10100101
	Calcular AND:	10100101
	Convertir a hexadecimal:	A5
C.	Convertir a binario:	11010011
		10001110
	Calcular AND:	10000010
	Convertir a hexadecimal:	82

Suponiendo que el usuario introduce los valores D3 y 8E, el fragmento del Programa 3.7 correspondiente al AND binario produciría la siguiente salida:

```
Introduzca un numero hexadecimal del 00 al FF => D3
Introduzca otro numero hexadecimal del 00 al FF => 8E
El AND binario de D3 y 8E produce => 82
```

OR binario (bit a bit)

El **OR binario** entre dos números equivale a aplicar la función OR sobre las correspondientes parejas de bits de las respectivas representaciones binarias de los números. En el Ejemplo 3.4 se muestran algunos casos.

Ejemplo 3.4 Calcular el OR binario de los siguientes números hexadecimales:

A. 00 | FF B. FF | A5 C. D3 | 8E

Solución

A.	Convertir a binario:	00000000
		11111111
	Calcular OR:	11111111
	Convertir a hexadecimal:	FF
B.	Convertir a binario:	11111111
		10100101

Calcular OR:	11111111
Convertir a hexadecimal:	FF
C. Convertir a binario:	11010011
	10001110
Calcular OR:	11011111
Convertir a hexadecimal:	DF

Suponiendo que el usuario introduce los valores D3 y 8E, el fragmento del Programa 3.7 correspondiente al OR binario produciría la siguiente salida:

```
Introduzca un numero hexadecimal del 00 al FF => D3
Introduzca otro numero hexadecimal del 00 al FF => 8E
El OR binario de D3 y 8E produce => DF
```

XOR binario (bit a bit)

El **XOR binario** entre dos números equivale a aplicar la función XOR sobre las correspondientes parejas de bits de las respectivas representaciones binarias de los números. En el Ejemplo 3.5 se muestran algunos casos.

Ejemplo 3.5 Calcular el XOR binario de los siguientes números hexadecimales:

A. $00 \wedge FF$ B. $FF \wedge A5$ C. $D3 \wedge 8E$

Solución

A. Convertir a binario:	00000000
	11111111
Calcular XOR:	11111111
Convertir a hexadecimal:	FF
B. Convertir a binario:	11111111
	10100101
Calcular XOR:	01011010
Convertir a hexadecimal:	5A
C. Convertir a binario:	11010011
	10001110
Calcular XOR:	01011101
Convertir a hexadecimal:	5D

Suponiendo que el usuario introduce los valores D3 y 8E, el fragmento del Programa 3.7 correspondiente al XOR binario produciría la siguiente salida:

```
Introduzca un numero hexadecimal del 00 al FF => D3
Introduzca otro numero hexadecimal del 00 al FF => 8E
El XOR binario de D3 y 8E produce => 5D
```

Desplazamiento de bits

C permite realizar desplazamientos de los bits de un número tanto hacia la izquierda (operador <<) como hacia a la derecha (operador >>). En el Ejemplo 3.6 se muestran algunos casos.

Ejemplo 3.6

Calcular el desplazamiento del número de bits especificado de los siguientes números hexadecimales:

A. FF << 2 B. 5C >> 3 C. 05 << 4

Solución

A. Convertir a binario:	11111111
Calcular desplazamiento:	11111100
Convertir a hexadecimal:	FC
B. Convertir a binario:	01011100
Calcular desplazamiento:	00001011
Convertir a hexadecimal:	0B
C. Convertir a binario:	00000101
Calcular desplazamiento:	01010000
Convertir a hexadecimal:	50

Suponiendo que el usuario introduce el valor 5C y un desplazamiento a la izquierda de 3 bits, el fragmento del Programa 3.7 correspondiente al desplazamiento produciría la siguiente salida:

```
Introduzca un numero hexadecimal del 00 al FF => 5C
Introduzca el numero de posiciones que se desplazara a la
izquierda el primer numero => 3
Desplazar 5C 3 posiciones a la izquierda produce => 2E0
```

Conclusión

En esta sección se ha presentado el concepto de operaciones binarias, tanto de tipo Booleano como de desplazamiento. Compruebe su comprensión de esta sección usando el siguiente repaso.

Repaso de la Sección 3.4

1. ¿En qué consiste el complemento binario?
2. ¿En qué consiste el AND binario?
3. ¿En qué consiste el OR binario?
4. ¿En qué consiste el XOR binario?
5. ¿En qué consiste el desplazamiento de bits?

3.5. OPERACIONES LÓGICAS

Presentación

En esta sección se podrá aprender cómo se usan los operadores lógicos de C. En ella se aplicarán los conocimientos presentados en la primera parte del capítulo. El uso de los operadores lógicos aumentará la capacidad de toma de decisiones de los programas.

AND lógico

La operación **AND lógico** se expresa en C de la siguiente forma.

```
(expresión1) && (expresión2)
```

La operación anterior se evaluará como VERDADERA sólo si expresión₁ es VERDADERA y expresión₂ es VERDADERA; en cualquier otro caso se evaluará como FALSA. No se debe perder de vista que en C un valor FALSO es realmente un 0, mientras que VERDADERO se corresponde con un valor distinto de cero. La Tabla 3.3 resume el comportamiento de esta operación.

Nótese que se usan los símbolos && para representar esta operación, no permitiéndose que aparezcan espacios en blanco entre estos símbolos, aunque sí a la derecha o a la izquierda.

El Programa 3.8 ilustra el uso de la operación AND lógico.

Programa 3.8

```
#include <stdio.h>
main()
{
    float resultado;    /* Resultado de una expresión lógica. */
    resultado = 0 && 0;
    printf("0 && 0 = %f\n", resultado);
    resultado = 0 && 1;
    printf("0 && 1 = %f\n", resultado);
    resultado = 1 && 0;
    printf("1 && 0 = %f\n", resultado);
    resultado = 1 && 1;
    printf("1 && 1 = %f\n", resultado);
}
```

La Figura 3.10 muestra la estructura de este programa.

Tabla 3.3. La operación AND

expresión ₁	expresión ₂	Resultado
FALSO	FALSO	FALSO
FALSO	VERDADERO	FALSO
VERDADERO	FALSO	FALSO
VERDADERO	VERDADERO	VERDADERO

Análisis del programa

La ejecución del Programa 3.8 produce:

```
0 && 0 = 0.000000  
0 && 1 = 0.000000  
1 && 0 = 0.000000  
1 && 1 = 1.000000
```

Nótese que la variable `resultado` es de tipo `float`. Se recomienda que el lector pruebe con otros tipos de datos. Esta variable se usa para almacenar el resultado de cada operación AND.

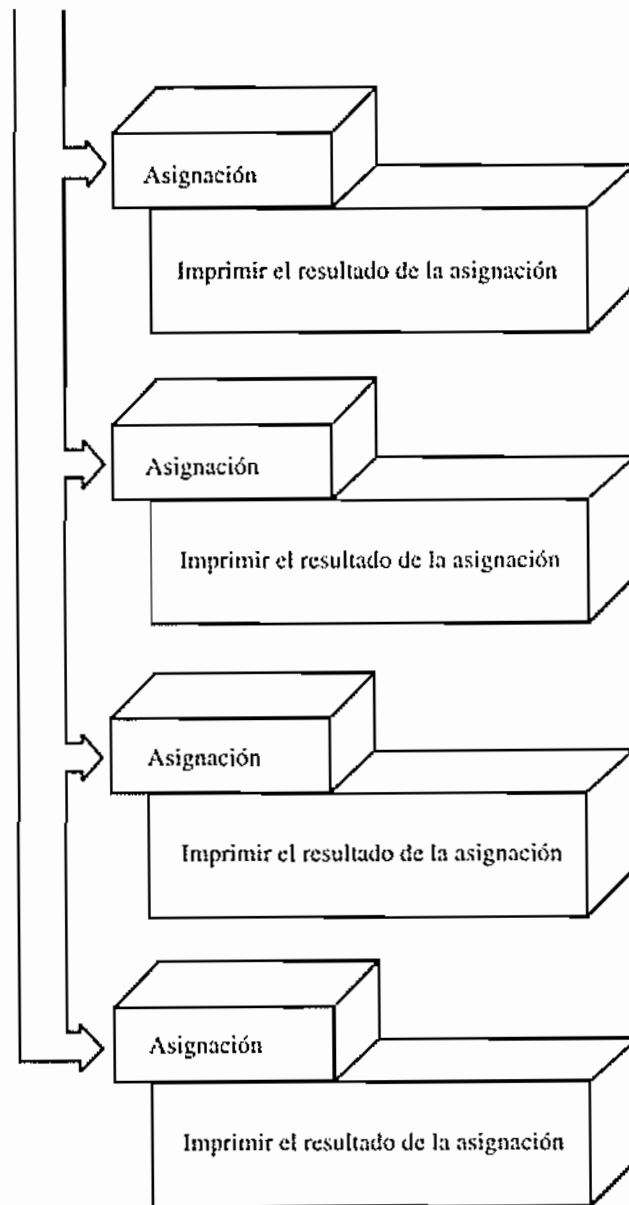


Figura 3.10. Operaciones del Programa 3.13.

La primera operación ($0 \ \&\& \ 0$) realiza el AND lógico de dos valores FALSO (representados en C como ceros) produciendo como resultado un valor FALSO que corresponde con un 0 en la variable `resultado`.

Las dos siguientes operaciones ($0 \ \&\& \ 1$) y ($1 \ \&\& \ 0$) se evalúan también como FALSO almacenándose, por tanto, un 0 en `resultado`.

La última ($1 \ \&\& \ 1$) es la única que produce un valor VERDADERO que corresponde con un 1.

La operación OR lógico

La operación **OR lógico** se expresa en C de la siguiente forma:

```
(expresión1) || (expresión2)
```

La operación anterior se evaluará como FALSA sólo si `expresión1` es FALSA y `expresión2` es FALSA; en cualquier otro caso se evaluará como VERDADERA. Recuerde de nuevo que en C un resultado FALSO es realmente un 0 mientras que un resultado VERDADERO se corresponde realmente con un valor distinto de 0. La Tabla 3.4 resume el comportamiento de esta operación.

El Programa 3.9 ilustra el uso de la operación OR lógico.

Programa 3.9

```
#include <stdio.h>

main()
{
    float resultado; /* Resultado de una expresión lógica. */
    resultado = 0 || 0;
    printf("0 || 0 = %f\n", resultado);
    resultado = 0 || 1;
    printf("0 || 1 = %f\n", resultado);
    resultado = 1 || 0;
    printf("1 || 0 = %f\n", resultado);
    resultado = 1 || 1;
    printf("1 || 1 = %f\n", resultado);
}
```

Tabla 3.4. La operación OR

expresión ₁	expresión ₂	Resultado
FALSO	FALSO	FALSO
FALSO	VERDADERO	VERDADERO
VERDADERO	FALSO	VERDADERO
VERDADERO	VERDADERO	VERDADERO

Análisis del programa

La ejecución del Programa 3.9 produce:

```
0 || 0 = 0.000000
0 || 1 = 1.000000
1 || 0 = 1.000000
1 || 1 = 1.000000
```

Nótese que la variable resultado es de tipo `float`. Se recomienda que el lector pruebe con otros tipos de datos. Esta variable se usa para almacenar el resultado de cada operación OR.

La primera operación (`0 || 0`) realiza el OR lógico de dos valores FALSOS (representados en C como ceros) produciendo como resultado un valor FALSO que corresponde con un 0 en la variable resultado.

Las tres siguientes operaciones (`0 || 1`), (`1 || 0`) y (`1 || 1`) se evalúan como VERDADERAS almacenándose, por tanto, un 1 en resultado.

Operaciones lógicas y de comparación (relacionales)

Los operadores de comparación relacionales se pueden usar conjuntamente con los operadores lógicos, ya que devuelven valores VERDADERO y FALSO que son los mismos usados por los operadores lógicos. La Tabla 3.5 muestra el orden de prioridad de los operadores presentados hasta ahora. Este orden indica que el operador `!` se evalúa antes que el `*`, que a su vez se evalúa antes que el `<`, y así sucesivamente.

En la tabla se muestra un nuevo operador denominado **NOT lógico** que se representa mediante el símbolo `!`. Obsérvese que la tabla muestra en qué orden se ejecutarán las operaciones contenidas en una línea de un programa cuando ésta incluya más de una. El siguiente ejemplo muestra el uso de esta tabla y el de la operación NOT lógico.

Tabla 3.5. Precedencia en C

Operadores	Nombre
<code>!</code>	Negación (NOT) lógica
<code>*</code> <code>/</code>	Multiplicación y división
<code>+</code> <code>-</code>	Suma y resta
<code><</code> <code><=</code> <code>>=</code> <code>></code>	Menor, menor o igual, mayor o igual, mayor
<code>==</code> <code>!=</code>	Igual, distinto
<code>&&</code>	Conjunción (AND) lógica
<code> </code>	Disyunción (OR) lógica

Ejemplo 3.7 Calcular el resultado de las siguientes expresiones:

- A. $(5 == 5) \parallel (6 == 7)$
- B. $(5 == 8) \parallel (6 != 7)$
- C. $(8 >= 5) \parallel (!(5 <= 2))$

Solución

- A. Se debe analizar primero el valor lógico de la operación contenida en los paréntesis internos. En este caso, $(5 == 5)$, es VERDADERO por lo que no es necesario evaluar resultado de la siguiente operación, puesto que el resultado final de una operación OR es siempre VERDADERO si al menos uno de los miembros de la expresión OR es VERDADERO. El compilador de C usa este mismo método de análisis: sólo evaluará la segunda parte de un OR si la primera es FALSA.
- B. Usando el mismo método se determina primero que la expresión $(5 == 8)$ es FALSA. tratarse de una operación AND, no será necesario evaluar $(6 != 7)$ (que sería VERDADERO), puesto que una operación AND con algún miembro FALSO devolverá siempre FALSO. El compilador utiliza el mismo método también en este caso: sólo se evaluará segunda parte de un AND si la primera es VERDADERA.
- C. Esta expresión usa el operador NOT lógico. La evaluación de la primera parte $(8 >= 5)$ es VERDADERA. Puesto que se trata de una operación AND, será necesario evaluar segunda parte $!(5 <= 2)$. Al ser ambas partes de la expresión VERDADERAS, el resultado final será también VERDADERO.

El Programa 3.10 muestra un ejemplo del uso combinado de operaciones de comparación y lógicas. El programa lee un número y comprueba si su valor está entre 1 y 10. Si es así, se imprimirá un mensaje.

Programa 3.10 #include <stdio.h>

```
main()
{
    float numero;    /* Valor proporcionado por el usuario. */
    printf("\n\nDeme un numero del 1 al 100 => ");
    scanf("%f",&numero);
    if ((numero >= 1.0) && (numero <= 10.0))
        printf("El numero leído esta entre 1 y 10.");
}
```

Análisis del programa

Observe la línea del programa que combina una expresión lógica y una de comparación:

```
if ((numero >= 1.0) && (numero <= 10.0))
```

Esta sentencia puede interpretarse como muestra la Figura 3.11.

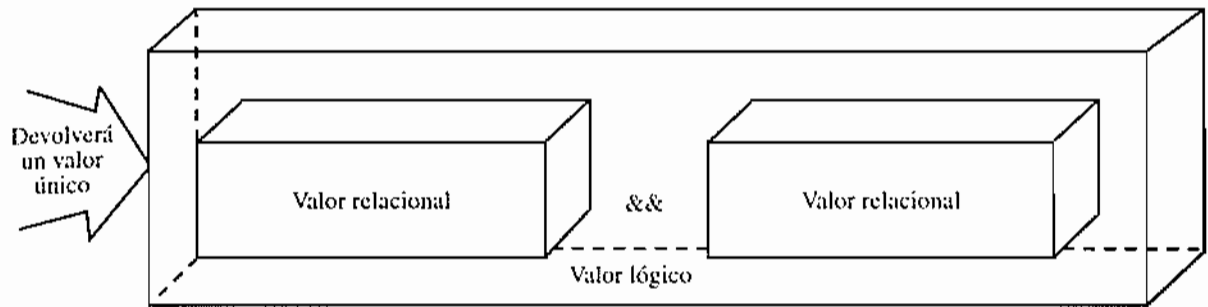


Figura 3.11. Operación relacional y lógica.

Los paréntesis alrededor de las operaciones de comparación podrían omitirse ya que el símbolo $\geq$ y el $\leq$ tienen mayor prioridad que el $\&\&$. Sin embargo, es recomendable utilizarlos ya que se mejora la claridad del programa.

Combinación de operaciones lógicas

Se puede usar cualquier combinación de operaciones de comparación y lógicas. Por ejemplo, el Programa 3.11 usa una expresión lógica compleja que combina operaciones AND y OR con operaciones de comparación. Dado un número del 1 al 100, esta sentencia comprueba si el número está incluido en el 10 % superior (del 90 % al 100 %) o en el inferior (del 1 % al 10 %).

Programa 3.11 `#include <stdio.h>`

```
main()
{
    float numero;      /* Valor proporcionado por el usuario. */

    printf("\n\nDeme un numero del 1 al 100 => ");
    scanf("%f",&numero);
    if (((numero >= 1.0) && (numero <= 10.0)) ||
        ((numero >= 90.0) && (numero <= 100.0)))
        printf("El numero esta en 10%% inferior o superior de 100.");
}
```

Programa de aplicación

El Programa 3.12 muestra una aplicación que usa operaciones de comparación y lógicas. Este programa lee la temperatura de un proceso introducida por el usuario, aunque lo podría haber obtenido a través de un sensor de temperatura conectado al computador, y produce un resultado que depende de la misma. La Figura 3.12 muestra la estructura del programa.

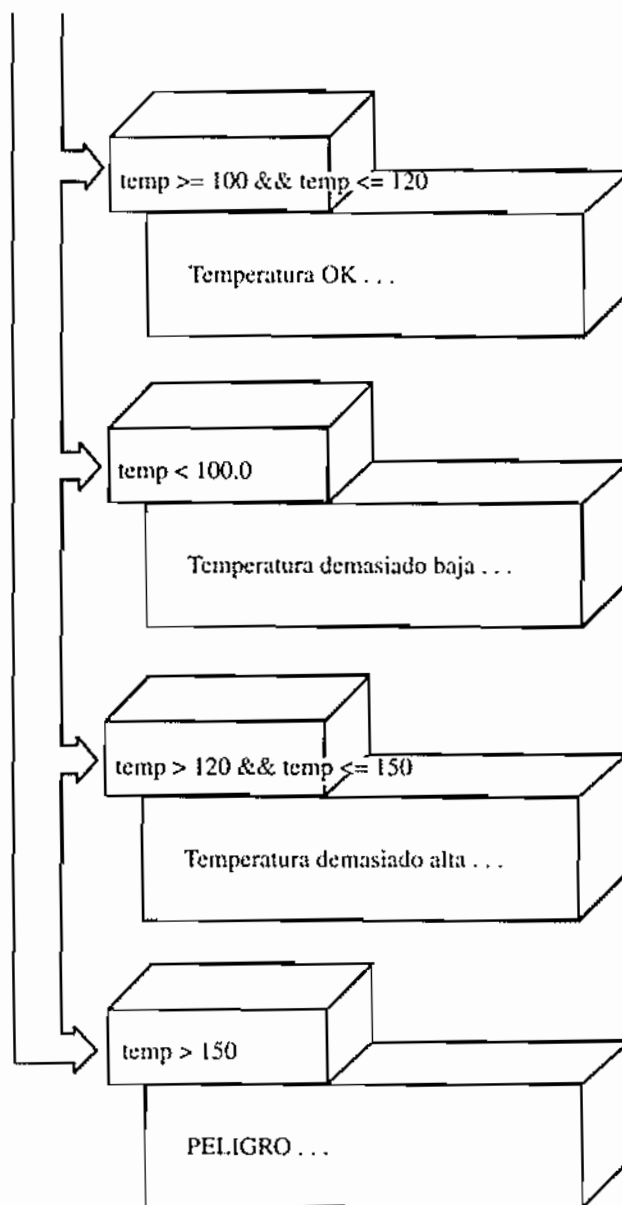


Figura 3.12. Programa para probar temperatura.

Programa 3.12 `#include <stdio.h>`

```

main()
{
    float temp;    /* Valor de la temperatura. */
    printf("\n\nDeme el valor de la temperatura => ");
    scanf("%f",&temp);
    if ((temp >= 100.0) && (temp <= 120.0))
        printf("Temperatura correcta, continúe el proceso.");
    if (temp < 100.0)

```

```

    printf("Temperatura demasiado baja, aumente la energia.");
    if ((temp > 120.0) && (temp <= 150.0))
        printf("Temperatura demasiado alta, disminuya la energia.");
    if (temp > 150.0)
        printf("Peligro! Desconecte los sistemas.");
}

```

Conclusión

En esta sección se ha demostrado la potencia que proporciona el uso combinado de operadores de comparación y lógicos. Se han presentado ejemplos muy simples, así como aplicaciones más complejas. En la próxima sección se explicará otro mecanismo que ayuda a la toma de decisiones. Antes compruebe su comprensión de los conceptos presentados en esta sección mediante el siguiente repaso.

Repaso de la Sección 3.5

1. Describir cómo opera el AND lógico ¿Con qué símbolo se representa?
2. Describir cómo opera el OR lógico ¿Con qué símbolo se representa?
3. Describir cómo opera el NOT lógico.
4. Ponga un ejemplo del uso combinado de una operación de comparación y una lógica.
5. Explicar cómo se evalúan las operaciones AND y OR.

3.6. CONVERSIÓN DE TIPOS

En esta sección se presenta información sobre lo que hasta ahora ha podido parecer un simple «detalle» pero que, como se verá a continuación, puede llegar a ser muy importante en programas más complejos.

Tipos de datos

El lector recordará que todo dato en C tiene un tipo. Puede ser `int`, `char` o cualquier otro tipo. El lenguaje C permite mezclar tipos de datos. Por ejemplo, se puede sumar un valor de tipo `int` a uno de tipo `float`.

Cuando se mezclan tipos en una expresión, el compilador convertirá todas las variables a un único tipo compatible y a continuación se llevará a cabo la operación. Esta conversión se realizará teniendo en cuenta el **rango** de cada tipo. Las variables de un tipo de rango inferior se convierten al tipo de las variables con mayor rango. El rango de los tipos es el siguiente:

Rango inferior $\leq$ `char`, `int`, `long`, `float`, `double` $\Rightarrow$ Rango superior

Así, si se produce una operación con un valor de tipo `char` y otro `int`, el resultado será de tipo `int`. Si se trata de un tipo `float` y uno `int`, el resultado será `float`. Es recomendable, sin embargo, no mezclar tipos y, en el caso de que sea necesario, usa el mecanismo de conversión de tipos.

Conversión de tipos

Este mecanismo permite convertir una variable a un determinado tipo. Para expresar esta operación, se debe anteponer el nombre del tipo de datos deseado al nombre de la variable. Por ejemplo, si `valor` estaba declarado como tipo `int`, se convertirá a tipo `float` de la siguiente forma:

```
resultado = (float)valor;
```

La operación de conversión debería usarse cuidadosamente cuando se pasa de un rango superior a uno inferior, ya que podría producirse una pérdida de datos al utilizar un tipo de datos con menos capacidad de representación.

No se pueden realizar conversiones del tipo `void` a cualquier otro tipo, pero sí de cualquier otro tipo a `void`.

Valor-i

El término **valor-i** (*Ivalue*) suele aparecer cuando se habla de lenguajes de programación y significa literalmente valor de la parte izquierda, refiriéndose a que la operación de asignación asigna el valor del operando de la parte derecha a la dirección de memoria indicada por el operando de la parte izquierda (valor-i). Por ejemplo, la sentencia `valor = 3 + 5;` es legal en C. Sin embargo, la sentencia `3 + 5 = valor;` no lo es, puesto que la expresión que aparece en el lado izquierdo no es un valor-i (no indica una dirección de memoria). Por lo tanto, el operando del lado izquierdo de una asignación debe ser una expresión cuyo resultado corresponda con una posición de memoria modificable, esto es, una expresión valor-i.

Conclusión

En esta sección se han presentado algunos detalles importantes del lenguaje. Se ha aprendido como trabaja el lenguaje cuando se mezclan tipos de datos diferentes. Se ha visto asimismo cómo convertir tipos de datos y se ha explicado el concepto de valor-i. Compruebe su comprensión de esta sección usando el siguiente repaso.

Repaso de la Sección 3.6

1. Explique qué significa mezclar tipos de datos.
2. ¿Qué sucede cuando se suma un tipo `int` a un tipo `float`?
3. Especifique cómo trabaja C cuando se mezclan tipos de datos diferentes.

4. ¿Cómo se expresa una conversión de tipo en C?
5. ¿Qué es una expresión valor-i?

3.7. LA SENTENCIA SWITCH

Presentación

En esta sección se presenta un mecanismo que permite realizar una selección entre varias posibilidades. En primer lugar, se justificará la necesidad del mismo para pasar a continuación a definirlo y aplicarlo a un problema.

Selección de una alternativa entre varias

El Programa 3.13 es un ejemplo de selección de una entre varias alternativas. Este programa muestra una de las tres formas de la ley de Ohm dependiendo de cuál elija el usuario tecleando la letra correspondiente.

Programa 3.13 #include <stdio.h>

```
main()
{
    char eleccion; /* Elección del usuario. */

    printf("\n\nElija la forma de la ley de Ohm que corresponda:\n");
    printf("A) Voltaje B) Intensidad C) Resistencia\n");
    printf("Su eleccion (A, B, o C) => ");
    scanf("%c",&eleccion);
    if (eleccion == 'A')
        printf("V = I * R");
    else
        if (eleccion == 'B')
            printf("I = V / R");
        else
            if (eleccion == 'C')
                printf("R = V / I");
            else
                printf("Elección incorrecta.\n");
}
```

Análisis del programa

La Figura 3.13 ilustra la lógica del Programa 3.13.

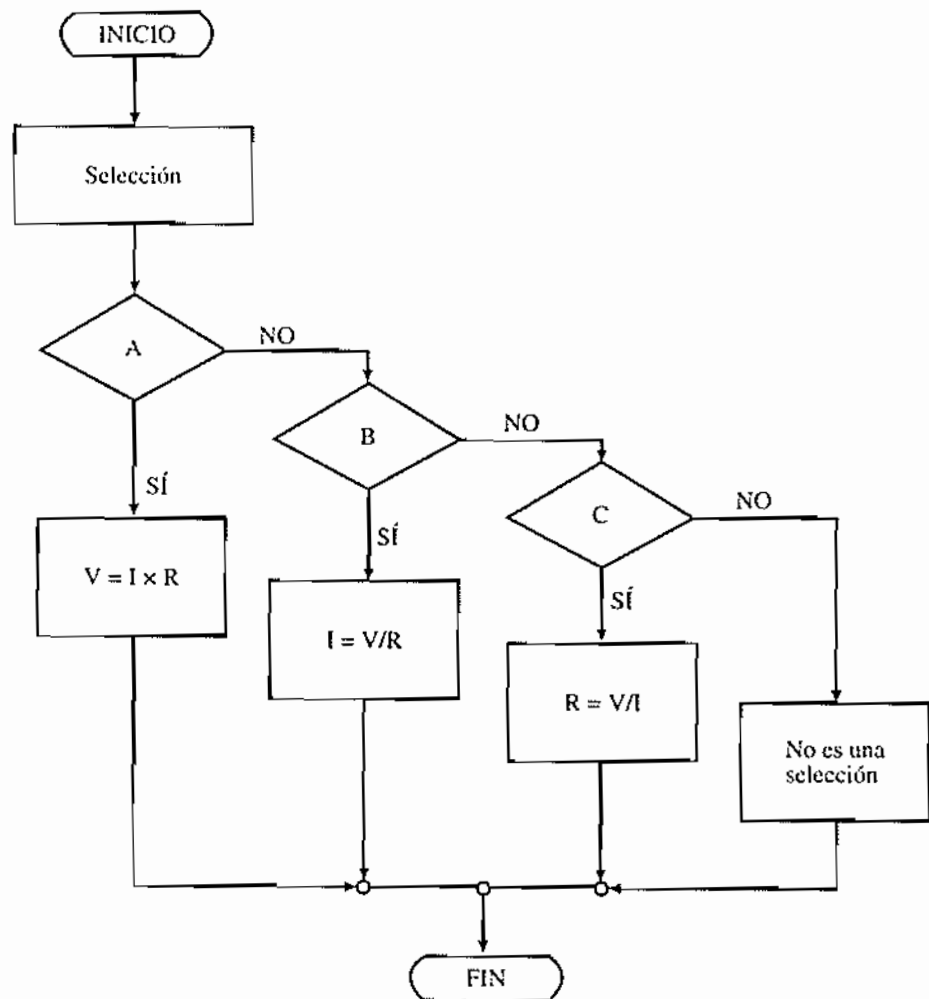


Figura 3.13. Lógica del Programa 3.13.

Como se puede ver en la figura, el programa proporciona al usuario tres alternativas mediante el uso de sentencias `if...else`. Observe el uso de la variable de tipo `char` en la selección. El usuario debe introducir una de las tres letras mayúsculas siguientes: A, B, o C.

La selección se realizará usando sentencias de comparación como la siguiente:

```

if (eleccion == 'A')
    printf("V = I * R");

```

Si la selección es igual al carácter A, se ejecutará la función `printf()`. Si el usuario no selecciona una opción correcta, se ejecutará el `printf()` asociado al último `else`:

```

else
    printf("Eleccion incorrecta.\n");

```

Este tipo de programa, en el que se hace una elección entre varias posibilidades, es tan común que existe una sentencia especial en C para este caso.

La sentencia switch

La sentencia switch es una forma más fácil de codificar múltiples sentencias if...else. Esta sentencia tiene la siguiente sintaxis:

```
switch (expresion)
{
    case expresion-constante: (expresión)
    default: (expresión)
}
```

Donde:

- switch = Palabra reservada que indica el comienzo de esta sentencia.
- expresion = Cualquier expresión legal en C.
 - { Define el comienzo del cuerpo de la sentencia switch.
- case = Palabra reservada que precede a la expresión constante que corresponde a una alternativa.
- expresion-constante = Especifica qué debe cumplirse para que se active la alternativa.
- default = Palabra reservada que indica la opción que se llevará a cabo cuando no se cumpla ninguna alternativa.
 - } Define el final del cuerpo del switch.

La sentencia switch requiere dos nuevas palabras reservadas: case y break. La sentencia switch especifica una expresión cuyo valor determina qué alternativa case se activará. El término break indica cuando termina de ejecutarse el código asociado a una alternativa. El Programa 3.14 hace lo mismo que el Programa 3.13 pero usando la sentencia switch en un lugar de varias sentencias if...else.

Programa 3.14 #include <stdio.h>

```
main()
{
    char eleccion; /* Elección del usuario. */
    printf("\n\nElija la forma de la ley de Ohm que corresponda:\n");
    printf("A] Voltaje B] Intensidad C] Resistencia\n");
    printf("Su eleccion (A, B, o C) => ");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A': printf("V = I * R");
                  break;
        case 'B': printf("I = V / R");
                  break;
        case 'C': printf("R = V / I");
                  break;
    }
```

```

        default : printf("Eleccion incorrecta.\n");
    } /* Fin del switch. */
}

```

Análisis del programa

Este programa muestra que es más fácil leer el código del programa cuando se usa la sentencia `switch` en vez de múltiples sentencias `if...else`.

En el programa se observa que la variable elección es de tipo `char` y que, por tanto, las etiquetas asociadas a cada `case` deben ser del mismo tipo ('A', 'B' y 'C'). Se puede ver también cómo están sangradas hacia la izquierda la llave de apertura y la de cierre para que se identifique claramente donde comienza y termina la sentencia `switch`. Otro aspecto importante es el comentario que sigue a la llave de cierre que permite recordar que se trata del fin de una sentencia `switch`. Es una buena costumbre incluir un comentario, sobre todo cuando existe una lista de alternativas muy larga. Obsérvese también cómo está sangrada la sentencia `break` para identificar claramente donde termina la ejecución de cada alternativa. Como antes, si no se seleccionó ninguna alternativa válida, el programa ejecutará la sentencia asociada a `default`.

```
default : printf("Eleccion incorrecta.\n");
```

Se pueden usar también sentencias compuestas dentro de un `switch`.

Sentencias compuestas en un switch

Como muestra el Programa 3.15, se pueden usar sentencias compuestas dentro de un `switch`. Este programa permite seleccionar al usuario una de las formas de la ley de Ohm y realiza los cálculos correspondientes a la misma.

Programa 3.15

```

#include <stdio.h>

main()
{
    char eleccion;           /* Elección del usuario. */
    float voltaje;           /* Voltaje del circuito en voltios. */
    float intensidad;        /* Intensidad del circuito en amperios. */
    float resistencia;       /* Resistencia del circuito en ohmios. */

    printf("\n\nElija la forma de la ley de Ohm que corresponda:\n");
    printf("A) Voltaje B) Intensidad C) Resistencia\n");
    printf("Su eleccion (A, B, o C) -> ");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A':{ /* Cálculo del voltaje. */
                    printf("Introduzca la intensidad en amperios => ");

```



```

scanf("%f",&intensidad);
printf("Valor de la resistencia en ohmios => ");
scanf("%f",&resistencia);
voltaje = intensidad * resistencia;
printf("El voltaje es %f voltios.",voltaje);
}
break;
case 'B':{/* Cálculo de la intensidad. */
printf("Introduzca el voltaje en voltios => ");
scanf("%f",&voltaje);
printf("Valor de la resistencia en ohmios => ");
scanf("%f",&resistencia);
intensidad = voltaje / resistencia;
printf("La intensidad es %f amperios.", intensidad);
}
break;
case 'C':{/* Cálculo de la resistencia. */
printf("Introduzca el voltaje en voltios => ");
scanf("%f",&voltaje);
printf("Valor de la intensidad en amperios => ");
scanf("%f",&intensidad);
resistencia = voltaje / intensidad;
printf("La resistencia es %f ohmios.", resistencia);
}
break;
default: printf("Eleccion incorrecta.\n");
printf("Ejecute el programa seleccionando A,B,o C");
} /* Fin del switch. */
}

```

La Figura 3.14 muestra la estructura general del programa.

Obsérvese la estructura de bloque usada en el `switch`. Esta estructura facilita la lectura y comprensión del código del programa. Nótese cómo está sangrada la sentencia compuesta y cómo quedan claramente definidos su comienzo y fin mediante las llaves. Fíjese también en la posición del `break` que identifica donde termina el cuerpo de cada opción. Aunque al usuario del programa le es indiferente la estructura del mismo, es muy importante para cualquiera que desee comprender o modificar el programa que su estructura sea lo más clara posible.

Análisis del programa

La característica más importante del programa es el uso de sentencias compuestas dentro de un `switch`.

```

switch(eleccion)
{
    case 'A':{/* Cálculo del voltaje. */
        printf("Introduzca la intensidad en amperios => ");

```

```

scanf("%f",&intensidad);
printf("Valor de la resistencia en ohmios => ");
scanf("%f",&resistencia);
voltaje = intensidad * resistencia;
printf("El voltaje es %f voltios.",voltaje);
}
break;

```

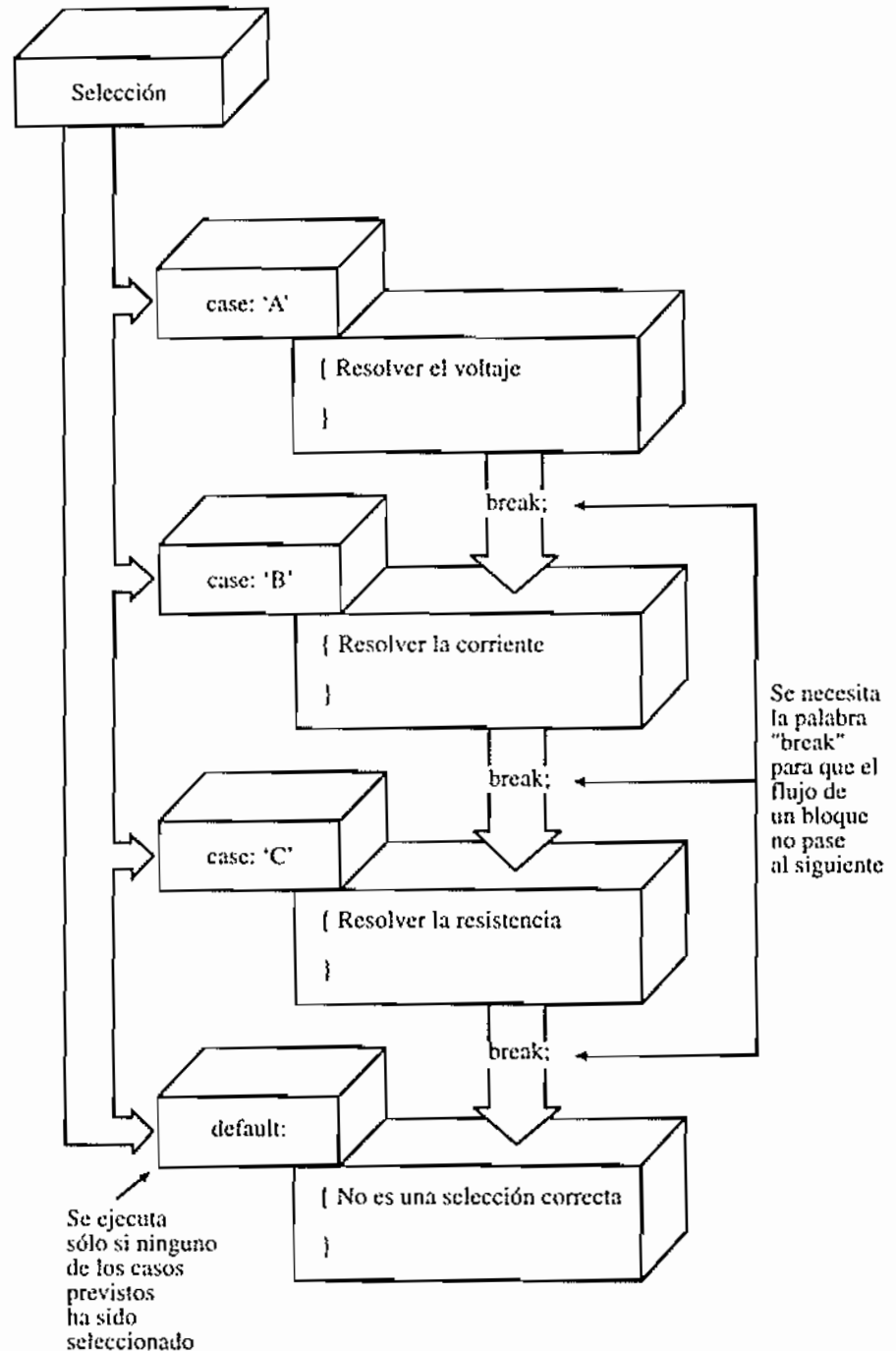


Figura 3.14. Estructura general del Programa 3.15.

Si se selecciona esta opción (`eleccion='A'`), se ejecutará toda la secuencia de sentencias incluidas en la sentencia compuesta.

Como el lector ya habrá imaginado, se pueden incluir llamadas a funciones dentro de un `switch`.

Sentencia `switch` con funciones

Además de poderse incluir sentencias compuestas dentro de un `switch`, se pueden también realizar llamadas a funciones desde el mismo. Se deja como ejercicio para el lector modificar el programa anterior para que en lugar de usar sentencias compuestas dentro de las opciones del `switch`, se definan funciones para cada una de las opciones y se realicen llamadas a las mismas desde dentro del `switch`. Con esta estructura mejorará considerablemente la legibilidad del programa.

Expresión de selección

La expresión que determina qué alternativa del `switch` se activa puede ser de tipo `int` o `char`, pero no de tipo `float`.

Sentencias `switch` anidadas

Se pueden incluir múltiples `switch` dentro de otras sentencias `switch`. La Figura 3.15 ilustra esta posibilidad.

Conclusión

En esta sección se ha presentado la sentencia `switch` mostrándose su estructura y las diferentes formas de usarla. Se ha visto también que se pueden usar sentencias compuestas dentro de un `switch`, así como realizar llamadas a funciones.

En la próxima sección se presentará la última sentencia que permite realizar toma de decisiones. Antes de pasar a ella, compruebe su comprensión de esta sección mediante el siguiente repaso.

Repaso de la Sección 3.7

1. ¿Para qué se usa la sentencia `switch`?
2. ¿Qué palabras reservadas, además del propio `switch`, se usan con esta sentencia?
3. Explique el propósito de la palabra reservada `default` dentro de una sentencia `switch`.
4. ¿Se pueden hacer llamadas a funciones dentro de una sentencia `switch`?

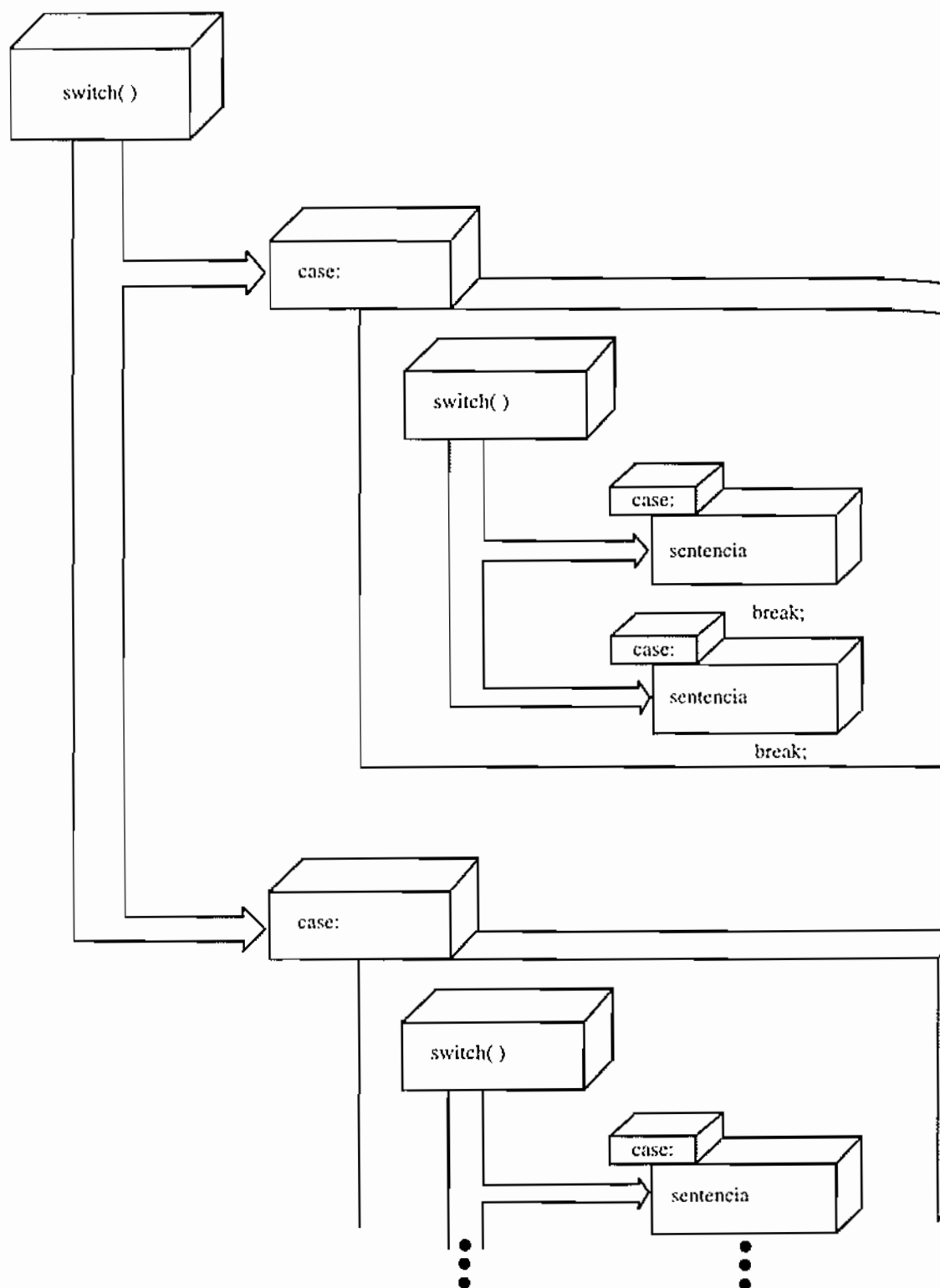


Figura 3.15. Visión conceptual de alternativas dentro de un switch.

3.8. MÁS SOBRE LA SENTENCIA `switch` Y EL OPERADOR CONDICIONAL

Presentación

Esta sección presenta una nueva aplicación de la sentencia `switch` con lo que se terminará la exposición de los mecanismos que proporciona el lenguaje C para la toma de decisiones.

Otro uso de la sentencia `switch`

Recuerde de la última sección que la sintaxis de la sentencia `switch` es la siguiente:

```
switch (expresion)
{
    case etiqueta:
        sentencia;
        break;
}
```

Como se podrá ver a continuación en el Programa 3.16, la sentencia `break` es importante para determinar la lógica del `switch`. El programa muestra las fórmulas requeridas para calcular la potencia suministrada por una fuente de potencia en un circuito serie con tres resistencias. El usuario sólo debe especificar qué parámetros del circuito se conocen. Nótese que se ha omitido el `break` del `switch`.

Programa 3.16 `#include <stdio.h>`

```
main()
{
    char eleccion;    /* Elección del usuario. */

    printf("Este programa muestra las formulas necesarias para\n");
    printf("calcular la energia que proporciona una fuente de \n");
    printf("voltaje a un circuito serie formado por tres \n");
    printf("resistencias conociendo el voltaje de la fuente\n");
    printf("\n\nElija usando la letra correspondiente:\n");
    printf("A] Se conoce el valor de las resistencias.\n");
    printf("B] Se conoce el valor total de la resistencia\n");
    printf("C] Se conoce el valor total de la intensidad\n");
    printf("Su eleccion => ");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A': printf("Rt = R1 + R2 + R3\n");
```

```

        case 'B': printf("It = Vt / Rt\n");
        case 'C': printf("Pt = Vt * It\n");
                    break;
        default : printf("Eleccion incorrecta!\n");
    } /* Fin del switch. */
}

```

Análisis del programa

A continuación se muestran las diferentes respuestas del programa dependiendo de la elección del usuario.

1. El usuario introduce la letra A:

```

Su eleccion => A
Rt = R1 + R2 + R3
It = Vt / Rt
Pt = Vt * It

```

2. El usuario introduce la letra B:

```

Su eleccion => B
It = Vt / Rt
Pt = Vt * It

```

3. El usuario introduce la letra C:

```

Su eleccion => C
Pt = Vt * It

```

4. El usuario introduce la letra T:

```

Eleccion incorrecta!

```

Lo que ha sucedido se debe a la omisión de la sentencia `break`.

```

switch(eleccion)
{
    case 'A': printf("Rt = R1 + R2 + R3\n");
    case 'B': printf("It = Vt / Rt\n");
    case 'C': printf("Pt = Vt * It\n");
                break;
    default : printf("Eleccion incorrecta!\n");
} /* Fin del switch. */

```

Obsérvese que si el usuario introduce una A, se ejecutarán todas las sentencias hasta el primer `break`. Si no existiera este `break`, se ejecutaría también la sentencia asociada a `default`. Como muestra la Figura 3.16, cuando se activa una opción, se ejecutarán todas las sentencias que aparecen desde ese punto hasta el primer `break` hasta el final del `switch`.

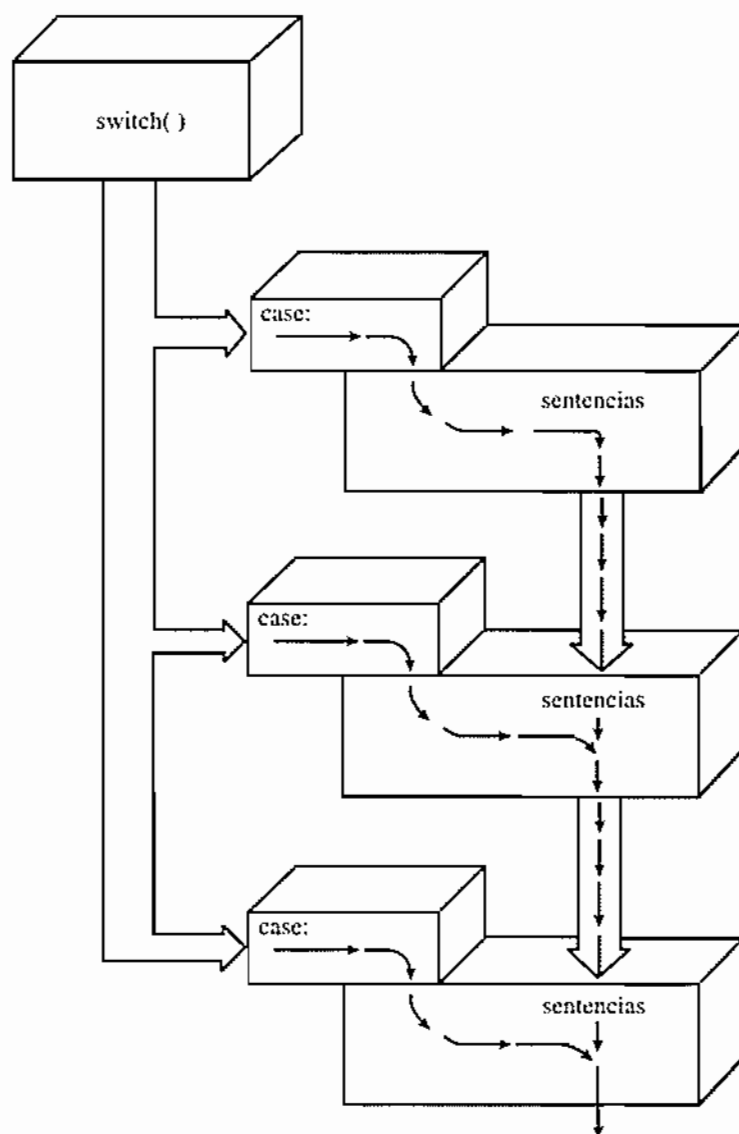


Figura 3.16. Alternativas sin ruptura.

El operador condicional

El último mecanismo disponible en C para que un programa pueda tomar decisiones es el operador condicional cuya sintaxis es la siguiente:

`expresion1 ? expresion2 : expresion3`

Lo cual implica que si `expresion1` es VERDADERA (cualquier valor distinto de cero), el operador condicional devuelve `expresion2`. En caso contrario, devuelve `expresion3`. El Programa 3.17 ilustra el uso de este nuevo operador.

Programa 3.17 #include <stdio.h>

```

main()
{
    int eleccion;    /* Elección del usuario. */

    printf("Introduzca un 1 o un 0 => ");
    scanf("%d",&eleccion);
    eleccion? printf("Un uno.\n") : printf("Un cero.\n");
}

```

Análisis del programa

En este programa, si el usuario introduce un 0, se ejecutará el segundo printf() que mostrará lo siguiente:

Un cero.

Si el usuario introduce un 1 (o cualquier valor distinto de 0), se ejecutará el primer printf() que mostrará lo siguiente:

Un uno.

Lo que ha ocurrido es que el valor de elección ha determinado cuál de las dos siguientes expresiones se ejecutará.

Una aplicación del operador condicional

Para la aplicación del operador condicional, se toma como base el circuito eléctrico mostrado en la Figura 3.17.

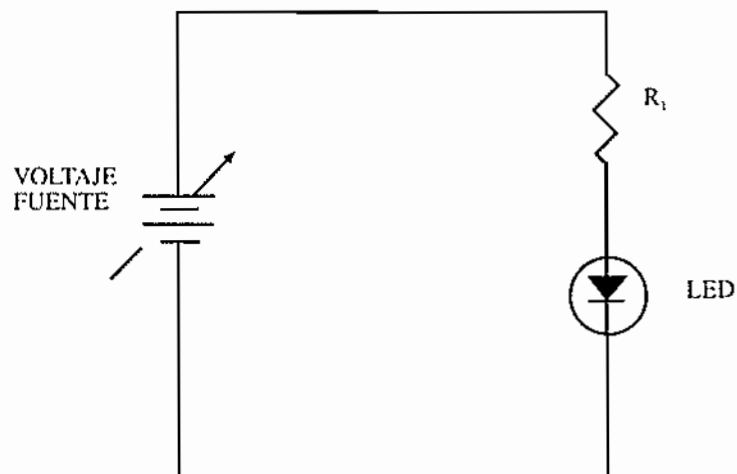


Figura 3.17. Circuito aplicación para un operador condicional.

En este circuito, el dispositivo denominado LED (*ligh emitting diode*, diodo emisor de luz) no conducirá intensidad hasta que el voltaje que se le aplica sea mayor que 2,3 voltios. Cuando esto sucede, los incrementos posteriores del voltaje de la fuente no implicarán cambios en el voltaje del LED, produciéndose la caída del voltaje restante en la resistencia. Para calcular la intensidad de la corriente en el circuito, se divide la caída de voltaje a través de la resistencia entre el valor de la resistencia. Como muestra el Programa 3.18, este cálculo puede realizarse fácilmente usando el operador condicional.

Programa 3.18

```
#include <stdio.h>

main()
{
    float voltaje_led;          /* Voltaje en el LED en voltios. */
    float voltaje_resistencia; /* Voltaje en resistencia en voltios. */
    float voltaje_fuente;      /* Voltaje de la fuente en voltios. */
    float intensidad_circuito; /* Intensidad en el LED en amperios. */
    float resistencia;         /* Valor de la resistencia en ohmios. */

    printf("\n\nIntroduzca el voltaje de la fuente en voltios => ");
    scanf("%f",&voltaje_fuente);
    printf("Introduzca el valor de la resistencia en ohmios => ");
    scanf("%f",&resistencia);
    voltaje_led = (voltaje_fuente < 2.3)? voltaje_fuente : 2.3;
    voltaje_resistencia = voltaje_fuente - voltaje_led;
    intensidad_circuito = voltaje_resistencia / resistencia;
    printf("La intensidad total del circuito es %f amperios.",
           intensidad_circuito);
}
```

Análisis del programa

Suponiendo que el usuario del programa introduce un valor de 2 voltios, la siguiente expresión:

```
(voltaje_fuente < 2.3)
```

será VERDADERA. Por tanto, se asignará `voltaje_fuente` a la variable `voltaje_led`.

A continuación, se evalúa la siguiente expresión:

```
voltaje_resistencia = voltaje_fuente - voltaje_led;
```

que dará un valor de 0 para el voltaje de la resistencia. Seguidamente se calcula la expresión:

```
intensidad_circuito = voltaje_resistencia / resistencia;
```

que volverá a dar 0. Éste es el resultado correcto ya que si el voltaje en el LED es menor de 2.3 voltios, no circulará corriente por el circuito.

Suponiendo ahora que el usuario introduce un valor mayor que 2.3 voltios, por ejemplo 5 voltios, para el voltaje de la fuente, la expresión siguiente:

```
(voltaje_fuente < 2.3)
```

será FALSA. Así, se asignará a la variable `voltaje_led` un valor de 2.3 voltios (la expresión a la derecha del símbolo `:`)

```
(voltaje_fuente < 2.3)? voltaje_fuente : 2.3
```

lo que da un valor de 2.7 voltios para el voltaje de la resistencia. El valor de la intensidad quedará determinado por el valor de la resistencia que introduzca el usuario.

Conclusión

Esta sección ha mostrado cómo usar la sentencia `switch` variando la utilización del `break`. Se ha presentado también el operador condicional y un ejemplo de su uso. Compruebe que ha asimilado correctamente los conceptos de esta sección con el siguiente repaso.

Repaso de la Sección 3.8

1. Explique el efecto de omitir el `break` en un `switch`.
2. ¿Qué se necesita hacer para asegurar que `default` siempre se ejecuta?
3. Explique cómo funciona el operador condicional.
4. ¿Qué valores hacen VERDADERO al operador condicional? ¿Qué valores lo hacen FALSO?

3.9. DEPURACIÓN E IMPLEMENTACIÓN DE PROGRAMAS

Presentación

En esta sección se muestra un método para seleccionar qué partes de un programa serán compiladas y qué partes no. El nombre de este método es **compilación condicional** y se usa a menudo en programas grandes que necesitan considerar muchas condiciones diferentes.

Directiva condicional

La Tabla 3.6 muestra algunas de las directivas de compilación disponibles en C.

Tabla 3.6. Instrucciones de compilación condicional

Instrucción	Significado
#ifdef	El código que sigue a esta instrucción se compila en ciertas circunstancias pero no en otras.
#endif	Termina la compilación seleccionada desde el #ifdef anterior.
#else	Presenta una opción de compilación.

El Programa 3.19 muestra el uso de estas directivas para realizar labores de depuración de un programa.

```

Programa 3.19 #include <stdio.h>

#define DEPURACION      /* Depuración activada. */

main()
{
#ifdef DEPURACION      /* Si depuración activada, hacer lo siguiente: */
    printf("Seccion de depuracion del programa.\n");
    printf("Este mensaje aparecera solo si se define DEPURACION.\n");
#endif DEPURACION      /* Fin de la sección de depuración. */

    printf("Parte principal del programa.\n");
    printf("No depende de si esta definido DEPURACION o no\n");

#ifdef DEPURACION
    printf("Aquí termina la depuracion.\n"); /* Si depuración
                                              activada. */
#else DEPURACION
    printf("No se ha usado depuracion.\n"); /* Si depuración
                                              desactivada. */
#endif DEPURACION
}

```

El programa producirá la siguiente salida:

```

Seccion de depuracion del programa.
Este mensaje aparecera solo si se define DEPURACION.
Parte principal del programa.
No depende de si esta definido DEPURACION o no
Aquí termina la depuracion.

```

Para lograr que el compilador ignore la parte de depuración, sólo se necesita eliminar la definición de DEPURACION, lo que se puede hacer poniendo dicha definición entre comentarios:

```

/* #define DEPURACION */ /* Depuración activada. */

```

Cuando se hace esto, el programa producirá la siguiente salida:

```
Parte principal del programa.
No depende de si esta definido DEPURACION o no
No se ha usado depuracion.
```

Conclusión

En esta sección se ha mostrado el uso de las directivas de compilación condicional que permiten seleccionar qué partes del programa se compilarán. Se trata de una característica que generalmente se usa en programas grandes. Compruebe su comprensión de esta sección mediante el siguiente repaso.

Repaso de la Sección 3.9

1. ¿En qué consiste la compilación condicional?
2. ¿Qué es una directiva de compilación?
3. Explique el propósito de la directiva `#ifdef`.
4. ¿Dónde se usan normalmente las directivas de compilación condicional?

3.10. PROGRAMA DE APLICACIÓN: REPARACIÓN DE UN ROBOT

Presentación

En esta sección se muestra una aplicación dentro del campo de la tecnología de la toma de decisiones dirigida por un programa. Los diagramas de reparación son actualmente bastante populares. Se trata de diagramas de flujo que dirigen al técnico durante la reparación de un determinado sistema. Esta sección muestra cómo desarrollar un programa C que replique el comportamiento de un diagrama de este tipo para el caso de la reparación de un robot. Éste es un primer paso hacia la reparación dirigida por computador donde éste se encarga de realizar las medidas reales (a través de los circuitos de interfaz correspondientes) y luego diagnostica el problema real.

El robot hipotético

Para mantener el programa de aplicación dentro de un tamaño razonable, sólo se considerará una parte del diagrama de flujo, la reflejada en la Figura 3.18.

El propósito es crear un programa C que replique el comportamiento de este diagrama. Por motivos de simplicidad, será el usuario del programa el encargado de introducir los valores requeridos por el programa.

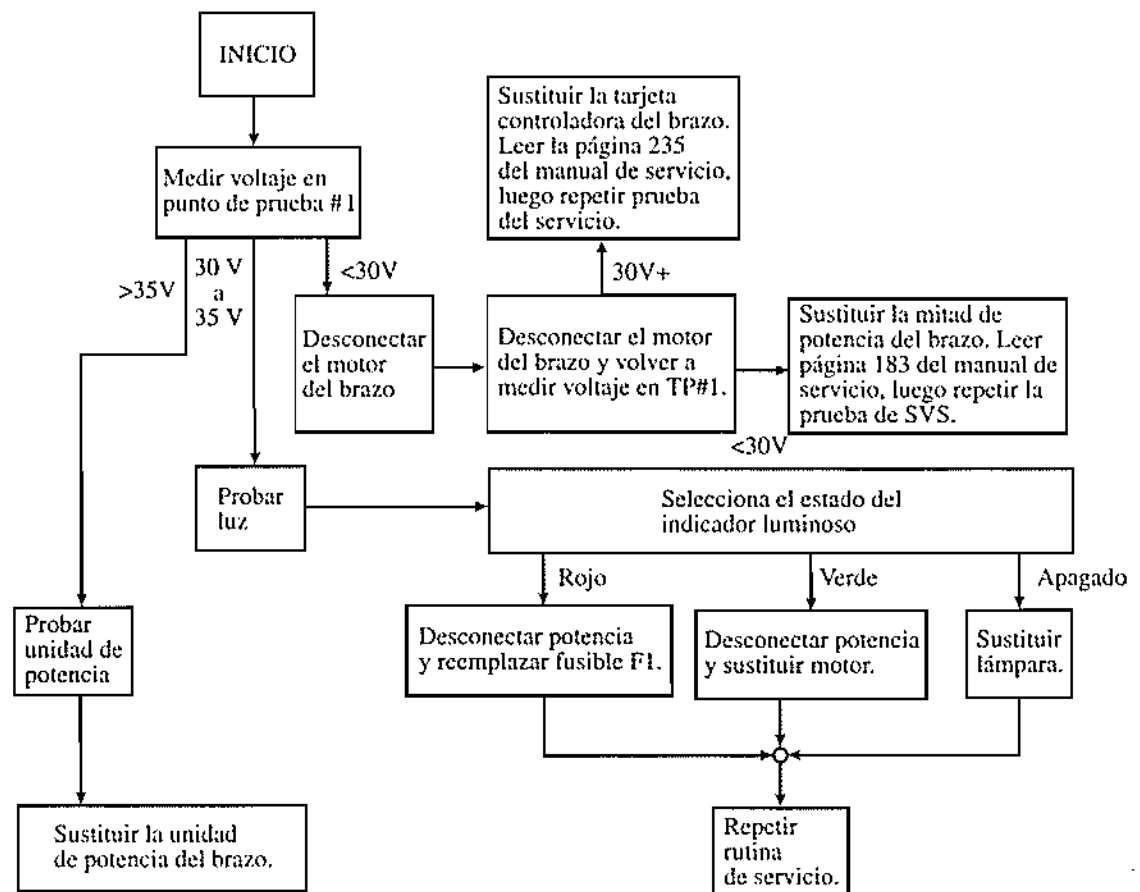


Figura 3.18. Diagrama de flujo para un robot hipotético.

Primer paso

El primer paso siempre es escribir lo siguiente:

- Propósito del programa
- Entrada requerida (fuente)
- Proceso sobre la entrada
- Salida requerida (destino)

Para el caso del programa que nos ocupa:

- Propósito del programa: Replicar la lógica de reparación de la Figura 3.18 para que pueda interactuar con el usuario.
- Entrada requerida (fuente): Los valores numéricos que representan los valores del voltaje medidos en los puntos de prueba. Las restantes entradas requieren una respuesta que seleccione entre múltiples opciones.
- Proceso sobre la entrada: Los valores de entrada introducidos por el usuario determinan qué instrucciones se le indicarán al usuario o la finalización del programa.
- Salida requerida (destino): Las instrucciones que le indican al usuario qué información debe suministrar al programa.

Esto deja claro qué debe hacer (y no hacer) el programa. El próximo paso es desarrollar el algoritmo.

Desarrollo del algoritmo

El algoritmo para este programa ya está básicamente hecho, puesto que el diagrama de reparación es un algoritmo gráfico. Lo único que hay que hacer es seguir la lógica del diagrama. El resto del programa será el prólogo estándar que incluye la información del programa, su explicación y la descripción de sus funciones. Esta información se desarrolla en la fase inicial.

Fase inicial de desarrollo

En esta fase, el programador debe construir una primera versión que incluya el nombre del programa, el nombre de su autor y el propósito del programa. El único prototipo que aparecerá en el programa será el correspondiente a una función que explique al usuario el propósito del programa mostrándolo por la pantalla cuando se ejecute el programa. De esta forma, la documentación del programa queda incluida en el mismo, evitándose tener que consultar otros documentos.

Fases intermedias de desarrollo

Durante estas etapas, el programador irá construyendo de forma incremental las distintas partes del programa. Deberá identificar qué funciones formarán parte del programa y especificar sus prototipos. Asimismo, se escribirá la estructura de la función `main()` con las llamadas a las funciones correspondientes e irá desarrollando de forma progresiva cada una de éstas. En un determinado instante algunas funciones estarán totalmente especificadas mientras que otras estarán sin desarrollar. A estas últimas se les denomina suplentes del programa.

Programa final

El Programa 3.20 presenta el desarrollo final del programa de reparación del robot hipotético. Nótese que en él se han utilizado la mayoría de las técnicas de toma de decisiones presentadas en este capítulo.

Programa 3.20

```
#include <stdio.h>

/*
Programa: Reparación de un Robot
Desarrollado por: Robert Shooter
```

Descripción: Este programa muestra al usuario el proceso de análisis y reparación de un robot hipotético. El programa también muestra las diferentes facilidades disponibles en el lenguaje C para la toma de decisiones.

Variables:

medida = Medida en voltios tomada por el usuario.
estado_luz = Valor del estado de la luz introducido por el usuario.

Constantes: ninguna

Prototipos de funciones:

```
*/  
  
void explicar_programa(void);  
/*  
    Esta función explica al usuario el funcionamiento del programa.  
*/  
  
float brazo(void);  
/*  
    Esta función contiene la rutina de servicio del brazo del robot.  
*/  
  
void unidad_de_potencia(void);  
/*  
    Esta función es la rutina de servicio de la unidad de potencia.  
*/  
  
void comprobar_luz(void);  
/*  
    Esta función muestra las instrucciones para comprobar el estado del  
    indicador luminoso.  
*/  
  
void desconexion_dispositivo_brazo(void);  
/*  
    Esta función muestra las instrucciones para comprobar la unidad de  
    desconexión del dispositivo del brazo.  
*/  
  
main()  
{  
    float medida;          /* Medida introducida por el usuario. */  
  
    explicar_programa(); /* Explicar el programa al usuario. */  
    medida = brazo();     /* Valor de la medida del brazo. */
```

```

    if (medida > 35) unidad_de_potencia();
    else
    if ((medida <= 35) && (medida >= 30)) comprobar_luz();
    else
    if (medida < 30) desconexion_dispositivo_brazo();

    exit(0);
}

void explicar_programa(void)    /* Explicar el programa al usuario. */
{
    printf("\n\nEste programa representa la reparacion\n");
    printf("de un hipotetico robot.\n");
    printf("\nLa informacion que usted introduzca simulara las \n");
    printf("medidas reales que se tomarian directamente del robot.\n");
    printf("\nEl programa le ira pidiendo las medidas que\n");
    printf("debe ir introduciendo.\n");
}

float brazo(void)    /* Realiza la rutina de servicio del brazo. */
{
    float medida;    /* Medida introducida por el usuario. */
    printf("\nMida el voltaje en el punto de comprobacion #1\n");
    printf("Su medida en voltios => ");
    scanf("%f",&medida);
    return(medida);
}

void unidad_de_potencia()    /* Instrucciones de servicio de la unidad de
                             potencia. */
{
    printf("\nConsulte en el manual la prueba de la unidad de\n");
    printf("potencia y reemplace la unidad de potencia del brazo.");
}

void comprobar_luz()    /* Comprobación de indicador de estado. */
{
    int estado_luz;    /* Estado de la luz introducido por el usuario. */

    printf("\nIntroduzca en que condicion esta la luz de estado:\n");
    printf("1] Rojo 2] Verde 3] Apagado \n");
    printf("Introduzca el numero => ");
    scanf("%d",&estado_luz);
    switch(estado_luz)
    {
        case 1 : printf("Desconecte la alimentacion y reemplace\n");
                  printf("el fusible F1.\n");
                  break;
    }
}

```



```

        case 2 : printf("Desconecte la alimentacion y reemplace\n");
                  printf("la placa del dispositivo del brazo.\n");
                  break;
        case 3 : printf("Reemplace la bombilla.\n");
                  break;
        default : printf("Seleccion incorrecta!\n");
    }
    printf("\nRepita la rutina de servicio.\n");
}

void desconexion_dispositivo-brazo() /* Instrucciones del dispositivo
                                     del brazo. */
{
    float medida; /* Medida hecha por el usuario. */

    printf("Desconecte el dispositivo del brazo y vuelva \n");
    printf("a medir el voltaje en PP#1.\n");
    printf("Su medida en voltios => ");
    scanf("%f",&medida);

    medida = (medida > 30)? 30 : medida;

    if (medida < 30)
    {
        printf("Reemplace la unidad de potencia del brazo. \n");
        printf("Consultar la página 183 de manual, luego repita\n");
        printf("la prueba SVS.");
    }
    else
    if (medida == 30)
    {
        printf("Reemplace la placa del circuito del dispositivo\n");
        printf("del brazo. Consulte la pagina 235 del manual de\n");
        printf("servicio, luego repita la prueba.");
    }
}

```

Análisis del programa

Obsérvese la estructura de `main()` que básicamente refleja la del diagrama de flujo de reparación. En dicho diagrama, la primera medida es el voltaje en el punto de prueba 1. A partir de esta medida, el programa puede bifurcar a una dirección de las tres posibles. Después de llamar a la función `explicar_programa()`, se invoca a la función `brazo()` cuyo valor devuelto, que es de tipo `float`, se almacena en la

variable `medida`. El resto del programa queda condicionado por este valor que se usa dentro de la siguiente construcción `if...else if...else`:

```
if (medida > 35) unidad_de_potencia();
else
if ((medida <= 35) && (medida >= 30)) comprobar_luz();
else
if (medida < 30) desconexion_dispositivo_brazo();
```

De esta forma, la lectura del código de la función `main()` proporciona una visión general de la estructura del programa.

Se deja el análisis completo de este programa para la sección de auto-evaluación.

Conclusión

En esta sección se ha presentado un programa de aplicación que ha usado la mayoría de los mecanismos de toma de decisiones disponibles en C. Se ha mostrado de nuevo la estrategia de desarrollo incremental de programas. Compruebe con el siguiente repaso si ha entendido correctamente los conceptos de esta sección.

Repaso de la Sección 3.10

1. ¿Cuál es el primer paso en el desarrollo de cualquier programa?
2. ¿Para qué sirve un diagrama de reparación?
3. Defina qué es un suplente del programa.
4. ¿Qué representan para el programa los datos introducidos por el usuario?

3.11. PROGRAMAS DE APLICACIÓN ADICIONALES

El Programa 3.21 realiza la conversión de un número de decimal a hexadecimal. El número introducido por el usuario se divide en dos mitades y cada mitad tiene cuatro bits que contienen un número del 0 al 15. Usando un `switch`, los valores del 10 al 15 se muestran en la pantalla con la letra correspondiente (de la A a la F).

Programa 3.21 `#include <stdio.h>`

```
void ahex(int n);

main()
{
    int n;        /* Número leído */
    int a,b;      /* Almacenamiento temporal */

    printf("Introduzca un entero del 0 al 255 => ");
    scanf("%i",&n);
```

```

    a = n / 16;      /* Obtiene el valor de los 4 bits superiores. */
    b = n % 16;      /* Obtiene el valor de los 4 bits inferiores. */
    ahex(a);
    ahex(b);
}

void ahex(int n)
{
    if ((n >= 0) && (n <= 9))
        printf("%i", n);
    else
    {
        switch(n)
        {
            case 10 : printf("A"); break;
            case 11 : printf("B"); break;
            case 12 : printf("C"); break;
            case 13 : printf("D"); break;
            case 14 : printf("E"); break;
            case 15 : printf("F"); break;
            default : printf("Error!\n");
        }
    }
}

```

Se muestran a continuación el resultado de dos ejecuciones para ilustrar la conversión a hexadecimal realizada con la ayuda de la función `ahex()`.

Introduzca un entero del 0 al 255 => 100
64

En esta primera ejecución, el número decimal 100 se convierte en hexadecimal dando un valor de 64. Sin embargo, esta conversión no utiliza la sentencia `switch` ya que ninguna de las dos mitades del número es mayor que 9. En la segunda ejecución si se necesita el `switch` para convertir la mitad superior del número 200.

Introduzca un entero del 0 al 255 => 200
C8

Obsérvese que este programa sólo trata números de 8 bits. ¿Es posible convertir números de 16 bits?

En el Programa 3.22 se usa la recursividad para programar el algoritmo de Euclides (300 a.C.).

Programa 3.22 `#include <stdio.h>`

```
int euclid(int a, int b);
```

```

main()
{
    int m;    /* Primer número leído */
    int n;    /* Segundo número leído */

    printf("Introduzca el primer numero => ");
    scanf("%i",&m);
    printf("Introduzca el segundo numero => ");
    scanf("%i",&n);
    printf("\nEl MCD de %i y %i es %i",m,n,euclid(m,n)); }

int euclid(int a, int b)
{
    if (b == 0)
        return a;
    else
        return euclid(b,a % b);
}

```

El algoritmo de Euclides se usa para encontrar el máximo común divisor (MCD) de dos enteros. Por ejemplo, ¿cuál es MCD de 40 y 100? Los factores de estos números son:

```

40:  1  2  4  5  8  10  20  40
100: 1  2  4  5  10  20  25  50  100

```

Como se puede ver, el máximo divisor de ambos números es el 20. La función `euclid()` encuentra este número de forma recursiva usando el operador resto (%). La ejecución del programa para dichos números produce la siguiente salida:

```

Introduzca el primer numero => 40
Introduzca el segundo numero => 100
El MCD de 40 y 100 es 20

```

Intente también con estas parejas de números: 9 y 27, 9 y 28, 36 y 544.

El Programa 3.23 se ocupa de un tema importante para los usuarios de computadores: la velocidad de la memoria. El programa calcula este parámetro para dos tipos de sistema de memoria comunes: la memoria caché y la memoria entrelazada.

Programa 3.23 `#include <stdio.h>`

```

float Cacher(void);
float RAMer(void); main()

main()
{
    char eleccion;    /* Elección del usuario. */
    float Tacc;        /* Tiempo de acceso en nanosegundos. */

    printf("Que le gustaria calcular?\n");
    printf("1) Tiempo medio de acceso a la cache\n");

```

```

printf("2] Tiempo de acceso a la RAM entrelazada\n");
printf("Eleccion ? ");
scanf("%c",&eleccion);
switch(eleccion)
{
    case '1' : Tacc = Cacher(); break;
    case '2' : Tacc = RAMer(); break;
    default  : printf("Perdon. Esa eleccion no es valida.");
}
if ((eleccion == '1') || (eleccion == '2'))
    printf("\nEl tiempo de acceso es %f nanosegundos.", Tacc);
}

float Cacher()
{
    float tac; /* Tiempo de acceso a la cache. */
    float ram; /* Tiempo de acceso a la RAM. */
    float hit; /* tasa de aciertos. */

    printf("Introduzca el tiempo de acceso a cache (en nanoseg.) => ");
    scanf("%f",&tac);
    printf("Introduzca el tiempo de acceso a RAM (en nanoseg.) => ");
    scanf("%f",&ram);
    printf("Introduzca la tasa de aciertos => ");
    scanf("%f",&hit);
    return ((hit * tac) + (1.0 - hit)*(tac + ram));
}

float RAMer()
{
    float n; /* Número de módulos de RAM. */
    float ram; /* Tiempo de acceso a la RAM. */

    printf("Introduzca el numero de modulos de RAM => ");
    scanf("%f",&n);
    printf("Introduzca el tiempo de acceso a RAM (en nanoseg.) => ");
    scanf("%f",&ram);
    return (ram / n);
}

```

En un sistema de memoria caché, se usa una memoria de alta velocidad denominada caché. Este tipo de memorias son típicamente diez veces más rápidas que las memorias RAM estándar. El uso combinado de estos dos tipos de memoria proporciona un tiempo medio de acceso que está comprendido entre los tiempos de acceso de cada tipo. Este tiempo medio de acceso está influenciado por un tercer parámetro denominado **tasa de acierto**. Una tasa de acierto de 0,85 significa que el 85 % de las veces el dato pedido se encuentra en la caché y el 15 % restante se encuentra sólo en la RAM, por lo tanto, el tiempo medio de acceso será una mezcla de ambos. Suponiendo que el

tiempo de acceso a la caché es de 10 nanosegundos y el de la RAM de 70 nanosegundos, la ejecución del programa produce el siguiente resultado:

```
Que le gustaria calcular?
1) Tiempo medio de acceso a la cache
2) Tiempo de acceso a la RAM entrelazada
Eleccion ? 1
Introduzca el tiempo de acceso a cache (en nanoseg.) => 10 Introduzca
el tiempo de acceso a RAM (en nanoseg.) => 70
Introduzca la tasa de aciertos => .85
El tiempo de acceso es 20.499998 nanosegundos.
```

Otro tipo común de memoria es la entrelazada, que está formada por varios módulos de memoria idénticos conectados de manera que se puedan acceder simultáneamente. Gracias a ello, el tiempo medio de acceso es el tiempo de acceso de un módulo dividido por el número de módulos. Se muestra a continuación el resultado producido por el programa en este caso.

```
Que le gustaria calcular?
1) Tiempo medio de acceso a la cache
2) Tiempo de acceso a la RAM entrelazada
Eleccion ? 2
Introduzca el numero de modulos de RAM => 8
Introduzca el tiempo de acceso a RAM (en nanoseg.) => 70
El tiempo de acceso es 8.750000 nanosegundos.
```

Utilice el programa para calcular cuántos módulos de memoria de 80 ns. se necesitan para conseguir un tiempo medio de acceso de 16 ns.

El último programa muestra el uso de la operación AND binaria. Dicho programa se basa en lo que se expone a continuación. La representación ASCII de la letra 'a' tiene el valor hexadecimal 61. La representación de la letra 'A' tiene el valor hexadecimal 41. Comparemos ambos valores:

```
0 1 1 0 0 0 0 1 - 0x61 - 'a'
0 1 0 0 0 0 0 1 - 0x41 - 'A'
```

Obsérvese que sólo se diferencian en el valor del bit 5. En general, el código para cualquier letra minúscula se puede convertir a mayúscula poniendo simplemente el bit 5 a 0, lo cual puede hacerse haciendo un AND binario del código ASCII con la máscara 0xDF que contiene un 0 en la posición correspondiente al bit 5. El Programa 3.24 muestra el uso de esta técnica.

Programa 3.24 #include <stdio.h>

```
main()
{
    int n1 = 0x68, n2 = 0x69;
    printf("%c%c\n", n1, n2);
    n1 &= 0xdf;
    printf("%c%c\n", n1, n2);
}
```

```

    n2 &= 0xdf;
    printf("%c%c\n", n1, n2);
}

```

Este programa produce el siguiente resultado:

```

hi
Hi
HI

```

¿Qué sucedería si se aplica esta estrategia a un código ASCII que no se corresponda con una letra (por ejemplo al '0' o al '=')?

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido aquí para permitirle adquirir una valiosa experiencia y, lo que es más importante, para tener una realimentación inmediata de lo que hacen los conceptos y mandatos presentados en este capítulo. Además son divertidos.

Ejercicios

1. Intente averiguar qué imprimirá este programa.

Programa 3.25 `#include <stdio.h>`

```

main()
{
    printf("Que es esto => %d", (5 > 3));
}

```

2. El Programa 3.26 usa operadores binarios. Nótese que la entrada está en base decimal mientras que la salida está en base hexadecimal. Intente averiguar qué imprimirá este programa.

Programa 3.26 `#include <stdio.h>`

```

main()
{
    char bit1;
    char bit2;

    bit1 = 5;
    bit2 = 10;
    printf("%X\n", bit1&bit2);
    printf("%X\n", bit1%bit2);
    printf("%X\n", bit1=bit2);
    printf("%X\n", ~bit2);
}

```

3. El Programa 3.27 muestra el uso del AND lógico. Intente averiguar qué imprimirá este programa.

Programa 3.27

```
#include <stdio.h>

main()
{
    printf("Que es esto => %d", (1 && 23));
}
```

4. El Programa 3.28 muestra el uso del OR lógico. Intente averiguar qué imprimirá este programa.

Programa 3.28

```
#include <stdio.h>

main()
{
    printf("Que es esto => %d", (0 || 38));
}
```

5. El Programa 3.29 contiene una sentencia lógica compleja. Intente averiguar qué imprimirá este programa.

Programa 3.29

```
#include <stdio.h>

main()
{
    int variable_logica;

    variable_logica = (5 < 7) && ((8 > 5) || (7 > 3));
    printf("Que es esto => %d", variable_logica);
}
```

6. ¿Qué opinión tiene de cómo está escrito el Programa 3.30? ¿Cree que es más fácil de leer y de predecir su salida?

Programa 3.30

```
#include <stdio.h>
#define VERDADERO 1
#define FALSO 0
#define AND &&
#define OR ||
#define NOT !

main()
{
    int valor_logico;

    valor_logico = VERDADERO AND VERDADERO;
    if (valor_logico) printf("VERDADERO");
    else printf("FALSO");
}
```


7. Lea el Programa 3.31 e intente predecir su salida.

Programa 3.31

```
#include <stdio.h>
#define VERDADERO 1
#define FALSO 0
#define AND &&
#define OR ||
#define NOT !

main()
{
    int valor_logico;

    valor_logico = VERDADERO OR NOT VERDADERO;
    switch(valor_logico)
    {
        case FALSO :      printf("FALSO");
                          break;
        case VERDADERO :  printf("VERDADERO");
    }
}
```

8. Lea el Programa 3.32 y luego ejecútelo.

Programa 3.32

```
#include <stdio.h>
#define VERDADERO 1
#define FALSO 0
#define AND &&
#define OR ||
#define NOT !
#define Comprueba_Este_Valor switch
#define Stop break;
#define Es_El case

main()
{
    int valor_logico;

    valor_logico = NOT(VERDADERO OR NOT VERDADERO) AND VERDADERO;
    Comprueba_Este_Valor(valor_logico)
    {
        Es_El FALSO :      printf("FALSO");
                          break;
        Es_El VERDADERO :  printf("VERDADERO");
    }
}
```

Autoevaluación

DIRECTRICES

Responda a las siguientes preguntas acerca del Programa 3.20.

Preguntas

1. ¿Cuántas funciones contiene el programa? Enumérelas.
2. ¿Hay funciones que contengan bifurcaciones abiertas? Si es así, ¿cuáles?
3. ¿Hay funciones que contengan bifurcaciones cerradas? Si es así, ¿cuáles?
4. ¿Qué funciones contiene un switch?
5. Explique el significado de la siguiente sentencia:

```
medida = (medida > 30)? 30 : medida;
```

6. ¿Cuántas variables se usan en el programa?
7. ¿Qué funciones devuelven valores?
8. ¿Qué valores devuelven dichas funciones?
9. ¿Por qué la variable estado_luz no es de tipo float?

Problemas del final del capítulo

Conceptos generales

Sección 3.1

1. Explique los seis operadores de comparación presentados en este capítulo.
2. ¿Qué valor numérico se usa para representar FALSO? ¿Y VERDADERO?
3. Si el símbolo '==' significa igualdad, ¿qué significa el '='?
4. Indique el resultado numérico de las siguientes operaciones, dado que A = 0, B = 3 y C = 8:
 - A. !A
 - B. B==B
 - C. C > A

Sección 3.2

5. Explique el concepto de bifurcación abierta.
6. ¿Qué sentencia se usa para la bifurcación abierta?
7. ¿En qué consiste una sentencia compuesta?
8. ¿Se puede llamar a una función desde dentro de una bifurcación abierta?

Sección 3.3

9. Explique el concepto de bifurcación cerrada.
10. ¿Qué sentencia se usa para la bifurcación cerrada?
11. ¿En qué consiste un if...else compuesto?
12. ¿Se pueden usar sentencias compuestas e invocar a funciones desde una sentencia if...else if...else?

Sección 3.4

13. Calcule el complemento binario de los siguientes valores hexadecimales:
 - A. C
 - B. FE
 - C. 50
14. Calcule el AND binario de los siguientes valores hexadecimales:
 - A. 3 & 5
 - B. E & E
 - C. F & 7
15. Calcule el OR binario de los siguientes valores hexadecimales:
 - A. 3 | 5
 - B. A | 5
 - C. 0 | F
16. Calcule el XOR binario de los siguientes valores hexadecimales:
 - A. 3 ^ 5
 - B. A ^ 5
 - C. F ^ F

Sección 3.5

17. ¿En qué consiste una operación lógica?
18. ¿Qué hace la operación AND lógico y con qué símbolo se representa?
19. ¿Qué hace la operación OR lógico y con qué símbolo se representa?
20. ¿Qué hace la operación NOT lógico y con qué símbolo se representa?

Sección 3.6

21. ¿Es legal sumar un tipo int a un tipo char? ¿Cómo se denomina a este tipo de operación?

22. ¿Qué tipo de resultado daría la operación anterior?
23. Explique el propósito de una conversión.
24. ¿Qué es una expresión valor-i?

Sección 3.7

25. Explique cómo se usa la sentencia switch.
26. ¿Para qué se usa la palabra reservada case en el switch?
27. ¿Para qué se usa la palabra reservada break en el switch?
28. ¿Se pueden usar sentencias compuestas e invocar a funciones dentro de un switch?

Sección 3.8

29. ¿Para qué se usa la palabra reservada default en el switch?
30. ¿Explique cómo funciona el operador condicional?

Sección 3.9

31. ¿Cómo se denomina a una orden dirigida al compilador?
32. ¿En qué consiste la compilación condicional?
33. Nombre una directiva de compilación condicional.

Sección 3.10

34. Explique cómo usa un técnico un diagrama de flujo de reparación?
35. ¿Qué característica del lenguaje C permite desarrollar un programa interactivo que replique el comportamiento de un diagrama de flujo de reparación?

Sección 3.11

36. ¿Se puede usar recursividad para mostrar todos los factores de un número?
37. ¿Cómo se deberían generalizar las técnicas de conversión presentadas para permitir convertir a cualquier base?

Diseño de programas

Se deberá desarrollar la documentación de cada programa que se escriba. Esta documentación debería incluir las líneas generales del diseño que expliquen el propósito del programa, la entrada requerida, el proceso sobre dicha entrada y la salida requerida, junto con el algoritmo del programa. Asegúrese de que incluye toda esta documentación en la versión final del programa.

38. Desarrolle un programa que muestre al usuario todos los tipos de bifurcaciones disponibles en C.
39. Escriba un programa que genere la tabla de verdad correspondiente a la siguiente expresión Booleana:

$$F = AB + AC + ABC$$

40. Escriba un programa que convierta a octal un número introducido por el usuario.
41. Escriba un programa que determine si un entero es par o impar.
42. Escriba un programa que determine si un número representado en binario tiene paridad par o impar.
43. Escriba un programa que calcule el tiempo de ejecución medio de un microprocesador. El usuario introducirá el número medio de ciclos de reloj por instrucción y la frecuencia del reloj. Por ejemplo, 5 ciclos de reloj por instrucción con una frecuencia de reloj de 2 MHz. dará como resultado 2,5 microsegundos por instrucción. El programa deberá convertir el resultado a milisegundos, microsegundos o nanosegundos, la unidad que esté más cerca del resultado.
44. Cree un programa en donde el usuario seleccione el nombre de una figura (rectángulo, triángulo, círculo o paralelogramo) y el programa presente la fórmula del área de dicha figura.

Bucles y recursividad

Objetivos

Este capítulo le da la oportunidad de aprender lo siguiente:

1. El propósito de los bucles en un programa.
2. Diferentes tipos de bucles.
3. Bucles que se emplean en C.
4. La estructura y codificación de los bucles de C.
5. Aplicaciones prácticas de los bucles.
6. Bucles anidados.
7. Técnicas de depuración utilizadas en C.
8. El desarrollo de un programa técnico complejo que requiere el uso de bucles.
9. Los detalles de la recursividad.

Palabras clave

Bucle <code>for</code>	Bucle condicional
Sentencia compuesta	Bucle centinela
Post-incremento	Bucle <code>do-while</code>
Pre-incremento	Bucles anidados
Post-decremento	Error en tiempo de ejecución
Pre-decremento	Trazas
Operador coma	Función de auto-depuración
Operador de evaluación secuencial	Resursividad
Bucle <code>while</code>	Pila en tiempo de ejecución

Contenido

4.1. El bucle <code>for</code>	4.6. Recursividad
4.2. El bucle <code>while</code>	4.7. Programa de aplicación: máquina expendedora
4.3. El bucle <code>do while</code>	4.8. Programas de aplicación adicionales
4.4. Bucles anidados	
4.5. Implementación y depuración de programas	
-

Introducción

Recuerde que tres tipos diferentes de bloques de programa son todo lo que se necesita para solucionar un problema en cualquier lógica de programación, sin importar lo complejo que sea. Éstos son el bloque secuencial, el bloque de selección y el bloque repetitivo o de bucles. Ya se han estudiado los bloques de selección en el capítulo anterior y se ha trabajado con los bloques secuenciales desde los primeros capítulos. En cuanto finalice este capítulo usted ya habrá utilizado los tres tipos de bloques.

Usted descubrirá que una de las ventajas del uso de un computador para el análisis y solución de problemas tecnológicos reside en la utilización del bloque repetitivo o de bucles. El uso de un bloque de este tipo le permite comprobar de forma rápida las condiciones para un intervalo de valores, haciéndole posible indicar fácilmente valores mínimos y máximos.

Este capítulo hace también una introducción a la recursividad para completar el concepto de bucles.

4.1. EL BUCLE `for`

Presentación

Esta sección muestra cómo desarrollar un programa C que le permita hacer algo un número exacto de veces. Esto tiene muchas aplicaciones útiles en el mundo de la tecnología. Ya no estará limitado a solucionar un problema para una única respuesta; ahora será capaz de solucionar un problema cualquier número de veces con valores diferentes cada vez.

¿Cómo es un bucle `for`?

El bucle `for` de C contiene cuatro partes importantes:

1. El valor en el que comienza el bucle.
2. La condición bajo la cual el bucle continúa.
3. Los cambios que tienen lugar en cada bucle.
4. La instrucciones del bucle.

Estas partes se ponen en el bucle `for` de C de la siguiente manera:

```
for(expresión inicial; expresión condicional; expresión de bucle)
    {instrucciones del bucle};
```

El Programa 4.1 muestra el uso del bucle `for`. Este programa incrementa en uno el valor de una variable llamada `tiempo` cada vez que se ejecute el bucle. El valor inicial de la variable es 1 y el bucle se repite mientras el valor de la variable `tiempo` sea menor o igual que 5. En cuanto la variable tome un valor mayor de 5, el programa saldrá del bucle.

Programa 4.1 `#include <stdio.h>`

```
main()
{
    int tiempo;          /* Variable contador. */

    /* El bucle comienza aquí. */
    for(tiempo = 1; tiempo <= 5; tiempo = tiempo + 1)
        printf("Valor del tiempo = %d \n", tiempo);

    /* El bucle termina aquí. */
    printf("Fin del bucle.");
}
```

La ejecución del Programa 4.1 produce la siguiente salida:

```
Valor del tiempo = 1
Valor del tiempo = 2
Valor del tiempo = 3
Valor del tiempo = 4
Valor del tiempo = 5
Fin del bucle.
```

Análisis del programa

El bucle `for` en C comienza con la palabra reservada `for` seguida de la expresión del bucle. La expresión del bucle se divide en cuatro partes:

- El valor con el que comienza el bucle (la expresión inicial)
- La condición bajo la cual el bucle se repite (la expresión condicional)
- Los cambios que tienen lugar en cada vuelta del bucle `for` (la expresión del bucle)
- Qué se hace en cada vuelta del bucle (las instrucciones del bucle)

En el caso de este programa, el bucle `for` que se utiliza es:

```
for(tiempo = 1; tiempo <= 5; tiempo = tiempo + 1)
    printf("Valor del tiempo = %d \n", tiempo);
```

Esto significa que el valor con el que comienza el bucle es `tiempo = 1`, la condición bajo la cual se repite el bucle es `tiempo <= 5` y el cambio que tiene lugar en cada vuelta del bucle es `tiempo = tiempo + 1`. La estructura del bucle `for` en C se muestra en la Figura 4.1

Observe que la expresión del bucle `for` no finaliza con un punto y coma. Esto se debe a que la combinación de la palabra reservada `for`, la expresión del bucle y la sentencia que constituye el cuerpo del bucle forman una única sentencia en C:

```
for(tiempo = 1; tiempo <= 5; tiempo = tiempo + 1)
    printf("Valor del tiempo = %d\n", tiempo);
```

Más de una sentencia

Para tener más de una sentencia dentro de un bucle `for`, es necesario encerrar todas ellas entre llaves `{ y }`, dando lugar a lo que se denomina una **sentencia compuesta**. El uso de una sentencia compuesta se ilustra en el Programa 4.2. Este programa calcula la distancia cubierta por un cuerpo que cae (en metros por segundos) para los primeros 5 segundos de caída libre. Esta distancia viene dada por la siguiente ecuación:

$$S = 1/2 at^2$$

donde

S = La distancia en metros.

a = Aceleración debida a la gravedad (9,78 m/s²).

t = Tiempo en segundos

Programa 4.2

```
#include <stdio.h>
#define a 9.78

main()
{
    int tiempo;      /* Variable contador. */
    int distancia;   /* Distancia cubierta por el cuerpo que cae. */

    /* El bucle comienza aquí */
    for(tiempo = 1; tiempo <= 5; tiempo = tiempo + 1)
    {
        distancia = 0.5 * a * tiempo * tiempo;
        printf("La distancia después de %d segundos es de %d\n", tiempo, distancia);
    }
    /* El bucle termina aquí */
    printf("Fin del bucle.");
}
```

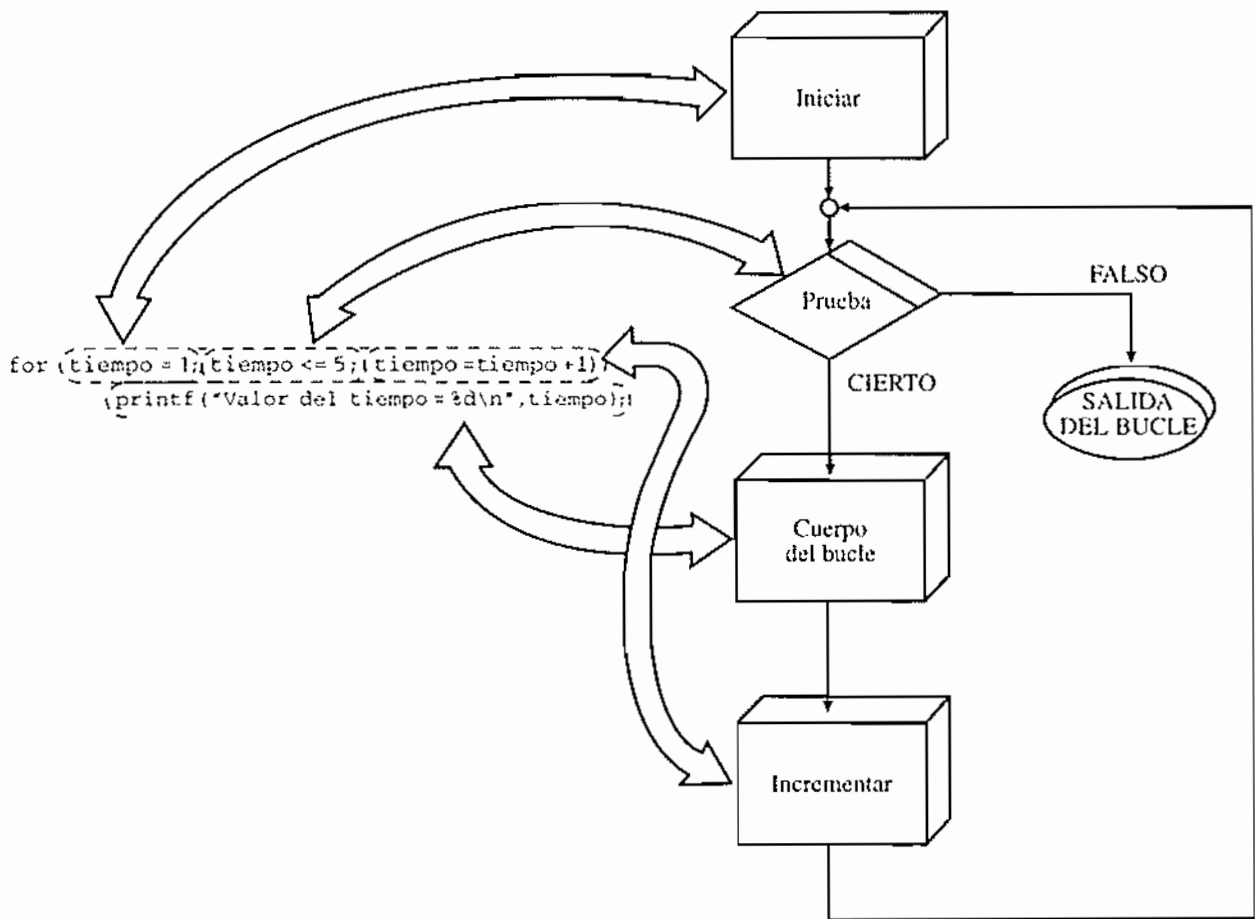


Figura 4.1. Estructura del bucle for, en C.

La ejecución del Programa 4.2 produce la siguiente salida:

```

La distancia después de 1 segundos es de 4 metros.
La distancia después de 2 segundos es de 19 metros.
La distancia después de 3 segundos es de 44 metros.
La distancia después de 4 segundos es de 78 metros.
La distancia después de 5 segundos es de 122 metros.
  
```

Observe que cada una de la sentencias dentro de las llaves { y } finalizan con un punto y coma. Esto se debe a que cada una de ellas es una sentencia completa. Observe también que la llave final de la sentencia compuesta no requiere un punto y coma, de forma semejante a otras sentencias compuestas.

Los operadores de incremento y decremento

C ofrece una notación abreviada para una operación de programación muy común. Esta es la capacidad para incrementar o decrementar un valor. El bucle for del programa anterior utiliza `tiempo = tiempo + 1` para incrementar el valor de la variable `tiem-`

po. En C este incremento puede expresarse de forma reducida como `tiempo++`. En la Tabla 4.1 se muestran los significados de los operadores de incremento y decremento.

El Programa 4.3 ilustra el uso de los operadores de post-incremento y pre-decremento.

Programa 4.3

```
#include <stdio.h>

main()
{
    int contador;           /* Contador. */
    int numero1 = 0;        /* Número que se post-incrementa. */
    int numero2 = 0;        /* Número que se pre-incrementa. */

    for(contador = 1; contador <= 5; contador++)
    {
        printf("Post-incremento del numero = %d ", numero1++);
        printf("Pre-incremento del numero = %d\n", ++numero2);
    }
}
```

La ejecución del Programa 4.3 produce la siguiente salida:

```
Post-incremento del numero = 0 Pre-incremento del numero = 1
Post-incremento del numero = 1 Pre-incremento del numero = 2
Post-incremento del numero = 2 Pre-incremento del numero = 3
Post-incremento del numero = 3 Pre-incremento del numero = 4
Post-incremento del numero = 4 Pre-incremento del numero = 5
```

Nótese que la expresión `numero1++` no incrementa el valor de `numero1` hasta después de su uso. Sin embargo, la sentencia `++numero2` incrementa la variable `numero2` antes de su uso. Observe que a ambos números se les asigna el valor 0 en la parte de declaración del programa.

Tabla 4.1. Operadores de incremento y decremento

Operador	Significado
<code>X++</code>	Incrementa <i>X</i> después de cualquier operación con ella (se llama post-incremento).
<code>++X</code>	Incrementa <i>X</i> antes de cualquier operación con ella (se llama pre-incremento).
<code>X--</code>	Decrementa <i>X</i> después de cualquier operación con ella (se llama post-decremento).
<code>--X</code>	Decrementa <i>X</i> antes de cualquier operación con ella (se llama pre-decremento).

Ejemplo 4.1 Dados los valores que se muestran a continuación, determine los resultados de las siguientes operaciones:

$i = 3$
 $c = 10$

A. $x = i++$	C. $x = c++$	E. $x = i-- + ++c$
B. $x = ++i$	D. $x = --c + 2$	

Solución

- A. $x = i++ = 3++ \Rightarrow$ incrementa después de la asignación, así $x = 3$
 B. $x = ++i = ++3 \Rightarrow$ incrementa antes de la asignación, así $x = 4$
 C. $x = c++ = 10++ \Rightarrow$ incrementa después de la asignación, así $x = 10$
 D. $x = --c + 2 = --10 + 2 = 9 + 2 = 11$
 E. $x = i-- + ++c = 3-- + ++10 = 3 + 11 = 14$

Como se muestra en el Programa 4.4 los resultados de las operaciones de este ejemplo son acumulativos.

Programa 4.4 `#include <stdio.h>`

```
main()
{
    int i = 3;
    int c = 10;

    printf("i = %d\n", i);
    printf("c = %d\n", c);
    printf("x = i++ => %d\n", i++);
    printf("x = ++i => %d\n", ++i);
    printf("x = c++ => %d\n", c++);
    printf("x = --c + 2 => %d\n", (--c + 2));
    printf("x = i-- + ++c => %d\n", (i-- + ++c));
}
```

La ejecución del Programa 4.4 produce la siguiente salida:

```
i = 3
c = 10
x = i++ => 3
x = ++i => 5
x = c++ => 10
x = --c + 2 => 12
x = i-- + ++c => 16
```

Análisis del programa

En el Programa 4.4, el resultado de cada acción es acumulativo. Esto significa que cuando se finaliza $x = i + ++i$, la variable i toma el valor 4 (se incrementa después de la operación). De esta forma, cuando se realiza la siguiente operación ($x = ++i + 5$), el incremento tiene lugar antes de la operación, y debido a que la variable i tiene el valor 4, el incremento antes de la operación producirá un 5.

En la operación $--c + 2$, la variable c comienza con el valor 11 (se incrementó después de la operación anterior, $x = c + ++i$). Debido a que este decremento se realiza antes de la operación, $--c$ hace que c tome el valor 10, y de esta forma $10 + 2 = 12$.

Para la última operación, $x = i - ++c + 16$, i comienza con el valor de su última operación (5) y c tiene el valor 10 de su última operación. Como $++c$ incrementa el valor de c antes de la operación, ésta toma el valor 11, y $5 + 11 = 16$.

Operador coma

Como regla general, dos expresiones en C puede separarse por el **operador coma**:
 expresión, expresión

Esto significa que en C se puede tener más de una expresión dentro de la expresión del bucle `for`. De esta forma se puede asignar valores iniciales a dos variables al mismo tiempo:

```
for (contador = 0, valor = 2; contador <= 3; contador++)
```

Observe que se asignan valores iniciales a dos variables separadas por una coma: `contador` y `valor`. De igual forma podría haber múltiples sentencias en la sentencia de incremento del bucle `for`. Sin embargo, sólo se permite una expresión en la parte de comprobación del bucle. Las expresiones separadas por el operador coma se evalúan de izquierda a derecha. El operador coma se denomina en algunas ocasiones **operador de evaluación secuencial**.

Conclusión

En esta sección se ha presentado la operación del bucle `for` de C. Se han visto los principales elementos de este bucle así como algunos ejemplos de su utilización. Se han presentado también los operadores de incremento y decremento y el operador coma. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 4.1

1. Indique los cuatro elementos importantes del bucle `for` de C.
2. Explique si en un bucle `for` puede haber más de una sentencia.

3. Explique el significado de ++y.
4. ¿Qué es el operador coma?

4.2. BUCLE while

Presentación

Esta sección presenta otro tipo de bucle que ofrece C, denominado bucle `while`. Como se verá, este bucle consta de los mismos elementos del bucle `for`. La diferencia entre ellos se encuentra en la disposición de los elementos dentro del programa. Verá que el uso del bucle `while` es más indicado en aquellas situaciones en las que no se conoce por adelantado el número de veces que se va a repetir el bucle (como por ejemplo un bucle que permita al usuario repetir un programa de forma automática).

Estructura del bucle while

La estructura del bucle `while` es:

```
while (expresión)
    sentencia;
```

La sentencia puede incluir una única sentencia o una sentencia compuesta (en este caso todas las sentencias se encierra entre llaves `{ }`). La sentencia se ejecutará cero o más veces hasta que la expresión tome el valor FALSO.

En un bucle `while` primero se evalúa expresión. Si el resultado de esta evaluación es FALSO (0), entonces no se ejecuta la sentencia y el control continúa después del bucle `while`. Si la evaluación es VERDADERO (distinto de cero), se ejecuta la sentencia y se repite el proceso de nuevo.

El Programa 4.5 muestra un ejemplo de utilización del bucle `while`. Este programa es similar al Programa 4.1. Evalúa la variable tiempo para cinco valores diferentes.

Programa 4.5 `#include <stdio.h>`

```
main()
{
    int tiempo = 1;    /* Variable contador. */
    while(tiempo <= 5)
    {
        printf("Valor del tiempo = %d\n", tiempo);
        tiempo++;
    } /* Fin del while. */
    printf("Fin del bucle.");
}
```

La ejecución de este programa produce la siguiente salida

```
Valor del tiempo = 1
Valor del tiempo = 2
Valor del tiempo = 3
Valor del tiempo = 4
Valor del tiempo = 5
Fin del bucle.
```

Análisis del programa

La operación del bucle `while` realiza la comprobación de la condición antes de la ejecución. El diagrama lógico del Programa 4.5 se ilustra en la Figura 4.2

Siempre que el valor de la condición del bucle `while` sea VERDADERO (distinto de cero), se ejecutará la sentencia que sigue. En el Programa 4.5, fue necesario asignar a la variable contador `tiempo` el valor inicial 1:

```
int tiempo = 1;           /* Variable contador. */
```

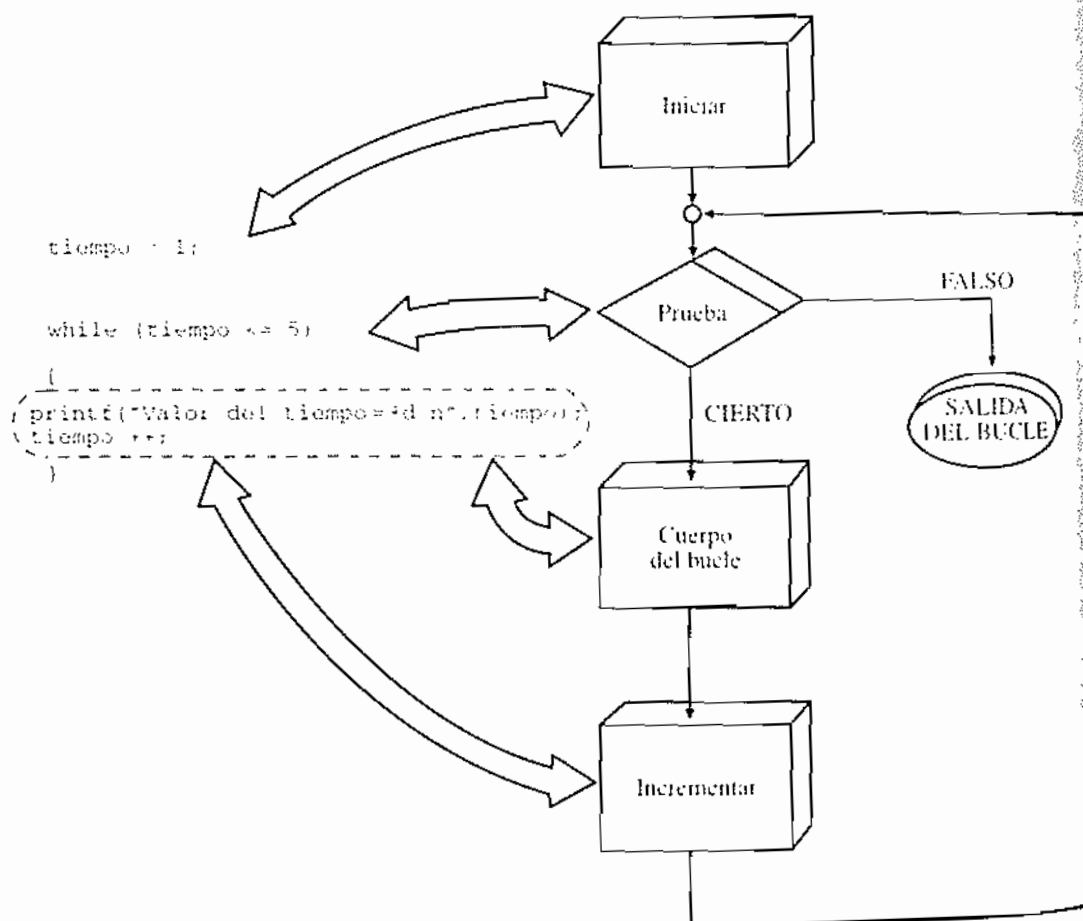


Figura 4.2. Estructura del bucle `while`.

Esto es necesario para asegurar un valor de comienzo. La condición del bucle tomará valor VERDADERO mientras el contador `tiempo` sea menor o igual que 5.

También es necesario actualizar el valor de la variable `contador` dentro del bucle. Si esto no se hiciera, el bucle se repetiría de forma indefinida. La variable `contador`, `tiempo`, se incrementa dentro del cuerpo del bucle:

```
{
    printf("Valor del tiempo = %d\n", tiempo);
    tiempo++;
} /* Fin del while. */
```

Observe que esta sentencia compuesta requiere que las dos sentencias se encuentren encerradas entre llaves { }.

Aplicación del bucle while

El Programa 4.6 ilustra una aplicación del bucle `while` de C. Este bucle es más apropiado que el bucle `for` de C cuando no se conoce de antemano la condición que hace terminar al bucle.

Este programa utiliza una nueva función llamada `getchar()`. Esta función lee un único carácter del teclado y lo imprime en la pantalla. Es una función muy útil cuando se necesita leer un carácter en un programa.

El Programa 4.6 simplemente comprueba la entrada del usuario para ver si el cuerpo del programa debe ser repetido. Observe que para finalizar el bucle, el usuario del programa debe introducir la letra mayúscula N.

```
Programa 4.6 #include <stdio.h>

main()
{
    char respuesta = 'S'; /* Respuesta del usuario. */
    while(respuesta != 'N')
    {
        printf("Este es el cuerpo del programa. \n");
        printf("Quiere repetir este programa (S/N) => ");
        respuesta = getchar();
        printf("\n");
    } /* Fin del while. */
    printf("Gracias por usar este programa. ");
}
```

Un ejemplo de salida para este programa es:

```
Este es el cuerpo del programa.
Quiere repetir este programa (S/N) => S
Este es el cuerpo del programa.
```

```

Quiere repetir este programa (S/N) => n
Este es el cuerpo del programa.
Quiere repetir este programa (S/N) => N
Gracias por usar este programa.

```

Análisis del programa

Este programa comienza asignando a la variable de tipo char respuesta el valor inicial 'S'. Esto asegura que esta variable no comience con un valor de N (una posibilidad remota pero posible).

```
char respuesta = 'S';      /* Respuesta del usuario. */
```

A continuación aparece la condición de continuación del bucle:

```
while (respuesta != 'N')
```

Esto significa que mientras la variable de tipo char respuesta no sea igual a N, se ejecutará el cuerpo del bucle:

```

while(respuesta != 'N')
{
    printf("Este es el cuerpo del programa. \n");
    printf("Quiere repetir este programa (S/N) => ");
    respuesta = getchar();
    printf("\n");
} /* Fin del while. */

```

En cuanto la condición del bucle while tome valor FALSO (lo que significa que la condición respuesta != 'N' es FALSO), el programa continuará con la siguiente sentencia:

```
printf("Gracias por usar este programa. ");
```

Funciones dentro del bucle while

Se puede situar una función C dentro de la expresión del bucle while. Esto con frecuencia permite reducir la cantidad de código fuente. Por ejemplo:

```
while(getchar() != '\r')
```

permitirá al usuario introducir caracteres e imprimirlos en la pantalla hasta que se teclee el retorno de carro. En este caso, lo primero que se hace es ejecutar la función getchar() y a continuación realizar la evaluación.

Conclusión

La característica importante del bucle while de C es que la condición se comprueba antes de ejecutar el bucle. Este tipo de bucle debería utilizarse cuando no se conoce

por anticipado el número de veces que tiene que repetirse el bucle (este bucle se conoce también como **bucle condicional** o **bucle centinela**).

Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 4.2

1. Especifique la construcción del bucle `while` en C.
2. ¿Bajo qué condiciones se repetirá un bucle `while`?
3. ¿Cuándo se comprueba la condición en el bucle `while`?
4. Indique un buen uso para el bucle `while`.

4.3. EL BUCLE `do while`

El último de los tres tipos de bucles de C es el **bucle `do while`**. Como se verá, esta estructura es similar al bucle `while` con la diferencia de que la condición se evalúa después de ejecutar el cuerpo del bucle. Recuerde en el bucle `while`, la condición se evalúa antes de la ejecución del cuerpo del bucle.

La sentencia `do while` de C tiene la siguiente forma:

```
do
    sentencia
while (expresión);
```

donde `sentencia` puede ser una única sentencia o una sentencia compuesta (encerrada entre llaves). `sentencia` se ejecuta una o más veces hasta que `expresión` tome el valor FALSO (valor 0). La ejecución de este bucle es la siguiente: primero se ejecuta `sentencia`; a continuación se evalúa `expresión`. Si `expresión` es FALSO, se termina la ejecución del bucle `do while` y el control continúa en la siguiente sentencia del programa. Por el contrario, si `expresión` es VERDADERO (valor distinto de 0), entonces se repite `sentencia` y el proceso se repite de nuevo.

Empleo del bucle `do while` de C

El Programa 4.7 muestra un ejemplo de utilización del bucle `do while` de C. Observe que éste es el bucle contador de las secciones anteriores. La variable `contador` es `tiempo`. Este programa simplemente incrementa el contador de 1 a 5 en pasos de 1.

Programa 4.7 `#include <stdio.h>`

```
main()
{
    int tiempo;    /* Variable contador. */
    tiempo = 1;
```

```

do {
    printf("Valor del tiempo = %d\n", tiempo);
    tiempo++;
}while(tiempo <= 5);
printf("Fin del bucle.");
}

```

Este programa produce la siguiente salida:

```

Valor del tiempo = 1
Valor del tiempo = 2
Valor del tiempo = 3
Valor del tiempo = 4
Valor del tiempo = 5
Fin del bucle.

```

Observe que la salida no se diferencia en nada a la del programa que utiliza el bucle `while` (Programa 4.5). La lógica del programa, sin embargo, es diferente puesto que el cuerpo del bucle se ejecuta antes que la evaluación de la condición.

Análisis del programa

En primer lugar, a la variable contador se le asigna el valor inicial 1:

```
tiempo = 1;
```

A continuación el programa ejecuta el cuerpo del bucle `do` de C:

```

do {
    printf("Valor del tiempo = %d\n", tiempo);
    tiempo++;
}

```

Observe que la palabra reservada `do` es seguida por una sentencia compuesta. Esta es la parte de acción del bucle. A continuación se evalúa la condición que permite repetir el bucle. Esto se realiza en la sentencia `while`:

```
while (tiempo <= 5);
```

Esta sentencia indica que mientras la variable `tiempo` sea menor o igual que 5 el bucle continuará repitiéndose. En la Figura 4.3 se ilustra la construcción del bucle `do while` de C.

Detalles del bucle

Para resaltar el hecho de que el bucle `do` de C siempre se ejecuta al menos una vez, observe el Programa 4.8. En este programa la condición del `while` toma valor `FALSO` (su argumento es cero).

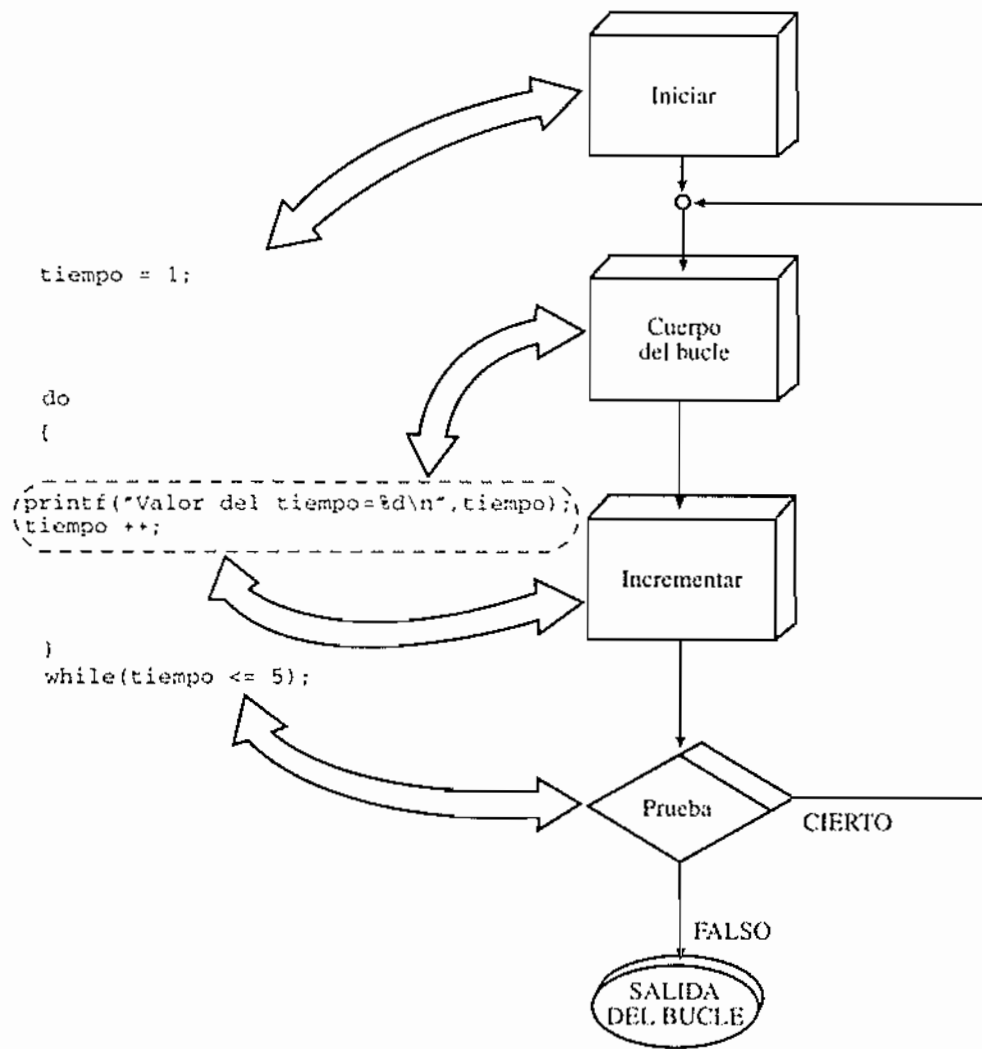


Figura 4.3. Estructura del bucle do while.

Programa 4.8 `#include <stdio.h>`

```

main()
{
    do
    printf("Esto siempre sucede al menos una vez...");
    while(0);
    printf("Fin del bucle.");
}
    
```

La salida de este programa es:

```

Esto siempre sucede al menos una vez ...
Fin del bucle.
    
```

Ahora considere el Programa 4.9. Este programa contiene un bucle `while`. El argumento de este es FALSO. En este caso el cuerpo del bucle nunca se ejecuta ya que en este bucle la condición siempre se evalúa en primer lugar.

Programa 4.9

```
#include <stdio.h>

main()
{
    while(0)
        printf("Esto no sucedera si la condicion no se cumple...");
    printf("Fin del bucle.");
}
```

Bucles centinelas

Tanto el bucle `do` como el bucle `while` de C pueden ser utilizados como bucles centinelas. El Programa 4.10 ilustra el empleo de un bucle `do` como bucle centinela. En este programa se pide al usuario que introduzca un valor que sea menor que 5. El programa comprueba si el valor introducido es efectivamente menor que 5. Si esto no ocurre, el programa avisa al usuario y repite el bucle hasta que se introduzca un valor dentro del intervalo solicitado.

Programa 4.10

```
#include <stdio.h>

main()
{
    float entrada;    /* Número leído */

    do
    {
        printf("Introduzca un numero menor que 5 => ");
        scanf("%f",&entrada);
        if (entrada >= 5)
            printf("El valor es demasiado grande, pruebe de nuevo:\n");
    }
    while(entrada >= 5);
    printf("Gracias...");
}
```

La ejecución del Programa 4.10 podría producir como salida:

```
Introduzca un numero menor que 5 -> 7
El valor es demasiado grande, pruebe de nuevo:
```

```

Introduzca un numero menor que 5 => 25
El valor es demasiado grande, pruebe de nuevo:
Introduzca un numero menor que 5 => 3
Gracias...

```

Análisis del programa

En este programa no existe ningún contador puesto que se trata de un bucle centinela. En primer lugar se ejecuta el cuerpo del bucle `do`:

```

do
{
    printf("Introduzca un numero menor que 5 => ");
    scanf("%f",&entrada);

```

Esta primera parte espera hasta que el usuario introduzca un número. La siguiente parte del programa comienza con un `if` de C:

```

if (entrada >= 5)
    printf("El valor es demasiado grande, pruebe de nuevo:\n");
}

```

Esta parte del programa es necesaria para indicar al usuario el hecho de que el valor introducido no es aceptable (es decir, es mayor o igual que 5) y solicitarle que introduzca de nuevo un valor dentro del intervalo solicitado. Si se introduce un valor correcto dentro del intervalo, entonces no se ejecutará la función `printf` situada dentro del bucle.

El bucle `do while` de C finaliza con la comprobación `while`:

```

while (entrada >= 5)

```

Comparación entre bucles

Tanto el bucle `while` como el bucle `do while` de C pueden utilizarse cuando no se conoce de antemano el número de veces que se repetirá el bucle. A la hora de elegir entre uno de los dos, se recomienda emplear el bucle `while` en vez del bucle `do while`. La razón de esto es que el bucle `while` evalúa en primer lugar la condición antes de ejecutar el bucle. Esto es análogo a comprobar la temperatura del agua antes de sumergirse en ella.

Conclusión

En esta sección se ha presentado el último de los tres tipos de bucles diferentes disponible en C. Se ha visto el bucle `do while` de C y se ha comparado con el bucle `while`. La diferencia entre estos bucles se encuentra en que mientras en el primero se ejecuta el cuerpo del bucle y luego se evalúa la condición, en el segundo primero se evalúa la condición y después se ejecuta el cuerpo del bucle.

Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 4.3

1. Especifique la construcción del bucle `do while` de C.
2. Bajo qué condiciones se repite el bucle `do while`?
3. ¿Cuándo se comprueba la condición en el bucle `do while` de C?
4. Explique qué tipo de bucle es más conveniente, el bucle `while` o el bucle `do while`.

4.4. BUCLES ANIDADOS

Presentación

En esta sección se presenta el concepto de inclusión de un bucle dentro de otro. Esto le da una mayor flexibilidad en el desarrollo de programas en C que solucionan problemas tecnológicos. También se verán métodos que le ayudarán a estructurar sus bucles anidados, de tal manera que éstos sean más fáciles de leer y menos propensos a errores.

Idea básica

El Programa 4.11 contiene un **bucle anidado**, es decir, un bucle dentro de otro. El programa tiene dos contadores e imprime el valor de cada uno de ellos. Ambos son bucles `for` de C con un contador diferente cada uno.

Programa 4.11 `#include <stdio.h>`

```
main()
{
    int contador_externo;    /* Contador para el bucle externo. */
    int contador_interno;    /* Contador para el bucle interno. */

    for(contador_externo = 0; contador_externo <= 3; contador_externo++)
    {
        printf("Comienzo del bucle externo: n");
        printf("Contador del bucle externo=>
                %d\n", contador_externo);

        for(contador_interno = 0; contador_interno <= 3;
            contador_interno++)
        {
            printf("Contador del bucle interno =>
                    %d\n", contador_interno);
        } /* Fin del bucle interno. */
    }
```

```

        printf("Fin del bucle externo. \n\n");
    } /* Fin del bucle externo. */
}

```

La ejecución del Programa 4.11 produce la siguiente salida:

```

Comienzo del bucle externo.
Contador del bucle externo=> 0

Contador del bucle interno => 0
Contador del bucle interno => 1
Contador del bucle interno => 2
Contador del bucle interno => 3

Fin del bucle externo.

Comienzo del bucle externo.
Contador del bucle externo=> 1

Contador del bucle interno => 0
Contador del bucle interno => 1
Contador del bucle interno => 2
Contador del bucle interno => 3

Fin del bucle externo.

Comienzo del bucle externo.
Contador del bucle externo=> 2

Contador del bucle interno => 0
Contador del bucle interno => 1
Contador del bucle interno => 2
Contador del bucle interno => 3

Fin del bucle externo.

Comienzo del bucle externo.
Contador del bucle externo=> 3

Contador del bucle interno => 0
Contador del bucle interno => 1
Contador del bucle interno => 2
Contador del bucle interno => 3

Fin del bucle externo.

```

Análisis del programa

El bucle externo es el primer bucle del Programa 4.11:

```

for(contador_externo = 0; contador_externo <= 3; contador_externo++)
{
    printf("Comienzo del bucle externo.\n");
    printf("Contador del bucle externo=> %d\n\n", contador_externo);
}

```

El contador de este bucle comienza en 0 y en su primera pasada activa el bucle anidado:

```
for(contador_interno = 0; contador_interno <= 3; contador_interno++)
{
    printf("Contador del bucle interno => %d\n", contador_interno);
} /* Fin del bucle interno. */
```

El bucle anidado es también otro bucle `for` de C y su contador comienza en 0. Una vez iniciado el bucle, éste se repetirá hasta que la condición `contador_interno <= 3` se haga FALSO, es decir, `contador_interno` sea mayor que 3. Cuando esto ocurre, se sale del bucle y el flujo del programa continúa en el resto del bucle externo:

```
    printf("Contador del bucle interno => %d\n", contador_interno);
} /* Fin del bucle interno. */
```

Aquí termina el bucle externo el cual vuelve al comienzo incrementando su propio contador en uno. De esta manera, el bucle interno incrementa su contador de 0 a 3 en cada incremento del bucle externo.

Estructura de un bucle anidado

La Figura 4.4 muestra la estructura recomendada para los bucles anidados.

Como se puede observar en la figura, los bucles deberían anidarse de tal manera que permitan fácilmente determinar el comienzo y fin de cada uno. Esta estructura debería incluir algún comentario, que indique claramente a que bucle pertenece la llave de cierre `}`. Esto ayudará a minimizar los errores de programación.

```
/* BUCLE EXTERIOR */
{
    /* BUCLE ANIDADO */
    {
        /* FIN DEL BUCLE ANIDADO */
    }
}
/* FIN DEL BUCLE EXTERIOR */
```

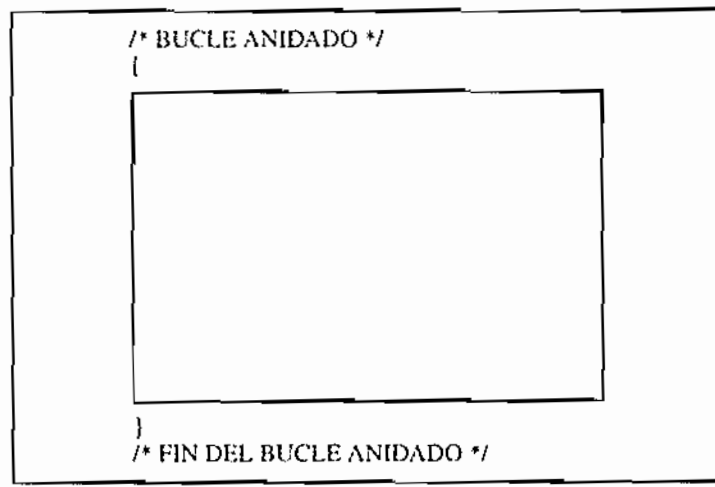


Figura 4.4. Estructura recomendada para bucles anidados.

Anidamiento con bucles diferentes

Se pueden anidar muchos niveles de bucles, tantos como permita el hardware. Sin embargo, cuando se anidan varios niveles y diferentes tipos de bucles, debería tenerse especial cuidado, puesto que los resultados pueden no ser exactamente los esperados. Considere, por ejemplo, los bucles anidados del Programa 4.12. Este programa contiene tres bucles anidados. El bucle más externo es un bucle `for`, el siguiente es un bucle `while` y el más interno un bucle `do`. Lea el programa e intente predecir los resultados. Luego vea la salida que se presenta en el texto puesto que pueden ser resultados sorprendentes.

Programa 4.12 `#include <stdio.h>`

```
main()
{
    int contador_1 = 0; /* Contador para el primer bucle. */
    int contador_2 = 0; /* Contador para el segundo bucle. */
    int contador_3 = 0; /* Contador para el tercer bucle. */

    for(contador_1 = 0; contador_1 <= 2; contador_1++)
    {
        while(contador_2++ <= 3)
        {
            do
            {
                printf("contador_1 = %d\n", contador_1);
                printf("contador_2 = %d\n", contador_2);
                printf("contador_3 = %d\n", contador_3);
            }
            while(contador_3++ <= 3); /* Fin del tercer bucle. */
        } /* Fin del segundo bucle. */
    } /* Fin del primer bucle. */
}
```

Cuando se ejecuta el Programa 4.12, su salida produce:

```
contador_1 = 0
contador_2 = 1
contador_3 = 0
contador_1 = 0
contador_2 = 1
contador_3 = 1
contador_1 = 0
contador_2 = 1
contador_3 = 2
contador_1 = 0
contador_2 = 1
```

```

contador_3 = 3
contador_1 = 0
contador_2 = 1
contador_3 = 4
contador_1 = 0
contador_2 = 2
contador_3 = 5
contador_1 = 0
contador_2 = 3
contador_3 = 6
contador_1 = 0
contador_2 = 4
contador_3 = 7

```

Análisis del programa

Los resultados del Programa 4.12 pueden no ser los esperados. A continuación se realiza un análisis más exhaustivo del mismo.

Primero, a todas las variables utilizadas como contador se les asigna el valor inicial 0:

```

int contador_1 = 0; /* Contador para el primer bucle. */
int contador_2 = 0; /* Contador para el segundo bucle. */
int contador_3 = 0; /* Contador para el tercer bucle. */

```

A continuación se activa el bucle externo, for:

```

for(contador_1 = 0; contador_1 <= 2; contador_1++)
{

```

Este bucle se ejecutará mientras su contador sea menor o igual que 2. En la primera pasada el valor de su contador se pone a 0. Una vez hecho esto se ejecuta el segundo bucle (la primera instrucción del bucle más externo):

```

while(contador_2++ <= 3)
{

```

Este es un bucle while. Primero se evalúa el contador y luego se incrementa en 1. Como el resultado de la condición es VERDADERO, se ejecutará la primera instrucción del bucle, que es el comienzo de un bucle do:

```

do
{
    printf("contador_1 = %d\n", contador_1);
    printf("contador_2 = %d\n", contador_2);
    printf("contador_3 = %d\n", contador_3);
}

```

Recuerde que un bucle `do` en C ejecuta su bucle antes de comprobar la condición. De esta manera todas las funciones `printf` se ejecutarán sin importar el valor del contador del bucle `do`, y el programa mostrará el valor de cada uno de los contadores:

```
contador_1 = 0
contador_2 = 1
contador_3 = 0
```

La razón por la que el contador del tercer bucle es aún 0 se debe a que todavía no ha sido incrementado. No se incrementará hasta el `while` final, donde el contador se incrementará y evaluará.

```
while(contador_3++ <= 3); /* Fin del tercer bucle. */
```

Debido a que el resultado de esta comprobación es VERDADERO, el bucle `do` interno se repetirá hasta que su contador alcance el valor de 4. En este momento se saldrá del bucle y se continuará en el siguiente nivel, que es el bucle `while`.

Aquí, se incrementa el segundo contador. Nótese que el primer contador no se ha incrementado debido a que el programa ha salido sólo al siguiente nivel de bucles (del más interno al siguiente nivel). Este siguiente nivel no ha completado aun su condición, y por tanto continúa activo.

Debido a que el bucle `while` continúa activo, se ejecuta su primera instrucción, que es el bucle `do`. Aunque el contador del bucle `do` haya alcanzado ya su cuenta, se ejecuta antes de comprobar la condición (sin importar cuál sea ésta). De esta manera su contador vuelve a ser incrementado, y esto es por lo que `contador_3` toma eventualmente el sorprendente valor de 7. Esta es otra razón por la que es preferible el uso del bucle `while` sobre el bucle `do` de C.

En cuanto el bucle `while` cumple la condición de terminación, vuelve de nuevo a la ejecución del bucle `for`. Este bucle incrementa ahora su contador y ejecuta la primera instrucción de su bucle, que es el bucle `while`. Sin embargo, el bucle `while` ha completado ya su requisito y por tanto no se activa la ejecución del bucle `do` ni ninguna de sus funciones `printf`. Así no se verán los resultados del contador del bucle externo.

Conclusión

Esta sección ha presentado el concepto de los bucles anidados en C. Sin embargo, a diferencia de otros lenguajes de programación, el uso de bucles anidados en C puede ser un poco complicado. Es importante comprender el funcionamiento de los bucles y la acción del operador `++` de C cuando se utiliza en una instrucción de un bucle.

Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 4.4

1. ¿Qué es un bucle anidado?
2. Explique la estructura que debería utilizarse en los bucles anidados.

3. ¿Cuál de los tres tipos de bucles de C puede anidarse?
4. Indique un posible problema que lleva asociado el uso del bucle `do` de C.

4.5. IMPLEMENTACIÓN Y DEPURACIÓN DE PROGRAMAS

Errores en tiempo de ejecución

Usted probablemente haya experimentado la frustración que supone desarrollar un programa que compila de forma correcta y encontrar que cuando se ejecuta no se obtienen los resultados esperados. Este tipo de errores se denominan **errores en tiempo de ejecución**. Estos errores se pueden deber a una fórmula incorrecta o a un error cometido al introducir una fórmula. Pueden ocurrir también cuando se olvida asignar un valor inicial a ciertas variables o cuando no se estructuran de forma correcta los bucles o bifurcaciones.

Muchas veces, los errores en tiempo de ejecución pueden ser los más difíciles de analizar. El compilador no ofrece ninguna ayuda, debido a que el código es correcto en C. El problema se encuentra en algún lugar del programa puesto que no se obtienen los resultados requeridos o los que se obtienen son incorrectos.

Existen varias técnicas para encontrar este tipo de errores. Una es leer cuidadosamente el programa (o disponer de alguien que lo lea). Ésta es una técnica muy adecuada para programas pequeños. Sin embargo, en programas grandes compuestos de muchas funciones con bucles y bifurcaciones complejas, la lectura del código puede no ser el método más eficiente. Muchos programadores utilizan una técnica denominada **traza**. Es fácil de añadir a cualquier programa y es posible que usted quiera considerar este método para su propio uso.

Un problema de ejemplo

En primer lugar se muestra un programa que presenta un problema que será explicado. A continuación se introducirá en el programa una rutina de depuración y se verá cómo activar dicha función para obtener una pista de lo que está ocurriendo en el programa. Observe el Programa 4.13.

Programa 4.13

```
main()
{
    int contador; /* Contador del bucle. */
    contador = 0;
    while(contador != 9)
    {
        contador += 2;
        /* Cuerpo del bucle... */
    } /* Fin del while. */
}
```

El Programa 4.13 se ejecuta de forma indefinida debido a que la condición de parada del bucle nunca se cumple. Este programa compila con éxito debido a que no tiene ningún error sintáctico. Con este programa usted experimentará un error en tiempo de ejecución.

Todo lo que necesita hacer es observar qué es lo que está ocurriendo en el bucle `while` del programa. Un bloque de depuración consta de dos partes, una sección de visualización y una sección de arranque/parada.

La sección de visualización provoca que los valores de una o más variables sean impresos por pantalla, mientras que la sección de arranque/parada permite ejecutar paso a paso el programa. En el Programa 4.14 se muestra un bloque de depuración que imprime el valor del contador del bucle.

Programa 4.14 `#include <stdio.h>`

```
main()
{
    int contador; /* Contador del bucle. */
    char c;       /* Entrada para continuar. */

    contador = 0;
    while(contador != 9)
    {
        contador += 2;
        /* Bloque de depuración */
        printf("Contador = %d\n", contador);
        scanf("%c", &c);
        /* Cuerpo del bucle... */
    } /* Fin del while. */
}
```

Cuando se ejecuta el Programa 4.14 la salida es la siguiente:

```
Contador = 0
[Return]
Contador = 2
[Return]
Contador = 4
[Return]
Contador = 6
[Return]
Contador = 8
[Return]
Contador = 10
[Return]
```

Observe que ahora se puede ejecutar el programa paso a paso y ver la acción del contador del bucle. Su valor se incrementa de forma continuada, lo que confirma lo

que se había sospechado. Sin embargo, la rutina de depuración muestra rápidamente que el programa nunca alcanzará la condición de salida del bucle.

Auto-depuración

Si usted se encuentra en el proceso de desarrollo de un gran programa y está comprobando varias funciones completadas parcialmente dentro del programa, usted puede considerar adecuado el uso de un bloque de auto-depuración. Este es similar al método descrito anteriormente, con la ventaja añadida de que se puede fácilmente activar o desactivar su ejecución con un simple mandato. Esto es especialmente útil cuando se tienen varias funciones con bloques de depuración en ellas. El Programa 4.15 muestra un ejemplo de uso de este método.

Programa 4.15

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

main()
{
    int contador;    /* Contador del bucle. */
    char c;          /* Entrada para continuar. */
    int depuracion;  /* Indicador de depuración. */

    depuracion = TRUE;
    contador = 0;
    while(contador != 9)
    {
        contador += 2;
        /* Bloque de depuración */
        if (depuracion)
        {
            printf("Contador = %d\n", contador);
            scanf("%c", &c);
        }
        /* Cuerpo del bucle... */
    } /* Fin del while. */
}
```

En el Programa 4.15 se han definido TRUE (VERDADERO) y FALSE (FALSO) utilizando la directiva #define. Si se utiliza depuracion = FALSE, el bloque de depuración nunca se ejecutará.

Función de auto-depuración

Una **función de auto-depuración** es simplemente una función C que se incluye en un programa. Esto no es factible con todos los tipos de programa, pero es particularmente

útil para ciertos programas donde los tipos de las variables son los mismos. El Programa 4.16 ilustra el uso de una función de auto-depuración.

Programa 4.16

```
#include <stdio.h>
#define VERDADERO 1
#define FALSO 0

void depuracion(int activa, int muestra, int pasoapaso);

main()
{
    int contador; /* Contador del bucle. */

    contador = 0;
    while(contador != 9)
    {
        contador += 2;
        /* Rutina de depuración */
        depuracion(VERDADERO, contador, VERDADERO);
        /* Cuerpo del bucle... */
    } /* Fin del while. */
    exit(0);
}

void depuracion(int activa, int muestra, int pasoapaso)
{
    char c; /* Entrada para continuar. */

    if (activa == VERDADERO)
    {
        printf("%d", muestra);
        if (pasoapaso == VERDADERO)
            scanf("%c", &c);
    } /* Fin del if */
}
```

Observe que esta función puede ser llamada desde cualquier parte del programa. Ésta muestra el valor de una variable dada, ofrece la posibilidad de ejecución paso a paso y una opción de activación o desactivación. Éste puede ser un método muy útil para depurar programas grandes de forma muy rápida. Las funciones de auto-depuración pueden eliminarse del programa final. Muchos programadores conservan dos copias del código fuente de sus programas, una con funciones de auto-depuración y otra sin ellas. El procedimiento para hacer esto consiste en modificar la versión con las funciones de auto-depuración, hacer una copia de la misma, y eliminar a continuación las órdenes de depuración del código. En este momento se dispondrá de una nueva copia de la versión modificada pero sin las funciones de depuración. Otra posibilidad es utilizar la característica de compilación condicional que se presentó en la Sección 3.9. La mayoría de los entornos de C tienen depuradores.

Conclusión

Esta sección ha presentado algunos métodos de depuración de programas grandes escritos en C. Se ha visto cómo incluir órdenes básicas de depuración así como el modo de desactivarlas fácilmente. También se ha visto cómo introducir depuración automática que puede ser utilizada en ciertos tipos de programas.

Utilice alguno de estos métodos en los siguientes programas que desarrolle. Ahora, compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la misma.

Repaso de la Sección 4.5

1. Explique el significado de error en tiempo de ejecución.
2. ¿Puede detectar el compilador los errores en tiempo de ejecución? Explíquelo.
3. Indique lo que contiene normalmente un bloque de depuración.
4. ¿Qué se entiende por función de auto-depuración?

4.6. RECURSIVIDAD

Presentación

Un programa que hace uso de la recursividad contiene una función que se llama a sí misma. Como se muestra en la Figura 4.5, un programador puede utilizar *recursividad directa* o *indirecta* (o ambas). En la Figura 4.5(a), la función `yoismo()` se llama a sí misma directamente. En la Figura 4.5(b), las funciones `primera()` y `segunda()` forman un ciclo recursivo que se repite. Un programa que contiene una función recursiva trabaja de forma similar a uno que contenga un bucle. Sin embargo, cada vez que una función recursiva se llama a sí misma, se sitúa en memoria una copia completa de la misma. Esto no ocurre, sin embargo, cuando se ejecuta un bucle.

Pila en tiempo de ejecución

El entorno de programación de C soporta recursividad mediante el uso prudente de una **pila en tiempo de ejecución**, una estructura de datos especial utilizada para almacenar las variables y los valores de los parámetros para cada función en un programa. Cuando un programa llama a una función detrás de otra, la pila en tiempo de ejecución crece. Cuando una función termina, la pila en tiempo de ejecución se hace más pequeña. La pila en tiempo de ejecución es gestionada automáticamente por el programador. Lo que el programador tiene que hacer es escribir código de tal manera que la recursividad pare en algún nivel. De otro modo, la pila en tiempo de ejecución crecerá tanto que se acabará la memoria y el programa terminará con un error.


```

        return(1);
    else
        return(num * fact(num-1));
}

```

La función `fact()` calcula el *factorial* de un número entero no negativo. Por ejemplo, el factorial de 4 (escrito de forma matemática como $4!$) es igual a $4*3*2*1$, o lo que es lo mismo, 24. `fact()` es una función recursiva, debido a que la sentencia

```
return(num * fact(num-1));
```

representa una llamada directa a `fact()`. Pero esta es la segunda de dos sentencias `return`. La primera es:

```
return(1);
```

Esta no llama a `fact()` y permite una parada de la recursividad. Puesto que cada nueva llamada a `fact()` se realiza con un valor más pequeño de `num`, en algún momento `fact()` será llamada con `num` igual a cero, es decir, `fact(0)`. La comprobación dentro de la sentencia `if`

```

if (num == 0)
    return(1);
else
    return(num * fact(num-1));

```

determina el momento en el que se para la recursividad.

Supóngase que cuando se ejecuta el Programa 4.17, el usuario introduce 4 como valor de `N`. La Figura 4.6 muestra los valores asociados con el parámetro `num` y los valores que se devuelven en cada nivel de la recursividad. Observe que esta información se pasa en cada nueva copia de `fact` y se devuelve de igual forma cuando cada una de ellas finaliza.

Hay cuatro niveles de recursividad, uno para cada llamada recursiva a `fact()`. Aunque hay cinco copias de `fact()` en memoria cuando finaliza la recursividad, no se cuenta la llamada que realiza la función `main` a `fact()` como una llamada recursiva. Sólo cuando `fact()` se llama a sí misma se obtiene un nuevo nivel de recursividad.

Cuando cada llamada recursiva a `fact()` finaliza, devuelve un valor al nivel anterior (desde el cual fue llamado). La última llamada a `fact()` devuelve 1 para obligar a que $0!$ sea 1, como se define de forma matemática.

Un ejemplo de ejecución del Programa 4.17 es el siguiente:

```

Introduzca el valor de N : 4
N! es igual a 24

```

Pruebe también con otros valores de `N`. Por ejemplo, $8!$ es igual a 40,320, pero éste podría no ser el valor devuelto por el programa. Puede usted explicar porqué.

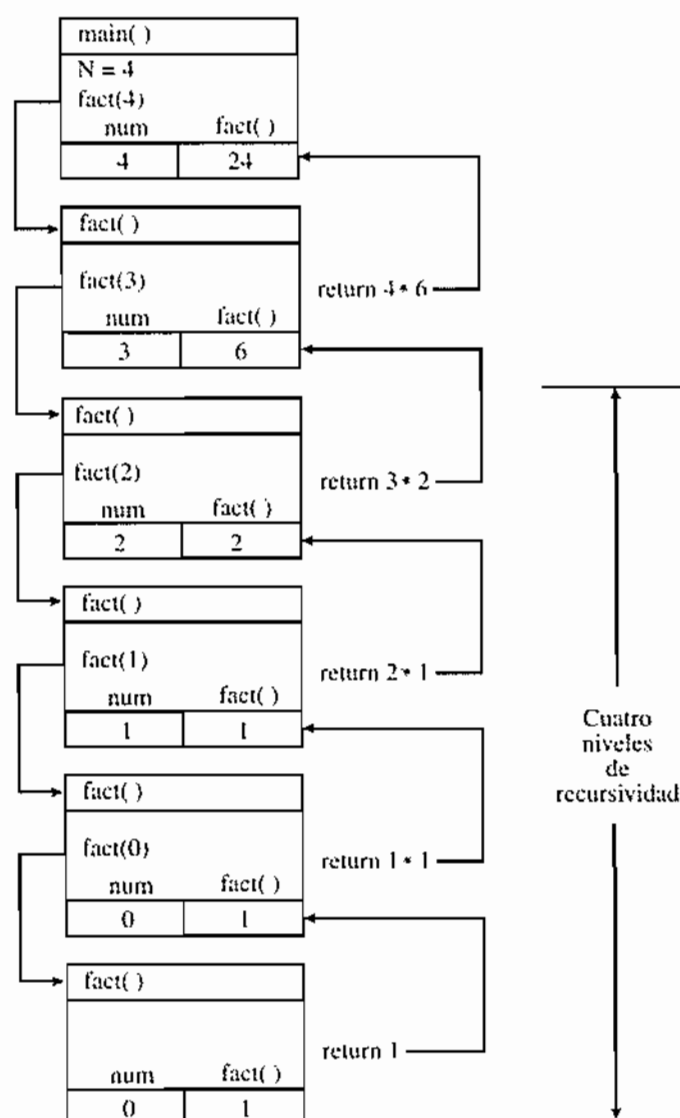


Figura 4.6. Llamadas a función recursivas.

Conclusión

La recursividad es una técnica de programación muy potente que requiere habilidad y paciencia para implementarse correctamente. Experimente con algunos programas anteriores de este capítulo para convertirlos en programas recursivos. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 4.6

1. ¿Qué es una función recursiva?
2. ¿Cómo se implementan las funciones recursivas?
3. ¿Cómo se para la recursividad cuando dos o más funciones se encuentran involucradas en recursividad indirecta?

4. ¿Por qué es necesario parar la recursividad en algún punto?
5. ¿Qué sentencia `for` realiza el mismo trabajo que la función `printf`?

4.7. PROGRAMA DE APLICACIÓN: MÁQUINA EXPENDEDORA

Presentación

En esta sección se desarrolla y examina la operación de una simple máquina expendedora. Esta máquina acepta monedas de 5, 10 y 25 pesetas. Todos los productos de la máquina cuestan 50 pts. Cuando se compra un producto, se utiliza como cambio el mínimo número de monedas de 5, 10 y 25 pesetas.

El problema

Para que la máquina expendedora funcione apropiadamente, la máquina debe advertir al usuario de que para recibir un producto deben insertarse al menos 50 pesetas. De esta forma, es necesario conocer la cantidad de dinero introducida por el usuario. Cuando se adquiere un producto el cambio mínimo puede ser diferente cada vez, debido a que el usuario puede introducir más dinero que el requerido por la máquina. Por ejemplo, si el usuario introduce cuatro monedas de 5 pesetas y dos monedas de 25 pesetas, el cambio mínimo serán dos monedas de 10 pesetas.

Desarrollo del algoritmo

Las etapas requeridas en la aplicación de la máquina expendedora son las siguientes:

1. Obtener el dinero de usuario moneda a moneda.
2. Restar el precio marcado (siempre 50 pts.).
3. Devolver el cambio mínimo (si es necesario).

Cada etapa requiere conocer la cantidad de dinero existente en la máquina. Observe que el cambio se realiza teniendo en cuenta que la máquina nunca se quedará sin dinero. Por ejemplo, si un usuario paciente introduce 1000 monedas de 5 pesetas, la máquina devolverá 198 monedas de 25 pesetas como cambio. Una cosa diferente es considerar el número de monedas disponible de cada tipo, sin embargo, esto no se implementa aquí.

La introducción de las monedas es relativamente sencilla. El usuario introduce '5', '2' o '1' (cinco, dos o un duro) tantas veces como desee. Una variable se incrementa en 25, 10 o 5 cada vez, dependiendo de la entrada del usuario.

Cuando el usuario quiere comprar un producto, introduce una 'P'. Si la cantidad introducida es menor de 50 pesetas, se necesita más dinero. Si la cantidad es suficiente, se resta 50 a la cantidad introducida.

Para devolver el cambio mínimo, se utilizan tres etapas similares en el siguiente orden:

- Restar tantas monedas de 25 pesetas como sea posible.
- Restar tantas monedas de 10 pesetas como sea posible.
- Restar tantas monedas de 5 pesetas como sea posible.

El proceso total

Examine el Programa 4.18.

Programa 4.18 #include <stdio.h>

```
main()
{
    int cantidad = 0, D5 = 0, D2 = 0, D1 = 0;
    char moneda;

    printf("Deposite moneda (1-2-5 duros) o pida articulo (P) ...\n");
    do
    {
        moneda = getchar(); /* Obtiene la respuesta del usuario */
        printf("\n");
        switch(moneda)      /* Realiza la selección */
        {
            case '5' : cantidad += 25; break;
            case '2' : cantidad += 10; break;
            case '1' : cantidad += 5; break;
            case 'P' : if (cantidad < 50)
                        printf("Por favor introduzca mas dinero ...\n");
        }
        if (moneda != 'P')
            printf("Cantidad = $%5.2f\n", cantidad/100.0);
    } while ((moneda != 'P') || (cantidad < 50));

    cantidad -= 50;          /* Compra el artículo */
    while (cantidad >= 25)    /* Cambio en monedas de 25 pts */
    {
        cantidad -= 25;
        D5++;
    }
    if (D5 != 0)
        printf("Monedas de 25 pts : %2d\n", D5);

    while (cantidad >= 10)    /* Cambio en monedas de 10 pts */
    {
        cantidad -= 10;
        D2++;
    }
}
```

```

    }
    if (D2 != 0)
        printf("Monedas de 10 pts : %2d\n",D2);

    while (cantidad > 5) /* Cambio en monedas de 5 pts */
    {
        cantidad -= 5;
        D1++;
    }
    if (D1 != 0)
        printf("Monedas de 5 pts : %2d\n",D1);
}

```

La primera parte del programa contiene un bucle `do` que acepta monedas del usuario. Se emplea una sentencia `switch()` para determinar cómo se incrementa la variable `cantidad`. Si se introduce 'P', la sentencia `if`:

```

case 'P' : if (cantidad < 50)
    printf("Por favor introduzca mas dinero...\n");

```

permite determinar si el usuario ha introducido suficiente dinero.

El bucle `do` se repite hasta que el usuario introduzca 'P' y la cantidad introducida sea al menos 50 pesetas. Esta condición se comprueba en la siguiente sentencia:

```

) while ((moneda != 'P') || (cantidad < 50));

```

Después de vender el producto, la variable `cantidad` se decrementa en 50. A continuación se utilizan tres bucles `while` para devolver el cambio. El primero determina el número de monedas de 25 pesetas necesarios para el cambio.

```

while (cantidad >= 25) /* Cambio en monedas de 25 pts */
{
    cantidad -= 25;
    D5++;
}
if (D5 != 0)
    printf("Monedas de 25 pts : %2d\n",D5);

```

Si `cantidad` es menor de 25, se salta el bucle y no se ejecuta. En caso contrario, a `cantidad` se le resta 25 hasta que sea menor que 25. El número de veces que es posible este decremento se indica mediante la variable `q`. Si `q` es 0 no se imprime nada.

El segundo y tercer bucle `while` realizan las mismas operaciones para monedas de 10 y 5 pesetas.

A continuación se muestra un ejemplo de ejecución del Programa 4.18.

```

Deposite moneda (1 2-5 duros) o pida articulo (P) ...
5
Cantidad = 25

```

```

1
Cantidad = 30
2
Cantidad = 40
P
Por favor introduzca mas dinero ...
2
Cantidad = 50
5
Cantidad = 75
1
Cantidad = 80
P
Monedas de 25 pts : 1
Monedas de 5 pts : 1

```

Conclusión

La máquina expendedora es un buen ejemplo de situación en que se utilizan bucles en el mundo real. Incluso aunque nuestra máquina expendedora no tiene límite de monedas con las que realizar el cambio, requiere el uso de bucles para devolver el cambio correcto al usuario. Para ser completamente útil, nuestra máquina debería conocer el número de monedas de cada tipo del que dispone. Se necesita prestar atención a esta reserva de monedas puesto que podría encontrarse vacía en un momento determinado, siendo necesario la introducción del precio exacto.

Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso de la sección.

Repaso de la Sección 4.7

1. ¿Cómo se convierte la cantidad introducida en pesetas dentro del bucle `do`?
2. ¿Qué ocurre si se introducen otros caracteres diferentes a 5, 2, 1 y P durante la ejecución?
3. Explique el razonamiento que hay detrás de la comprobación del bucle `while`
`((moneda != 'P') || (cantidad < 50))`
4. ¿Cuántas monedas se pueden introducir en la máquina?
5. Explique por qué no es posible que la variable `cantidad` sea negativa después de una venta.
6. Suponga que se entra en los bucles que realizan el cambio con la variable `cantidad` igual a 115 ¿Qué tipo de cambio se genera?
7. ¿Por qué es necesario utilizar `>=` en las comprobaciones de los bucles de cambio en vez de `>`?
8. ¿Es importante el orden en el que se sitúan los bucles que realizan el cambio? Justifique su respuesta.

4.8. PROGRAMAS DE APLICACIÓN ADICIONALES

Los tres primeros programas de esta sección muestran usos adicionales de las estructuras de bucles que se han estudiado en este capítulo. Un cuarto programa ofrece otro ejemplo de recursividad.

Los números de Fibonacci son el tema del Programa 4.19. Un número de Fibonacci es siempre igual a la suma de los dos números de Fibonacci anteriores. Cualquier lista de números de Fibonacci siempre comienza del mismo modo, con un 0 y un 1 como los dos primeros términos. El tercer término es igual a 0 más 1, es decir, 1. El cuarto término es igual a 1 más 1, o sea 2. Este concepto se ilustra en la Figura 4.7, que muestra una secuencia de Fibonacci de siete términos.

El Programa 4.19 utiliza un número introducido por el usuario para generar una secuencia de números de Fibonacci.

Programa 4.19 #include <stdio.h>

```
main()
{
    int n;                /* Número leído. */
    int anterior,nuevo;   /* Variables de la serie de Fibonacci. */
    int temp;             /* Almacenamiento temporal. */
    int j;                /* Contador de bucle */

    printf("Introduzca el numero de términos que se generaran => ");
    scanf("%i",&n);
    anterior = 0;
    nuevo = 1;
    printf("%4i %4i ",anterior,nuevo);
    for (j = 1; j <= n-2; j++)
    {
        temp = anterior + nuevo;
        anterior = nuevo;
        nuevo = temp;
        printf("%4i ",nuevo);
    }
}
```

Los dos últimos términos de la secuencia se almacenan siempre en las variables anterior y nuevo. Se utiliza una variable temporal (temp) cuando se necesita generar un nuevo término. A continuación se presenta una ejecución del Programa 4.19 que muestra los primeros 12 números de la serie de Fibonacci.

Introduzca el número de terminos que se generaran => 12
 0 1 1 2 3 5 8 13 21 34 55 89

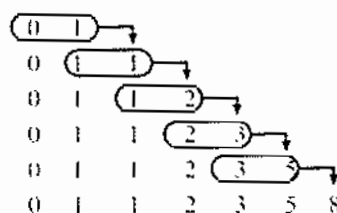


Figura 4.7. Serie de Fibonacci de siete términos.

¿Qué ocurre cuando el usuario introduce un número mayor que 30?
El Programa 4.20 muestra una simple máquina calculadora implementada en C.

```

Programa 4.20 #include <stdio.h>
               #include <stdlib.h>

               #define FALSO      0
               #define VERDADERO 1

main()
{
    int estado;    /* Estado actual de la máquina. */
    char car;      /* Carácter leído. */
    int err;       /* Indicador de error. */
    int op;        /* Operación matemática seleccionada. */
    int X,Y,Z;     /* Se usan en los cálculos. */

    printf("Introduzca una ecuacion de la forma:\n");
    printf(" X + Y = o X - Y =\n");
    printf("donde X e Y son enteros de un solo digito.\n");
    printf("=> ");
    estado = 1;
    err = FALSO;
    do
    {
        do        /* Saltar los espacios en blanco. */
        {
            car = getch();
            printf("%c",car);
        }
        while (car == ' ');
        switch(estado)
        {
            case 1 : if ((car >= '0') && (car <= '9'))
                        {
                            X = atoi(&car);
                            estado = 2;
                        }
                    else err = VERDADERO;
                    break;
            case 2 : op = 0;
                    if (car == '+') op = 1;
                    if (car == '-') op = 2;
                    if (op == 0) err = VERDADERO;
                    estado = 3;
                    break;
            case 3 : if ((car >= '0') && (car <= '9'))
                        {
                            Y = atoi(&car);

```

```

        estado = 4;
    }
    else err = VERDADERO;
    break;
case 4 : if (car == '=') estado = 5;
        else err = VERDADERO;
    }
}
while ((estado != 5) && !err);
if (!err)
{
    Z = (op == 1) ? X + Y : X - Y;
    printf(" %i", Z);
}
else printf("\nCaracter ilegal en la entrada de datos.");
}

```

La calculadora realiza únicamente sumas y restas de enteros positivos de un solo dígito. Se utiliza una nueva función para obtener la entrada del usuario. Esta función es `getchar()`, que se define en el archivo de cabecera `<stdlib.h>`. `getchar()` espera la introducción de un único carácter desde el teclado y devuelve el carácter ASCII asociado sin mostrarlo.

Otra nueva función, `atoi()`, se utiliza para convertir el carácter ASCII a un valor entero numérico. Esta función también es parte de `<stdlib.h>`.

La Figura 4.8 muestra el *diagrama de transición de estados* de la máquina calculadora. El objetivo del programa es alcanzar el estado 5 en el que se muestra la suma o diferencia de dos números. Si se introduce un carácter ilegal en cualquier estado, se

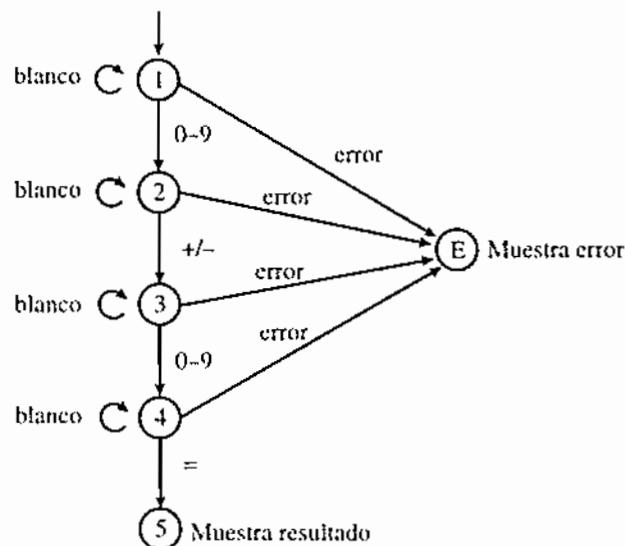


Figura 4.8. Diagrama de transición de estados para una máquina de calcular.

produce una condición de error. Se permite la introducción de múltiples blancos entre los números. Un ejemplo de posibles ejecuciones es el siguiente:

```
Introduzca una ecuación de la forma:
X + Y = o X - Y =
donde X e Y son enteros de un solo dígito.
=> 7 + 3 = 10
```

```
Introduzca una ecuación de la forma:
X + Y = o X - Y =
donde X e Y son enteros de un solo dígito.
=> 4 - 6 = -2
```

Observe que aunque el programa no permite la introducción de números enteros negativos, éstos pueden mostrarse en los resultados.

El siguiente programa de esta sección realiza una tarea simple muy útil, la *enumeración*. Enumerar significa listar todos los miembros que pertenecen a un conjunto predefinido. Por ejemplo, si el conjunto dado contiene X e Y, se pueden enumerar cuatro cadenas de dos caracteres diferentes: XX, XY, YX e YY. El Programa 4.21 ilustra como se pueden emplear bucles anidados para enumerar cadenas de tres caracteres.

```
Programa 4.21 main()
{
    char i,j,k;      /* Contadores de los bucles */
    int num = 0;     /* Contador global.          */

    for (i = 'a'; i <= 'c'; i++)
        for (j = 'a'; j <= 'c'; j++)
            for (k = 'a'; k <= 'c'; k++)
            {
                num++;
                printf("%c%c%c ",i,j,k);
            }
    printf("\nHay %i cadenas de caracteres diferentes.\n",num);
}
```

La ejecución del Programa 4.21 produce el siguiente resultado:

```
aaa aab aac aba abb abc aca acb acc baa bab bac
bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc
cca ccb ccc
Hay 27 cadenas de caracteres diferentes.
```

El Programa 4.21 también cuenta el número de cadenas de caracteres enumeradas. Con un simple formato, es posible ver el patrón que emerge de las cadenas generadas.

El último programa de esta sección emplea la recursividad para determinar un número entre 1 y 1024 elegido por el usuario. Como se muestra en el Programa 4.22, la recursividad directa se utiliza cuando `adivinar()` se llama sí misma.

```

Programa 4.22 #include <stdio.h>
#define MAXNUM 1024      /* Número a adivinar entre 1 y 1024 */

int adivinar(int inferior, int superior);

main()
{
    int intentos;

    printf("Por favor piense un numero del 1 al %4d...\n",MAXNUM);
    intentos = adivinar(1,MAXNUM);
    printf("\nHe necesitado %2d intentos.\n",intentos);
}

int adivinar(int inferior, int superior)
{
    int prediccion;
    char respuesta;

    prediccion = inferior + (superior-inferior)/2;
    printf("\nEs su numero %4d? ",prediccion);
    respuesta = getch();
    switch(respuesta)
    {
        case 'S' : return 1; break;
        case 'm' : return 1 + adivinar(inferior,prediccion-1); break;
        case 'M' : return 1 + adivinar(prediccion+1,superior); break;
    }
}

```

Cada nueva llamada a `adivinar()`, reduce el intervalo de números que deben considerarse a la mitad. Por ejemplo, suponga que el número escogido es 907. Una ejecución del Programa 4.22 sería la siguiente:

```

Por favor piense un numero del 1 al 1024...
Es su numero 512? M
Es su numero 768? M
Es su numero 896? M
Es su numero 960? m
Es su numero 928? m
Es su numero 912? m
Es su numero 904? M
Es su numero 908? m
Es su numero 906? M
Es su numero 907? S
He necesitado 10 intentos.

```

El usuario debe contestar adecuadamente con una 'M' (si el número es mayor), una 'm' (si es menor) o una 'S' (si es el número indicado). El programa siempre escoge el número que se encuentra en la mitad del intervalo donde pertenece el número a adivinar. Esto es por lo que el primer intento es el 512. Debido a que el usuario

contesta con una 'M' al primer intento, el siguiente intervalo de búsqueda va del 513 al 1024. El segundo intento es 768 y de nuevo es incorrecto, escogiendo de nuevo el intervalo superior para la siguiente búsqueda, y así sucesivamente.

Repaso de la Sección 4.8

1. ¿Pueden generarse los números de Fibonacci sin emplearse un bucle?
2. ¿Por qué es necesario asignar a la variable estado el valor inicial 1 en el Programa 4.20?
3. ¿Cómo puede emplearse la recursividad para enumerar cadenas de tres caracteres?
4. Demuestre que el número máximo de intentos en el Programa 4.22 es 10.

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido para permitir adquirir una valiosa experiencia y, lo que es más importante, para tener una realimentación inmediata de lo que los conceptos y mandatos presentados en este capítulo hacen. Además son divertidos.

Ejercicios

1. Prediga que hará el Programa 4.23 y luego pruébelo.

Programa 4.23 #include <stdio.h>

```
main()
{
    char c; /* Un carácter. */
    for(c = 'a'; c <= 'z'; c++)
        printf(" c = %c",c);
}
```

2. El Programa 4.24 muestra el conjunto de caracteres ASCII, presentando el valor ASCII así como el carácter correspondiente.

Programa 4.24 #include <stdio.h>

```
main()
{
    int i; /* Un número. */
    int j; /* Otro número. */
    j = 0;
    for(i = 0; i <= 255; i++)
    {
        printf(" %d = %c",j,i);
```

```

    }
    printf("Fin\n");
}

```

3. En el bucle `for` del Programa 4.25, el contador del bucle tiene un `++` en vez de un `--`. ¿Qué diferencia supone que existirá en el resultado según se utilice uno u otro? Pruébalo.

Programa 4.25 `#include <stdio.h>`

```

main()
{
    int i;      /* Un número. */

    for(i = 0; i <= 5; ++i)
        printf(" i = %d", i);
    printf(" Ultimo valor de i = %d", i);
}

```

4. El Programa 4.26 contiene un «`while`» vacío. Esto significa que no existe ningún argumento en la sentencia `while`. ¿Qué cree que ocurrirá cuando intente compilar este programa? Verifique su conclusión.

Programa 4.26 `#include <stdio.h>`

```

main()
{
    while(1)
        printf("Se ejecuta este programa\n");
}

```

5. ¿Cuántas veces se ejecuta el bucle del Programa 4.27? Pruébalo. ¿Qué provoca la parada del bucle?

Programa 4.27 `#include <stdio.h>`

```

main()
{
    int i;      /* Un número. */

    i = 10;
    while(1)
    {
        printf("Cuántas veces se ejecuta este bucle?\n");
        printf(" i = %d\n", i--);
    }
}

```

6. ¿Qué imprime el Programa 4.28?

Programa 4.28

```
#include <stdio.h>

int potencia(int exp);

main()
{
    printf("%d", potencia(8));
}

int potencia(int exp)
{
    if (exp == 0)
        return (1);
    else
        return(2 * potencia(exp - 1));
}
```

Autoevaluación

DIRECTRICES

Utilice el Programa 4.18 para responder a las siguientes cuestiones.

Preguntas

1. ¿A cuántas variables se les asigna valor inicial?
2. ¿Cómo puede añadirse a la máquina la posibilidad de devolver una moneda (utilizando la tecla 'R')?
3. ¿Qué modificaciones se requieren para poder considerar reservas de monedas?
4. ¿Pueden sustituirse los bucles que realizan el cambio, por una o más sentencias que directamente calculen el cambio (utilizando posiblemente una división)?
5. ¿Porqué se necesita la condición (moneda != 'P') antes de la segunda sentencia `printf` del bucle `do`?

Problemas de fin de capítulo

Conceptos generales

Sección 4.1

1. ¿Qué es una sentencia compuesta? ¿Cómo se indica en un programa C?
2. Indique la notación empleada en C para pre-incrementar la variable contador.
3. En C, ¿qué le permite tener dos sentencias secuenciales?
4. ¿Cuáles son las partes principales del bucle `for` de C?

Sección 4.2

5. Indique la construcción del bucle `while` de C.
6. ¿Cuándo se comprueba la condición en un bucle `while` de C?
7. Indique la condición bajo la que se repite un bucle `while` de C.
8. ¿Cuál es un buen uso del bucle `while` de C?

Sección 4.3

9. Indique la construcción del bucle `do while` de C.
10. ¿Cuándo se comprueba la condición de un bucle `do while` de C?

11. ¿Cuáles son las condiciones para repetir un bucle `do while` de C?
12. Explique qué bucle es más recomendable, el `while` o el `do while`.

Sección 4.4

13. Indique el significado de un bucle anidado.
14. ¿Qué bucles de C pueden anidarse?
15. Explique si existe algún problema con el anidamiento de un bucle `do` de C.

Sección 4.5

16. Explique lo que se entiende por un error en tiempo de ejecución.
17. ¿Qué es una función de depuración? ¿Cómo se utiliza?
18. ¿Cómo se utiliza una función de auto-depuración?

Sección 4.6

19. ¿Qué es necesario en la recursividad?
20. ¿Cuál es la diferencia existente entre recursividad directa e indirecta?
21. ¿Por qué es necesario parar la recursividad en un nivel determinado?
22. ¿Cómo simula la recursividad la operación de un bucle?

Diseño de programas

Para cada programa, documente su diseño y manténgalo con su programa. Este proceso debería incluir el contenido del diseño que indique el propósito del programa, la entrada que se necesita, el proceso que se realiza sobre la entrada y la salida que se necesita, así como el algoritmo del programa. Asegúrese de que incluye toda esta documentación en su programa final. Ésta debería contener, al menos, el bloque del programador, los prototipos de las funciones y una descripción de cada función así como los parámetros formales que se utilizan.

23. Desarrolle un programa C que muestre el interés compuesto anual desde 1 a 30 años. El usuario debe introducir el capital invertido y la tasa de interés. La relación matemática es la siguiente:

$$Y = A (1 + N)^T$$

donde

Y = La cantidad.

A = El capital invertido.

N = La tasa de interés.

T = El número de años.

24. Desarrolle un programa en C que eleve un número a cualquier potencia. El usuario debe introducir la base y la potencia.
25. Escriba un programa C que muestre sólo un número específico de la serie de Fibonacci. El usuario especificará el término requerido.
26. Modifique el Programa 4.20 para permitir multiplicación y división.
27. Modifique el Programa 4.20 para permitir enteros negativos y positivos de uno y dos dígitos.
28. Escriba un programa en C que acepte sólo los dígitos binarios 0 y 1, hasta que se introduzca la letra 'b' o 'B'. El programa calculará el número decimal equivalente. Haga uso de la función `getch()` en su programa.
29. Escriba un programa C que compruebe si un número introducido por el usuario es un número real válido. Ejemplos son 1.7, -0.0354, 259.08 y -1067.25. Utilice la función `getch()` en su programa.
30. Modifique el Programa 4.18 para permitir elegir tres tipos de productos cuyo precio es el siguiente:

Producto 1: 100 pts.

Producto 2: 130 pts.

Producto 3: 250 pts.

31. Escriba una función recursiva que calcule el resto de una división. Por ejemplo, una llamada `resdiv(100, 12)` devolverá el valor 4 (debido a que 100 dividido por 12 es igual a 8 con resto 4).
32. Desarrolle un programa en C que calcule y muestre los valores de un intervalo de temperatura en grados Fahrenheit y grados centígrados. El usuario seleccionará la temperatura más baja y más alta así como el incremento de temperatura. La relación matemática es:

$$F = (9/5)C + 32$$

donde

F = Temperatura en grados Fahrenheit.

C = Temperatura en grados centígrados.

Punteros¹, ámbito y clases de variables

Objetivos

En este capítulo podrá aprender:

1. Cómo se almacena la información en la memoria del computador.
 2. De qué diferentes formas usa la memoria el lenguaje C.
 3. Los métodos que se utilizan para almacenar valores en la memoria.
 4. Cómo se representan en la memoria los números negativos.
 5. Qué son los punteros y cómo se usan.
 6. La manera de pasar variables usando punteros.
 7. La visibilidad de constantes y variables desde las distintas partes del programa.
 8. Las diferentes clases de variables.
 9. Algunos errores de programación comunes.
-

Palabras clave

Dirección	Declaración de puntero
Ciclo de búsqueda/ejecución	Operador de indirección
Direccionamiento inmediato	Variable local
Direccionamiento directo	Ámbito de una variable
Octeto (<i>byte</i>)	Vida de una variable
Palabra	Variable global
Complemento a 1	Variable automática
Complemento a 2	Variable externa
Cuarteto (<i>nibble</i>)	Variable estática
Puntero	Variable de registro
Operador de dirección	Dar valor inicial

¹ En Latinoamérica se utiliza el término *apuntador* (N. del R. T.)

Contenido

5.1. Organización de la memoria	5.6. Clases de variables
5.2. Cómo se utiliza la memoria	5.7. Depuración e implementación de programas
5.3. Punteros (apuntadores)	5.8. Programas de aplicación
5.4. Paso de variables	
5.5. El ámbito de las variables	
-

Introducción

Este capítulo presenta un concepto importante del lenguaje C: los **punteros (apuntadores)**. Como se podrá ver en el mismo, se trata simplemente de otra forma de poder acceder a los datos almacenados en la memoria del computador. Este nuevo concepto proporciona a los programadores un mecanismo muy poderoso y flexible para manejar los datos del programa.

Para la correcta comprensión de este capítulo, se necesita tener un conocimiento previo sobre la representación de los números en los sistemas binarios, octal y hexadecimal.

En la primera parte del capítulo se presentará brevemente la estructura interna del computador mostrando la organización de la memoria del mismo. Este conocimiento previo facilitará el entendimiento del concepto de puntero.

5.1. ORGANIZACIÓN DE LA MEMORIA

Presentación

En esta sección se explica cómo se almacena la información dentro del computador, lo que servirá de base para el material presentado en el resto del capítulo.

Idea básica

En la Figura 5.1 se representa el modelo básico de la estructura de la memoria. Como se observa en la figura, la memoria puede entenderse como una pila de posiciones de almacenamiento, donde cada posición tiene asignado un número diferente denominado dirección. Por lo tanto, cada posición de memoria del computador tiene una dirección única.

Almacenamiento de un programa

La Figura 5.2 intenta mostrar cómo se almacena en la memoria un hipotético programa que realice la suma de dos números almacenados en memoria y guarde el resultado de la operación en otra posición.

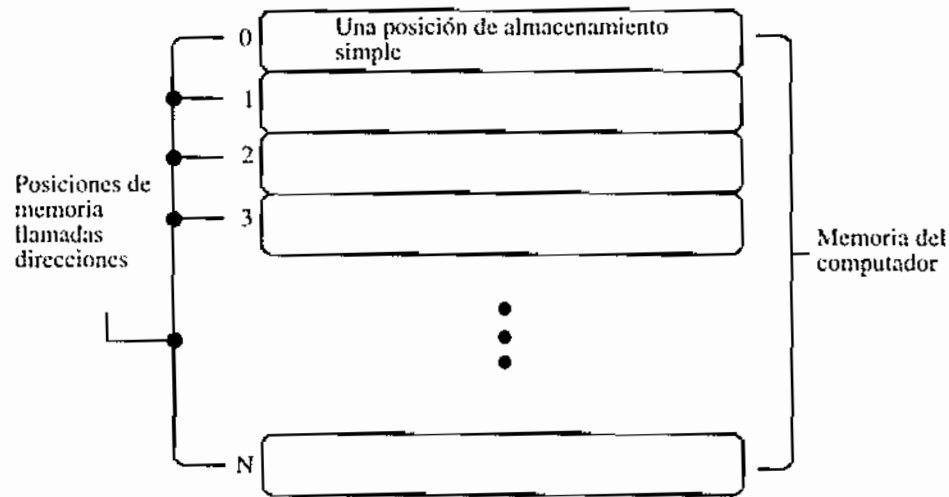


Figura 5.1. Visualización de la memoria del computador.

En la figura se observan siete posiciones de memoria, identificadas por su dirección. Cada posición puede contener una instrucción (una acción que debe realizar el computador) o un dato (un elemento sobre el que actúa una instrucción). Cuando se ejecuta este programa, la UCP (unidad central de proceso) comienza leyendo la información de la posición de memoria 1. La instrucción almacenada en dicha posición indica a la UCP que lea el número almacenado en la siguiente posición (dirección 2). Una vez leído, la UCP obtiene de la siguiente posición (dirección 3) la próxima instrucción. Este proceso repetitivo de lectura y ejecución de instrucciones se denomina el **ciclo de lectura/ejecución** de un computador.

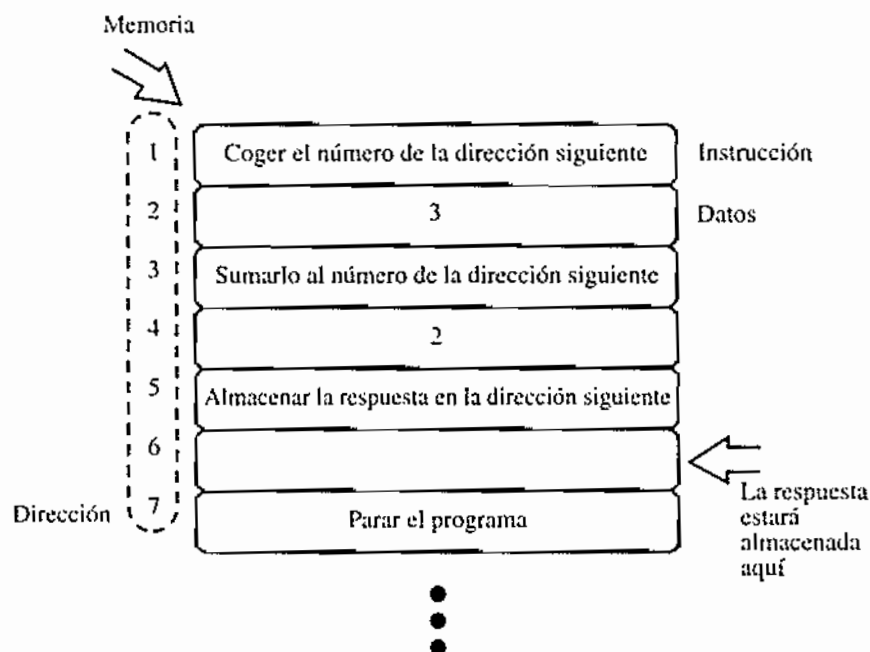


Figura 5.2. Concepto de programa en memoria.

La UCP sumará dos números (3 y 2) y almacenará el resultado (5) en la posición de memoria 6. Las direcciones se usan para distinguir las diferentes posiciones de memoria.

Una forma de almacenamiento alternativa

No es una práctica común almacenar los datos de un programa cerca de las instrucciones. Así, por ejemplo, la Figura 5.3 muestra cómo se almacena en memoria un programa que realiza la misma tarea que el programa anterior (la suma de dos números) pero de forma diferente.

El programa de la Figura 5.3 ejecuta las siguientes instrucciones:

1. Obtener el número almacenado en una determinada dirección: La UCP lee la siguiente posición de memoria (dirección 2) para conseguir la dirección donde está almacenado el número (dirección 8). A continuación, lee el número almacenado en la dirección 8 (el número 3) y con esto terminará esta instrucción que ocupa dos posiciones. La UCP calcula la dirección de la siguiente instrucción (dirección 3) y la lee.
2. Sumarle el número almacenado en una determinada dirección: Como antes, la instrucción ocupa dos posiciones. La UCP tiene que leer la siguiente dirección (dirección 4) para averiguar la dirección donde está almacenado el segundo sumando (dirección 9). Seguidamente, lee el valor almacenado en la dirección 9 (un 5) y lo suma al anterior. La UCP lee la siguiente instrucción (dirección 5).

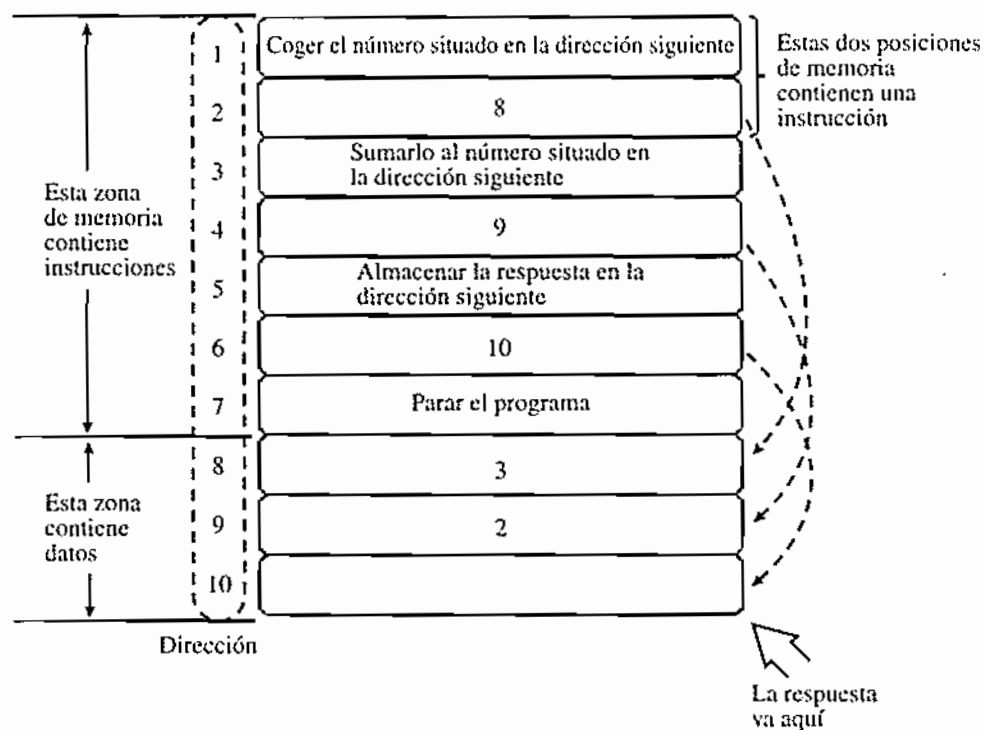


Figura 5.3. Una forma diferente de hacer lo mismo.

3. Almacenar el resultado en una determinada dirección: La UCP lee la siguiente posición (dirección 6) para obtener la dirección donde se almacenará el resultado (dirección 10). La instrucción termina almacenando en la dirección 10 el resultado de la suma (8). Por último, la UCP lee la siguiente instrucción (dirección 7).
4. Parar el programa: La UCP deja de procesar instrucciones.

Aunque pueda parecer un proceso un poco extraño, la mayoría de los computadores actuales procesan las instrucciones y los datos de esta forma. Para poder usar muchas de las facilidades proporcionadas por el lenguaje C, es importante comprender estos aspectos.

El primer método (representado por el programa de la Figura 5.2), donde los datos están justo después de cada instrucción, se denomina **direccionamiento inmediato**. El segundo (representado por el programa de la Figura 5.3), donde cada instrucción contiene la dirección del dato, se denomina **direccionamiento directo**.

Una de las diferencias entre ambos métodos es el número de posiciones de memoria usadas: 7 en el primer caso y 10 en el segundo.

Ejemplo 5.1 ¿Qué hace el programa de la Figura 5.4?

Solución

Este programa multiplica $5 \times 12 = 60$ y almacena el resultado en la posición 3445.

3438	Coger el número situado en la dirección siguiente:
3439	3447
3440	Multiplicarlo por el número de la dirección siguiente:
3441	3446
3442	Almacenar la respuesta en la dirección siguiente:
3443	3445
3444	Parar el programa
3445	
3446	12
3447	5

Figura 5.4. Programa para el Ejemplo 5.1.

Conclusión

Los aspectos presentados en esta sección proporcionarán una base teórica que permita comprender cómo se almacenan las instrucciones y los datos en la memoria del computador y entender la diferencia entre el direccionamiento inmediato y el directo. Compruebe si ha comprendido correctamente los conceptos presentados en la sección usando el siguiente repaso.

Repaso de la Sección 5.1

1. ¿Cómo puede representarse gráficamente la memoria de un computador?
2. Explique la diferencia entre una instrucción y un dato.
3. ¿Qué es una dirección?
4. Explique en qué consiste el direccionamiento inmediato.
5. Explique en qué consiste el direccionamiento directo.

5.2. CÓMO SE UTILIZA LA MEMORIA

Presentación

Para poder beneficiarse de toda la potencia que proporciona el lenguaje C, es necesario comprender cómo se organizan los datos. La información que se presenta en esta sección utiliza números representados en binario, octal y hexadecimal.

Tamaño de almacenamiento

La Figura 5.5 muestra cómo se puede clasificar la información almacenada en la memoria dependiendo del tamaño que ocupe. No se puede perder de vista que realmente todas las instrucciones y datos se representan mediante elementos electrónicos con dos estados: activado o desactivado.

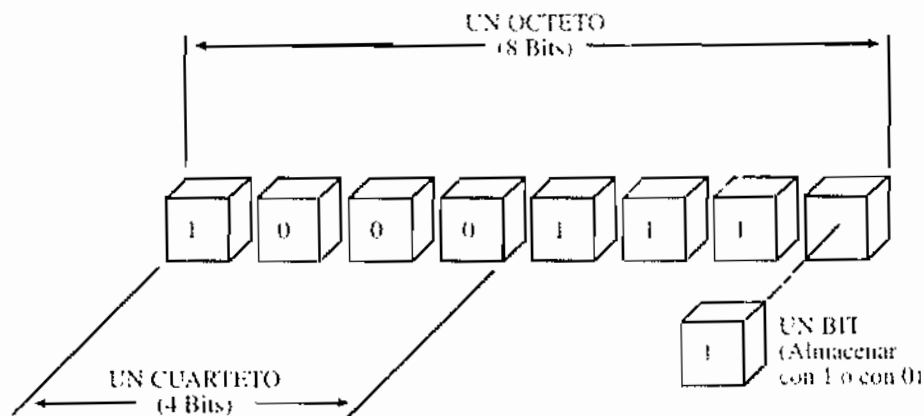


Figura 5.5. Clasificación del almacenamiento.

La memoria del computador se organiza usualmente en grupos de **octetos** (bytes) denominados **palabras**. Los tamaños de palabras más comúnmente utilizados son 8 bits (1 octeto), 16 bits (2 octetos), 32 bits (4 octetos) y 64 bits (8 octetos). La Figura 5.6 muestra un ejemplo donde la memoria tiene una palabra de 8 bits.

Tipo char

En C, el tipo `char` ocupa un único octeto. Por tanto, el intervalo de valores que se puede almacenar en este tipo va desde 00000000 a 11111111 en binario, que se corresponde en hexadecimal con el intervalo del 00 al FF y en decimal del 0 al 255. La Figura 5.7 ilustra este concepto.

El Programa 5.1 muestra los límites de este tipo de datos. El máximo valor representable es 255 y si se incrementa en uno este valor, se obtiene un 0.

Programa 5.1

```
#include <stdio.h>

main()
{
    unsigned char contador;

    contador = 0;
    while(contador != 255)
    {
        contador++;
        printf(" %d-%X ", contador, contador);
    }
}
```

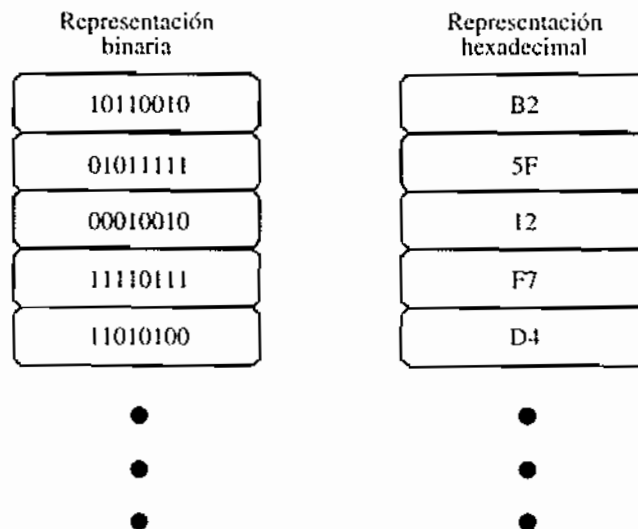


Figura 5.6. Organización de la memoria con palabras de 8-bits.

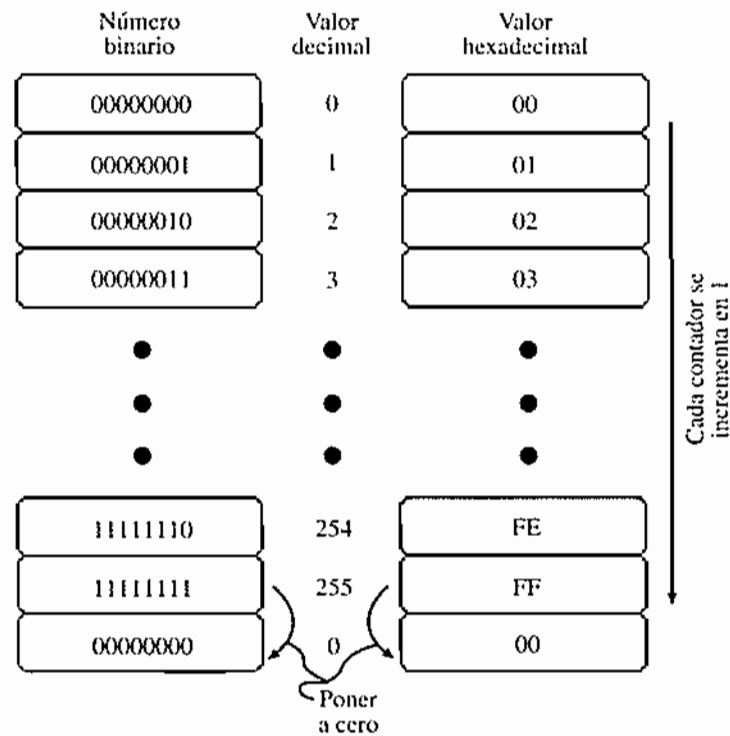


Figura 5.7. Contando en un octeto de memoria.

Cuando se ejecuta el Programa 5.1 se produce la siguiente salida:

```
1-1 2-2 ... 10-A 11-B ... 254-FE 255-FF
```

El programa genera todos los números representables con este tipo (de 0 a 255). Si una vez alcanzado el valor máximo 11111111, se incrementa en uno dicho valor se obtendrá el valor 00000000 (igual que le sucede al cuentakilómetros de un coche cuando alcanza el valor máximo). Obsérvese que se usa el tipo `unsigned char`. El motivo de ello se comprenderá en un breve plazo.

La función `sizeof`

La función `sizeof` devuelve el tamaño en octetos que ocupa una variable en memoria. También se puede aplicar a un tipo de datos, en cuyo caso devuelve cuánto ocuparán las variables de ese tipo. El Programa 5.2 ilustra el uso de esta función.

Programa 5.2 `#include <stdio.h>`

```
main()
{
    char caracter;
    int entero;
```



```

printf("El tamaño de un caracter en su sistema es %d octeto.\n",
      sizeof(caracter));
printf("El tamaño de un entero en su sistema es %d octetos.\n",
      sizeof(entero));
}

```

Cuando se ejecuta este programa, produce la siguiente salida:

```

El tamaño de un caracter en su sistema es 1 octeto.
El tamaño de un entero en su sistema es 2 octetos.

```

Representación de números con signo

Como el lector ya conoce, el computador puede trabajar tanto con números positivos como negativos. Sin embargo, puesto que los números se representan únicamente con unos y ceros, se deben usar también estos dos valores para distinguir los números positivos de los negativos.

Hay varios métodos para hacer esto. El método más común requiere conocer cómo se realiza la adición de números binarios.

Adición binaria

Para sumar números binarios, se necesitan conocer las cinco reglas siguientes:

				1
0	1	0	1	1
$\frac{+0}{0}$	$\frac{+0}{1}$	$\frac{+1}{1}$	$\frac{+1}{10}$	$\frac{+1}{11}$

Como ejemplo, la suma en binario de $2 + 3$ sería:

$$\begin{array}{r}
 1 \quad \leftarrow \text{Acarreo} \\
 10 \\
 +11 \\
 \hline
 101 = 5
 \end{array}$$

Un segundo ejemplo, la suma de $7 + 3$:

$$\begin{array}{r}
 111 \quad \leftarrow \text{Acarreo} \\
 111 \\
 +011 \\
 \hline
 1010 = 10
 \end{array}$$

Es posible realizar la sustracción binaria de una forma similar. Sin embargo, esto requeriría el uso de circuitos lógicos complejos. Por ello, la sustracción se realizará realmente mediante la adición. Para comprender cómo se realiza esta operación, se necesita conocer primero cómo se representan los números negativos.

El complemento de un número

En terminología informática el complemento de un número en binario se calcula cambiando todos sus unos por ceros y los ceros por unos. Por ejemplo, el complemento de 1011 es 0100. Este proceso se denomina **complemento a uno**.

Los números negativos se representan realmente mediante la notación denominada **complemento a dos**. Para calcular el complemento a 2 de un número binario, se debe hacer primero el complemento a 1 y luego sumar un uno al resultado. Por ejemplo, para calcular el complemento a 2 de 1011 se seguirán los siguientes pasos:

1. Se calcula el complemento a 1: 0100
2. Se suma 1 al resultado: 0101

Sustracción binaria

Para restar usando la notación complemento a 2, se deben llevar a cabo los pasos que se describen a continuación (utilizando como ejemplo $7 - 5$):

1. Calcular el complemento a 2 del sustraendo: 1011
2. Sumar ambas cantidades y olvidar el acarreo final: $0111 + 1011 = 0010$

El resultado es el previsto: 2 en base decimal. En efecto, lo que se ha hecho es convertir -5 a notación complemento a 2 causando que el bit más significativo (BMS) del número sea un 1 (1011). Por lo tanto, cuando se manejan números binarios con signo representados en complemento a 2, si el BMS es un 1 el número es negativo y su valor corresponde con el complemento a 2 del número real. Así, por ejemplo, el número 1011 es un número negativo cuyo valor puede obtenerse calculando su complemento a 2 (el complemento a 2 de 1011 es 0101, por lo que representa el valor -5).

En la Tabla 5.1 se representan todas las posibles combinaciones binarias de un **cuarteto**, junto con los valores correspondientes cuando se interpretan como números sin signo y como números con signo representados en complemento a 2.

Como se aprecia en la tabla, un cuarteto en notación sin signo representa un intervalo de valores del 0 al 15 (16 valores diferentes). Sin embargo, en notación con signo, representa un intervalo del -8 al $+7$ (también 16 valores diferentes). Por lo tanto, el intervalo de valores representables utilizando números con signo en notación complemento a 2 será:

$$\text{Intervalo} \Rightarrow \text{desde } -(2^{N-1}) \text{ a } +(2^{N-1}-1)$$

Tabla 5.1. Cuartetos con y sin signo

Valor sin signo	Binario	Valor con signo
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	-8
9	1001	-7
10	1010	-6
11	1011	-5
12	1100	-4
13	1101	-3
14	1110	-2
15	1111	-1

Ejemplo 5.2 Determina el intervalo de valores representables en notación complemento a 2 usando un octeto.

Solución

Intervalo $\Rightarrow -(2^7)$ a $(2^7 - 1) = -128$ a $+127$

Como muestra el ejemplo, un octeto en notación con signo representa un intervalo del -128 al 127 , mientras que usando notación sin signo representa un intervalo del 0 al 255 . El Programa 5.3 ilustra esta diferencia.

Programa 5.3 #include <stdio.h>

```

main()
{
    char valor;

    printf("Deme un numero nexadecimal de dos digitos => ");
    scanf("%X",&valor);
    printf("El valor del numero en decimal con signo es %d\n",valor);
    printf("El valor del numero en decimal sin signo es %u\n",valor);
}

```

La primera parte del programa trata el valor hexadecimal introducido por el usuario como número con signo en notación complemento a 2. La segunda, sin embargo, lo considera como un número sin signo. Para ello debe declarar la variable del tipo `unsigned char`.

Cuando se ejecuta el programa y el usuario introduce un valor hexadecimal menor o igual que 7F (o sea, con el BMS igual a 0), la primera parte del programa lo considera como un número positivo. Así, en este caso, tanto si se interpreta el número con signo como sin signo, ambas partes producen la misma salida:

```
Deme un numero hexadecimal de dos digitos => 7F
El valor del numero en decimal con signo es 127
El valor del numero en decimal sin signo es 127
```

Si el usuario introduce un valor hexadecimal mayor o igual que 80 (o sea, con el BMS igual a 1), la primera parte del programa lo considera negativo. Con el valor 80 se producirá la siguiente salida:

```
Deme un numero hexadecimal de dos digitos => 80
El valor del numero en decimal con signo es -128
El valor del numero en decimal sin signo es 128
```

Cuando el usuario introduce el valor FF, el programa imprimirá:

```
Deme un numero hexadecimal de dos digitos => FF
El valor del numero en decimal con signo es -1
El valor del numero en decimal sin signo es 255
```

Este programa ha mostrado la diferencia entre los tipos con signo y los tipos sin signo. La Tabla 5.2 muestra los intervalos de representación de los distintos tipos de datos, considerados con signo y sin signo. El Programa 5.4 muestra los tamaños de los distintos tipos de lenguaje.

Programa 5.4

```
main()
{
    printf("El tamaño en octetos de un:\n");
    printf("int => %d\n",sizeof(int));
    printf("char => %d\n",sizeof(char));
    printf("short => %d\n",sizeof(short));
    printf("long => %d\n",sizeof(long));
    printf("unsigned char => %d\n",sizeof(unsigned char));
    printf("unsigned int => %d\n",sizeof(unsigned int));
    printf("unsigned long => %d\n",sizeof(unsigned long));
    printf("float => %d\n",sizeof(float));
    printf("double => %d\n",sizeof(double));
    printf("long double => %d\n",sizeof(long double));
}
```

Tabla 5.2. Rango de valores para los tipos de C

Identificado como	También llamado	Rango de valores (IBM PC)
char	signed char	-128 a 127
unsigned char		0 a 255
int	signed int	-32.768 a 32.767
unsigned int	unsigned	0 a 65.535
short	signed short	-32.768 a 32.767
unsigned short	unsigned short int	0 a 65.535
long	long int, signed long	-2.147.483.648 a 2.147.483.647
unsigned long	unsigned long int	0 a 4.294.967.295
enum	enumerado	0 a 65.535
float		+/-3,4E+/-38 (7 dígitos)
double		+/-1,7E+/-308 (15 dígitos)
long double		+3,4E-4932 a 1,1E+4932

Cuando se ejecuta, este programa muestra el tamaño de los distintos tipos de datos. Este tamaño dependerá de la máquina donde se ejecuta el programa. En el caso de un PC de IBM típicamente se obtendrían los siguientes valores:

El tamaño en octetos de un:

```
int => 2
char => 1
short => 2
long => 4
unsigned char => 1
unsigned int => 2
unsigned long => 4
float => 4
double => 8
long double => 10
```

Conclusión

En esta sección se ha mostrado cómo el lenguaje C usa la memoria del computador. Asimismo, se ha explicado cómo se representan los números negativos y cómo se trabaja con ellos. Compruebe su comprensión de los conceptos presentados usando el siguiente repaso.

Repaso de la Sección 5.2

1. ¿Qué representa el término palabra en un computador?
2. ¿Cuál es el tamaño del tipo char?
3. ¿Cómo se representan los números con signo?
4. ¿Qué diferencia hay entre un tipo de datos con signo y uno sin signo?
5. ¿Qué tipo de datos ocupa menos memoria? ¿Cuál ocupa más?

5.3. PUNTEROS

Presentación

En esta sección se mostrarán los secretos de un concepto de programación muy importante: los **punteros (apuntadores)**. Para comprender adecuadamente este nuevo concepto, el lector deberá dedicar un buen rato a esta sección trabajando con los ejemplos presentados.

Idea básica

Imagínese la memoria del computador como se muestra en la Figura 5.8. Cada posición de memoria puede almacenar un octeto de información y está identificada por un número denominado dirección. Las direcciones de memoria se enumeran secuencialmente desde cero hasta el tamaño de la memoria.

El Programa 5.5 ilustra varios aspectos del uso de la memoria y de los punteros.

```

Programa 5.5 #include <stdio.h>

main()
{
    char valor;      /* Una posición de memoria para guardar un
                       carácter. */

    char *puntero;   /* Un puntero. */

    valor = 97;
    printf("%u => | %d | <= direccion y datos de valor.\n",
           &valor, valor);
    puntero = &valor;
    printf("%u => | %d | <= direccion y datos de puntero.\n",
           &puntero, puntero);
    printf("\n Valor almacenado en puntero = %d\n", puntero);
    printf(" Direccion de puntero : &puntero = %u\n", &puntero);
    printf(" Valor referenciado por puntero: *puntero = %d\n",
           *puntero);
}

```

Análisis del programa

La ejecución del Programa 5.5 produce la siguiente salida:

```

1204 => | 97 | <= direccion y datos de valor.
4562 => | 1204 | <= direccion y datos de puntero.

```

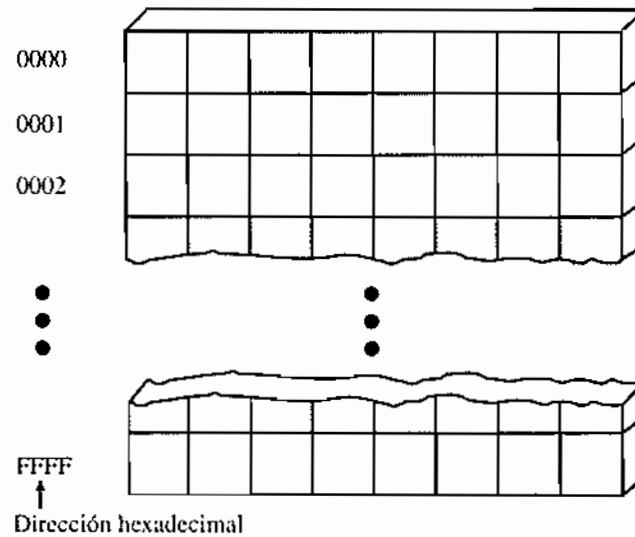


Figura 5.8. Construcción de una memoria con tamaño-octeto.

```
Valor almacenado en puntero = 1204
Dirección de puntero : &puntero = 4562
Valor referenciado por puntero: *puntero = 97
```

La primera sentencia simplemente almacena el valor en una posición de memoria:

```
valor = 97;
```

Se conoce el valor almacenado pero, si se precisa conocer la dirección donde se almacenó, es necesario usar el **operador de dirección &**. Cuando se aplica este operador a una variable, devuelve la dirección de memoria asociada a esa variable.

La siguiente sentencia usa la variable puntero que se declaró al principio del programa de la siguiente forma:

```
char *puntero; /* Un puntero. */
```

Se trata de una **declaración del puntero** que especifica que la variable declarada como puntero mantendrá una dirección de memoria que apuntará a un determinado tipo de objeto (en este caso, de tipo char). La siguiente sentencia almacena la dirección de la variable valor en el puntero.

```
puntero = &valor;
```

Ahora la variable puntero contiene la dirección de la variable valor como muestra la Figura 5.9 (Nota: se puede declarar un puntero que apunte a un tipo void. Hacer esto retrasa la especificación del tipo al que apunta dicho puntero. Cuando se usa el puntero, hay que especificar el tipo usando el mecanismo de conversión de tipos).

Seguidamente el programa accede al valor almacenado en la variable valor a través de la variable puntero usando el **operador de dirección ***.

```
printf(" Valor referenciado por puntero: *puntero = %d\n", *puntero);
```

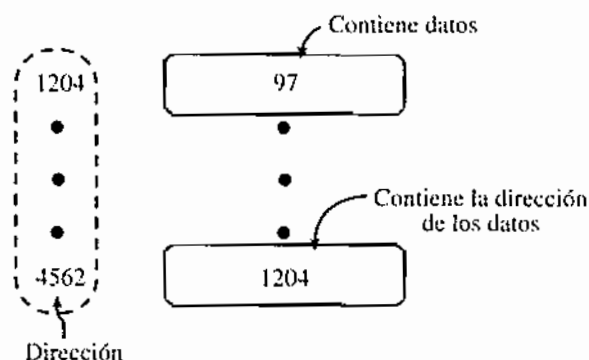


Figura 5.9. Idea de puntero y valor.

La Figura 5.10 ilustra lo que sucede en el Programa 5.5. Es importante comprender las diferencias entre las siguientes expresiones.

- puntero → Contiene el valor almacenado en la variable puntero.
- &puntero → Devolverá la dirección de la variable puntero.
- *puntero → Devolverá el valor almacenado en la posición de memoria cuya dirección se almacena en la variable puntero.

El paso de variables con punteros

Se puede modificar el valor de una variable mediante el uso de punteros. Esto permite pasar más de un valor desde una función invocada hacia la función que la invocó. El Programa 5.6 muestra cómo se hace esto.

Programa 5.6

```
#include <stdio.h>

main()
{
    char *puntero; /* Un puntero. */
    char variable; /* Una posición para guardar un valor. */
    variable = 1;
    puntero = &variable;
    printf("Valor almacenado en variable = %d\n", variable);
    printf("Valor almacenado en puntero = %d\n", puntero);
    *puntero = 2;
    printf("Nuevo valor almacenado en variable = %d\n", variable);
    printf("Valor almacenado en puntero = %d\n", puntero);
}
```

La ejecución del Programa 5.6 produce:

```
Valor almacenado en variable = 1
Valor almacenado en puntero = 7732
Nuevo valor almacenado en variable = 2
Valor almacenado en puntero = 7732
```

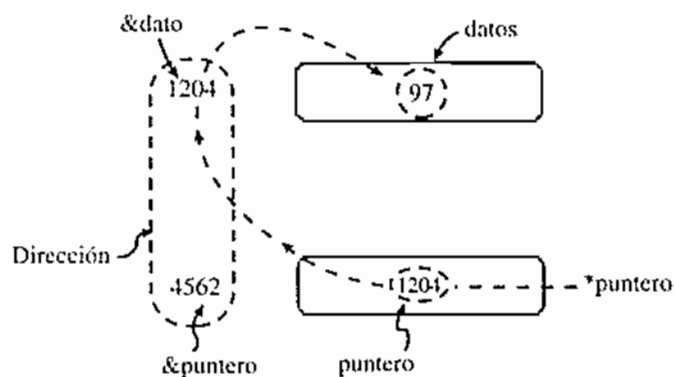



Figura 5.10. Concepto de puntero.

Análisis del programa

El programa declara una variable de tipo puntero a char y otra de tipo char:

```
char *puntero; /* Un puntero. */
char variable; /* Una posición para guardar un valor. */
```

A continuación, se asigna un 1 a la variable de tipo char y se asigna a la variable puntero la dirección de variable.

```
variable = 1;
puntero = &variable;
```

Para mostrar el contenido de variable y puntero se imprimen sus valores:

```
printf("Valor almacenado en variable = %d\n", variable);
printf("Valor almacenado en puntero = %d\n", puntero);
```

Lo que produce:

```
Valor almacenado en variable = 1
Valor almacenado en puntero = 7732
```

Seguidamente se asigna un 2 a *puntero.

```
*puntero = 2;
```

La Figura 5.11 muestra lo que sucede cuando se realiza dicha asignación. El valor 2 se almacena en la dirección de memoria contenida en el puntero. Así, el valor de esta posición pasa de 1 a 2, sin cambiar el valor almacenado en la variable puntero como se muestra con las siguientes líneas:

```
printf("Nuevo valor almacenado en variable = %d\n", variable);
printf("Valor almacenado en puntero = %d\n", puntero);
```

Las cuales producen la siguiente salida:

```
Nuevo valor almacenado en variable = 2
Valor almacenado en puntero = 7732
```

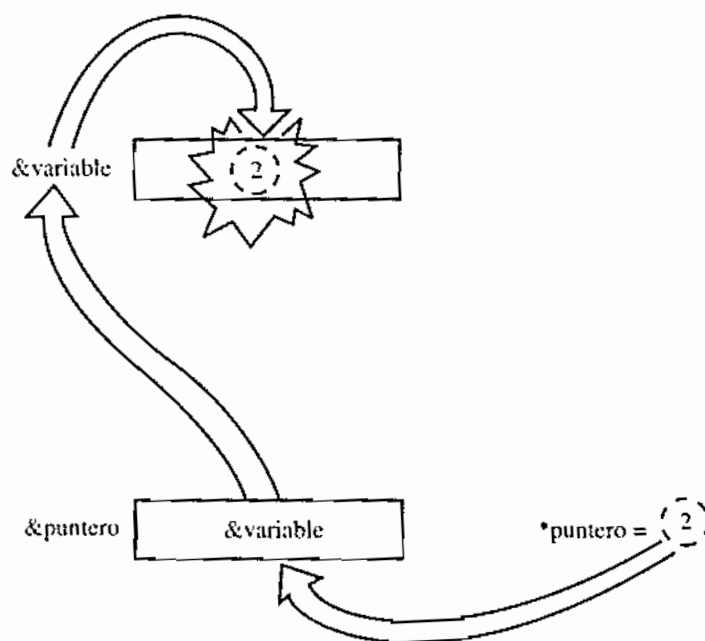


Figura 5.11. Acción de asignar un puntero.

Algunos ejemplos

El Ejemplo 5.3 presenta algunas muestras del uso de punteros.

Ejemplo 5.3 Dado un programa con las siguientes declaraciones.

```
char numero;
int *p,
```

Suponiendo que la dirección de `numero` es 7753 y la de `p` 8364, determinar los siguientes valores:

- a. `numero` b. `p` c. `&numero` d. `&p` e. `*p`

Después de ejecutar, de forma acumulativa, las siguientes sentencias:

- A. `numero = 5; p = 12`
- B. `numero = p`
- C. `numero = &p`
- D. `p = &p`
- E. `p = &numero`
- F. `*p = 10`

Solución

- A. `numero = 5, p = 12, &numero = 7753, &p = 8364, *p = dato almacenado en 12`
- B. `numero = 12, p = 12, &numero = 7753, &p = 8364, *p = dato almacenado en 12`

- C. `numero = 8364, p = 12, &numero = 7735, &p = 8364, *p = dato almacenado en 12`
 D. `numero = 8364, p = 8364, &numero = 7735, &p = 8364, *p = dato almacenado en 8364`
 $\Rightarrow 8364$
 E. `numero = 8364, p = 7735, &numero = 7735, &p = 8364, *p = dato almacenado en 7735`
 $\Rightarrow 8364$
 F. `numero = 10, p = 7735, &numero = 7735, &p = 8364, *p = dato almacenado en 7735`
 $\Rightarrow 10$

La Figura 5.12 muestra estos resultados.

Conclusión

Esta sección ha presentado una introducción al concepto de puntero. Se han explicado las diferencias entre el valor de la variable, su dirección y un puntero que contenga la

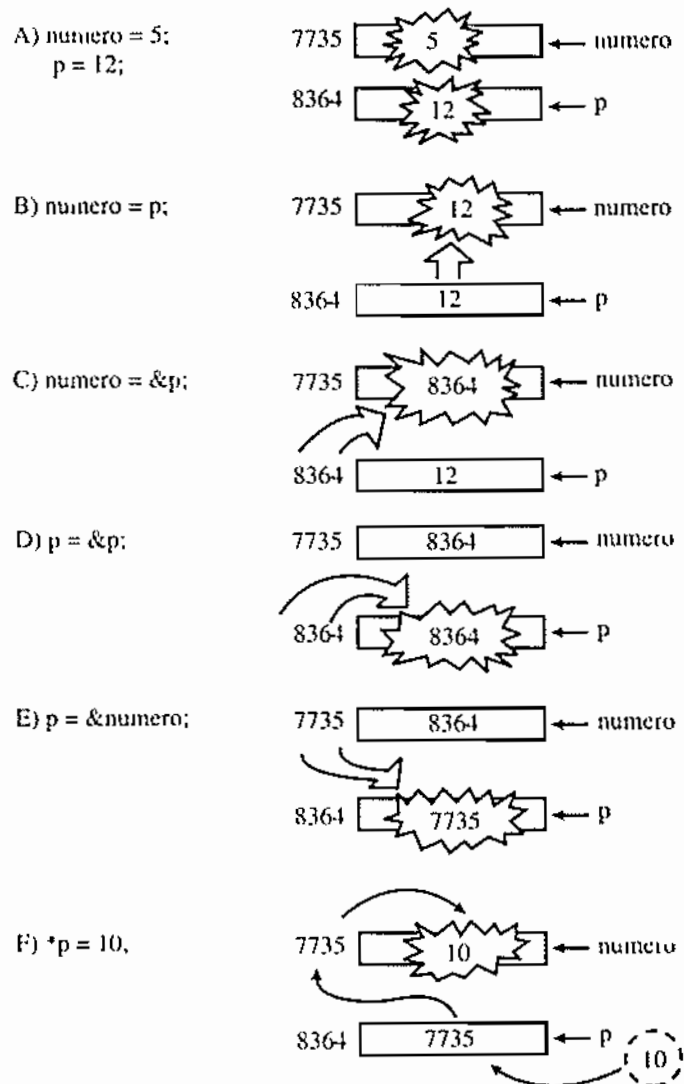


Figura 5.12. Solución para el Ejemplo 5.3.

dirección de la variable. Asimismo, se ha mostrado cómo cambiar el valor de una posición de memoria mediante el uso de un puntero. Compruebe que se han comprendido correctamente estos conceptos mediante el siguiente repaso.

Repaso de la Sección 5.3

1. ¿Qué es un puntero?
2. ¿Por qué a un puntero se le denomina así?
3. ¿Cómo se asigna a un puntero la dirección de una variable?
4. ¿Cómo se pasa un valor usando punteros? ¿Qué se debe asegurar antes de hacer esto?
5. ¿Cómo se declara un puntero?

5.4. PASO DE VARIABLES

Presentación

Esta sección muestra otro método para pasar un valor desde la función llamada hacia la función que la invoca: el uso de punteros.

Se trata de un mecanismo muy importante para el diseño de programas y, por lo tanto, es conveniente prestar mucha atención a esta sección y estudiar detenidamente los programas que se presentan en la misma.

Idea básica

El Programa 5.7 muestra una forma de pasar un valor desde la función llamada a la función que realizó la llamada. Obsérvese que cuando una función no tiene parámetros, esto se indica usando el tipo `void`.

Programa 5.7

```
#include <stdio.h>

int llameme(void); /* Función invocada desde main. */

main()
{
    int x; /* Variable que recibe lo devuelto por la función */
    x = llameme();
    printf("El valor de x es: %d",x);
    exit(0);
}

int llameme(void)
{
    return(5);
}
```

La ejecución del programa produce:

El valor de x es: 5

Como muestra la Figura 5.13, lo único que hace el programa es imprimir el valor generado por la función `llameme()`. El valor devuelto por la función mediante la sentencia `return` es asignado a la variable `x` en la función `main()`.

Este método es válido para el caso de que la función invocada deba devolver un único valor. Para devolver dos o más valores desde la función invocada, se necesitan usar punteros.

Paso de valores con punteros

La Figura 5.14 muestra cómo usar un puntero para pasar un valor de la función invocada a la función que realizó la llamada. El resultado final es igual que el del programa anterior, pero en vez de usar una sentencia `return` se usará un puntero.

Para ello, la función `llameme()` debe tener un argumento formal de tipo puntero:

```
void llameme(int *p);
```

Esto se puede interpretar como si se hubiera reservado una posición de memoria para almacenar la dirección de otra variable. Este concepto se observa en la Figura 5.15.

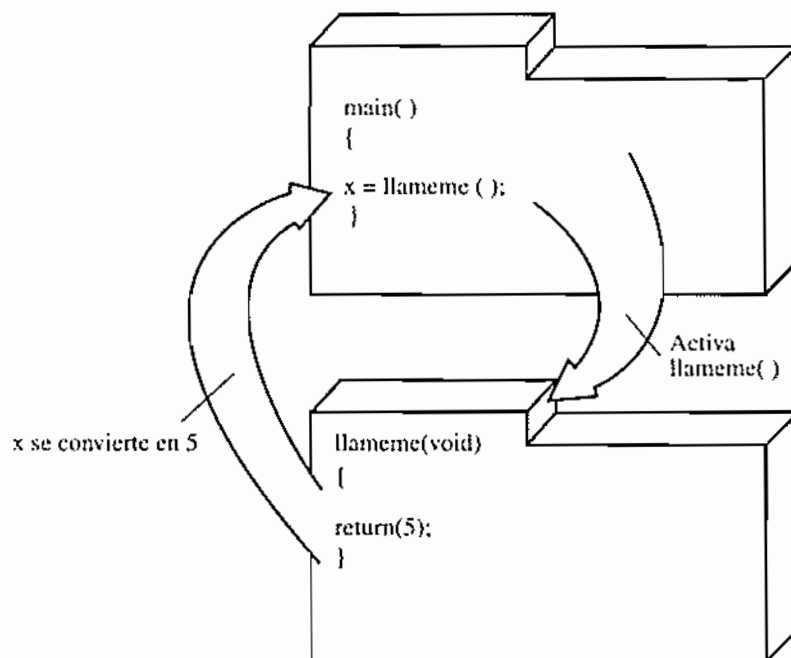


Figura 5.13. Operaciones del Programa 5.7.

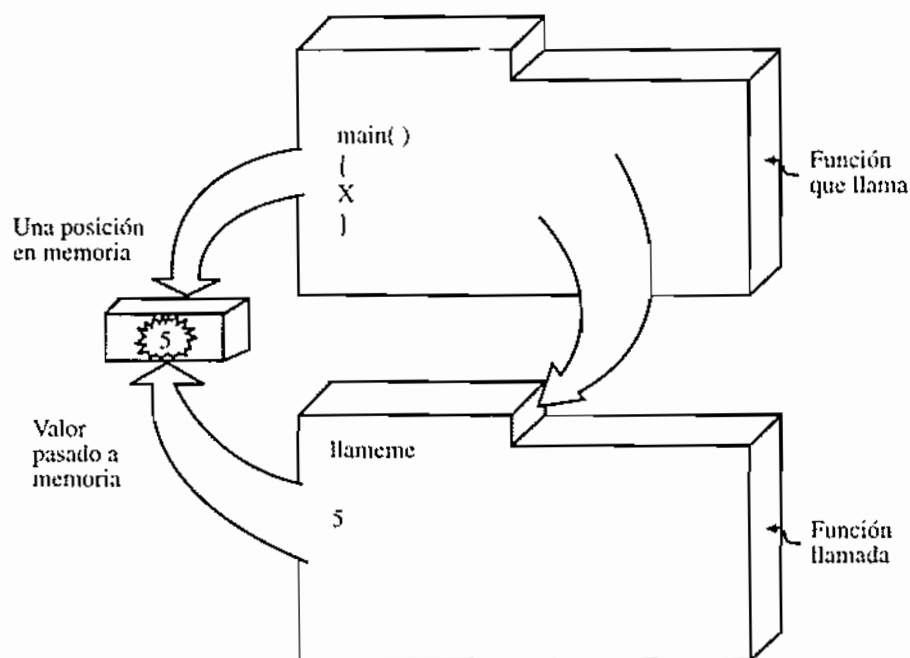


Figura 5.14. Concepto de paso de un valor desde una función llamada.

Cuando se invoca a la función `llameme()` se le pasa la dirección de la variable que se va a modificar.

```
main()
{
    int x;
    llameme(&x);
}
```

La Figura 5.16 muestra lo que sucede cuando se produce la llamada.

Cuando se ejecuta la función, almacena el valor 5 en la dirección apuntada por el puntero como se muestra en la Figura 5.17.

```
void llameme(int *p)
{
    *p = 5;
}
```

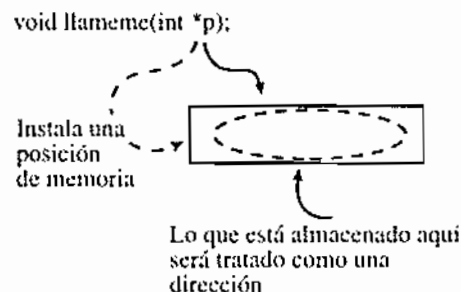


Figura 5.15. Instalando un argumento de tipo puntero.

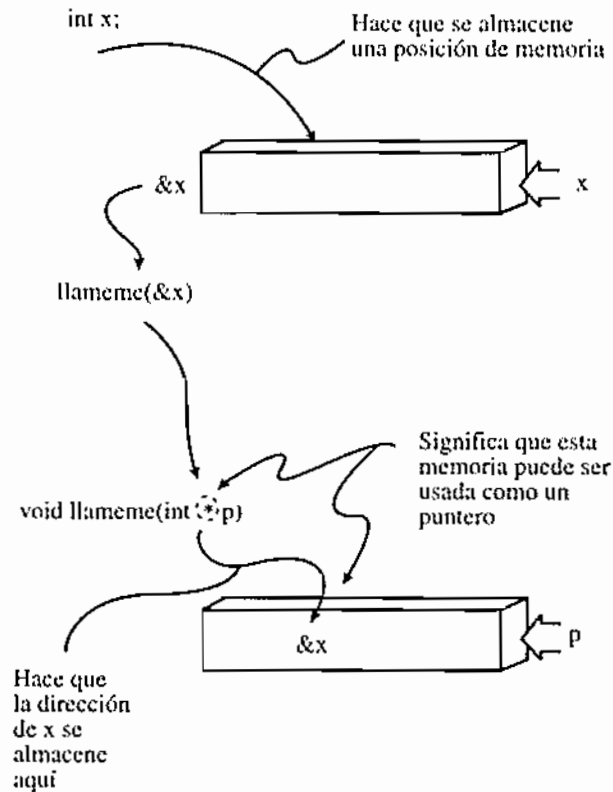


Figura 5.16. Acciones de la función que llama.

Lo que se está analizando corresponde con el Programa 5.8. Obsérvese que la función `llamame()` no devuelve ningún valor por lo que se declara de tipo `void`.

La ejecución del programa produce la siguiente salida:

```
El valor de x es 0
El nuevo valor de x es 5
```

Programa 5.8

```
#include <stdio.h>

void llamame(int *p);

main()
{
    int x;
    x = 0;
    printf("El valor de x es %d\n", x);
    llamame(&x);
    printf("El nuevo valor de x es %d\n", x);
}

void llamame(int *p)
{
    *p = 5;
}
```

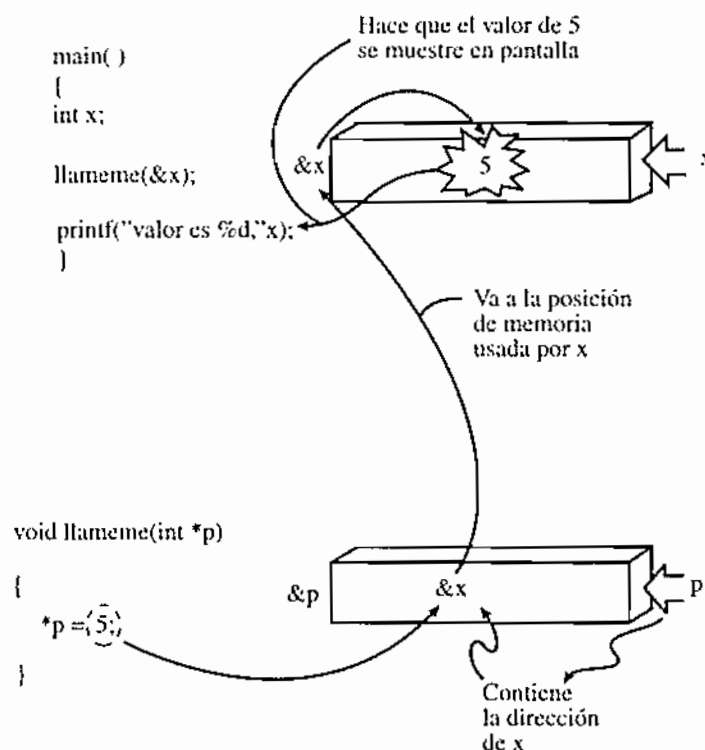


Figura 5.17. Acciones de una función llamada.

Parece demasiado esfuerzo para simplemente asignar un 5 a la variable `x`. Sin embargo, este ejemplo ha mostrado cómo pasar un valor de una función a otra. A continuación se verá una aplicación de lo anteriormente descrito.

Aplicación del paso de valores con punteros

El Programa 5.9 calcula la intensidad que circula en una resistencia dados el valor de la resistencia y el voltaje en la misma. La función `main()` se dedica únicamente a llamar al resto de las funciones que son las que realizan el trabajo.

Programa 5.9 `#include <stdio.h>`

```

void explicar_programa(void);
void obtener_valores(float *r, float *v);
float calcular(float resistencia, float voltaje);
void imprimir_respuesta(float intensidad);

main()
{
    float resistencia;
    float voltios;
    float intensidad;

```



```

    explicar_programa();
    obtener_valores(&resistencia, &voltios);
    intensidad = calcular(resistencia, voltios);
    imprimir_respuesta(intensidad);
}

void explicar_programa()
{
    printf("Este programa calcula el valor de la intensidad\n");
    printf("en amperios. Debera introducir el valor de la \n");
    printf("resistencia y el voltaje en voltios.\n");
}

void obtener_valores(float *r, float *v)
{
    float resistencia;
    float voltaje;

    printf("\n\nIntroduzca la resistencia en ohmios = ");
    scanf("%f",&resistencia);
    printf("\nIntroduzca el voltaje en voltios = ");
    scanf("%f",&voltaje);
    *r = resistencia;
    *v = voltaje;
}

float calcular(float resistencia, float voltaje)
{
    float intensidad;

    intensidad = voltaje / resistencia;
    return(intensidad);
}

void imprimir_respuesta(float intensidad)
{
    printf("El valor de la intensidad es %f amperios.", intensidad);
}

```

Suponiendo que el usuario introduce un valor de 10 para el voltaje y de 5 para la resistencia, la ejecución del programa produce la siguiente salida:

```

Este programa calcula el valor de la intensidad
en amperios. Debera introducir el valor de la
resistencia y el voltaje en voltios.
Introduzca la resistencia en ohmios = 5
Introduzca el voltaje en voltios = 10
El valor de la intensidad es 2 amperios.

```

Análisis del programa

El diseño del programa sigue la secuencia típica de los programas interactivos:

1. Explicar el programa al usuario.
2. Leer los valores introducidos por el usuario.
3. Realizar los cálculos.
4. Mostrar los resultados.

Esta secuencia queda reflejada en los nombres de las funciones empleadas:

```
void explicar_programa(void);
void obtener_valores(float *r, float *v);
float calcular(float resistencia, float voltaje);
void imprimir_respuesta(float intensidad);
```

Obsérvese que la función `obtener_valores()` tiene dos punteros como argumentos. Esto se debe a que tendrá que devolver dos valores a la función que la invocó, lo cual sólo se puede hacer usando punteros. Nótese que la única función que no es de tipo `void` es `calcular()` debido a que debe devolver el valor de la intensidad calculado. Al tratarse de un único valor puede usar la sentencia `return` para devolverlo sin necesidad de utilizar punteros.

La función `main()` declara tres variables que contendrán los valores que se pasan entre las funciones:

```
main()
{
    float resistencia;
    float voltios;
    float intensidad;
```

El cuerpo de la función `main()` contiene las llamadas a las demás funciones.

```
explicar_programa();
obtener_valores(&resistencia, &voltios);
intensidad = calcular(resistencia, voltios);
imprimir_respuesta(intensidad);
```

Obsérvese que la llamada a la función `obtener_valores()` usa el operador de dirección. Esto es necesario debido a que el argumento formal de la función es de tipo puntero. Se le pasan a la función las direcciones de las dos variables para que en ellas pueda devolver los valores introducidos por el usuario.

La primera función invocada explica al usuario el propósito del programa:

```
void explicar_programa()
{
    printf("Este programa calcula el valor de la intensidad\n");
    printf("en amperios. Debera introducir el valor de la \n");
    printf("resistencia y el voltaje en voltios.\n");
}
```

La siguiente función es la que tiene dos argumentos de tipo puntero. Define también dos variables que se usarán para almacenar los valores introducidos por el usuario:

```
void obtener_valores(float *r, float *v)
{
    float resistencia;
    float voltaje;
```

El cuerpo de esta función lee dichos valores usando `scanf()`:

```
printf("\n\nIntroduzca la resistencia en ohmios = ");
scanf("%f",&resistencia);
printf("\nIntroduzca el voltaje en voltios = ");
scanf("%f",&voltaje);
```

A continuación se asignan estos valores a los punteros para que así puedan accederse desde la función `main()`.

```
*r = resistencia;
*v = voltaje;
```

Una vez que `main()` ha obtenido los valores introducidos por el usuario, se los pasa a la siguiente función que realiza los cálculos:

```
float calcular(float resistencia, float voltaje)
{
    float intensidad;

    intensidad = voltaje / resistencia;
    return(intensidad);
}
```

Puesto que sólo se necesita devolver un valor, se ha usado la sentencia `return`.

La última función simplemente muestra en pantalla el valor de la intensidad calculada.

```
void imprimir_respuesta(float intensidad)
{
    printf("El valor de la intensidad es %f amperios.", intensidad);
}
```

Aspectos claves del programa

Aunque se trate de un programa muy sencillo, muestra cómo debería ser la estructura de un programa C. En primer lugar, la función `main()` sólo se dedica a llamar al resto de las funciones. Esto permite ver fácilmente cómo es la estructura del programa.

El siguiente aspecto es que cada función hace una tarea específica lo que consigue que el programa sea fácil de comprender, depurar y modificar. Se puede desarrollar cada función de forma independiente.

Conclusión

Esta sección ha mostrado cómo pasar valores usando punteros. El concepto de puntero se seguirá aplicando en el resto de capítulos de este libro. Compruebe si ha asimilado bien los conceptos presentados en esta sección con el siguiente repaso.

Repaso de la Sección 5.4

1. Enumere las dos maneras de pasar un valor desde una función invocada a la función que la llamó.
2. ¿Cuáles son las limitaciones de la sentencia `return` para devolver valores a la función que realizó la llamada?
3. Explique cómo se puede pasar más de un valor desde una función a la función que la invocó.
4. Describa el mecanismo para pasar valores a una función usando punteros como argumentos.
5. ¿Por qué es conveniente usar funciones separadas?

5.5. EL ÁMBITO DE LAS VARIABLES

Presentación

En esta sección se presenta información sobre el ámbito de las variables en C y su relación con las funciones. A partir de esta información, el lector podrá comprender mejor qué técnicas son recomendables en el desarrollo de un programa y cuáles no lo son.

Variables locales

Todas las variables utilizadas hasta ahora en los programas presentados han sido locales. El Programa 5.10 ilustra el concepto de una **variable local**. En este programa se intenta usar una variable declarada en una función desde otra función invocada por la primera.

Programa 5.10 `#include <stdio.h>`

```
void otra_funcion(void);

main()
{
    int una_variable;

    una_variable = 5;
    printf("El valor de una_variable es %d", una_variable);
    otra_funcion();
}
```

```
void otra_funcion(void)
{
    printf("El valor de una_variable en esta funcion es %d",
           una_variable);
}
```

Este programa producirá errores de compilación ya que la segunda función no sabe nada de la variable `una_variable` que está declarada en la función `main()`. Esto se debe a que cuando se declara una variable en una función, ésta sólo es visible desde dentro de la función, esto es, es local a dicha función. Dicho de otra forma, el **ámbito** de una variable local abarca sólo la función donde se declara y la **vida** de esta variable se restringe al tiempo en el que está activa esta función.

Variables globales

Para hacer que una variable sea visible desde todas las funciones del programa, debe declararse antes que la función `main()`, como muestra el Programa 5.11 que, a diferencia del anterior, se compilará sin errores.

Programa 5.11 `#include <stdio.h>`

```
void otra_funcion(void);
int una_variable;

main()
{
    una_variable = 5;
    printf("El valor de una_variable es %d\n", una_variable);
    otra_funcion();
}

void otra_funcion()
{
    printf("El valor de una_variable en esta funcion es %d",
           una_variable);
}
```

Puesto que `una_variable` se ha declarado fuera de la primera función, es ahora visible desde todas las funciones y por ello el programa producirá la siguiente salida:

```
El valor de una_variable es 5
El valor de una_variable en esta funcion es 5
```

Una variable declarada de esta forma se denomina una **variable global**. El ámbito de una variable de este tipo abarca todo el programa y su vida coincide con la del programa. La Figura 5.18 muestra este comportamiento.

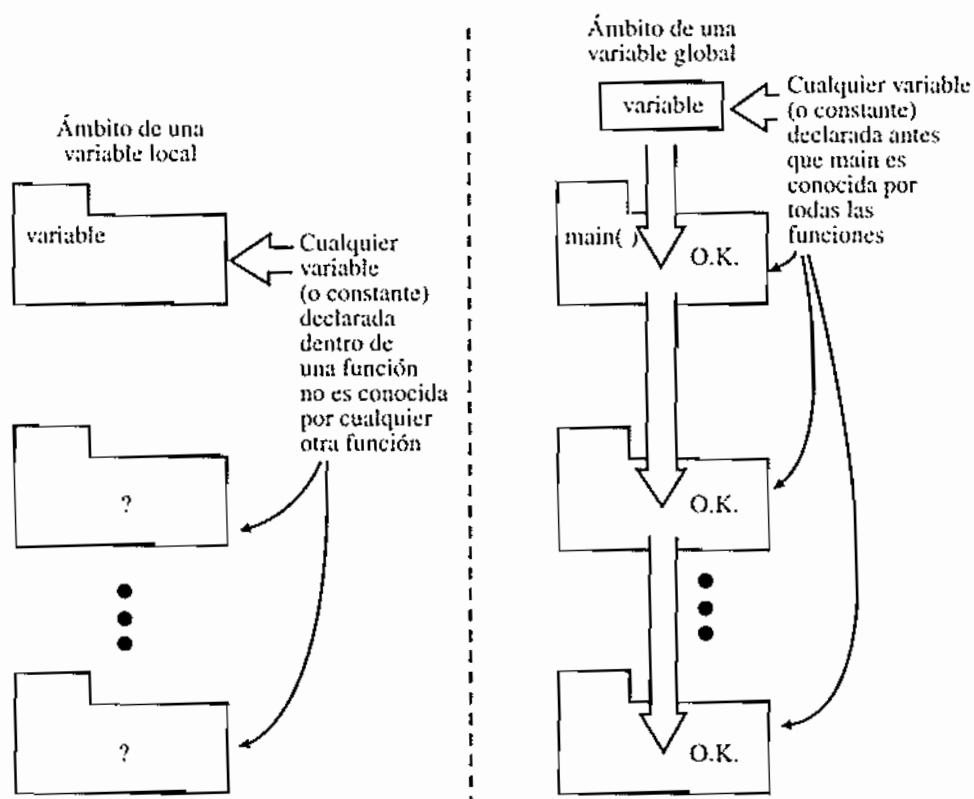


Figura 5.18. Concepto de ámbito.

Precaución con el uso de variables globales

El uso de variables globales puede causar a veces errores inesperados, debido a que cualquier función puede cambiar el valor de una variable global. De esta forma, cuando otra función utiliza una variable global, ésta puede tener un valor distinto del previsto. Esta situación es especialmente probable cuando se usan variables globales en funciones recursivas.

Se considera una buena técnica de programación mantener las variables lo más locales que se pueda. Así se logra protegerlas de cambios realizados por otras funciones. Cuando se precise hacer accesible el valor de una variable a una función, se pasará dicha variable como argumento.

Por tanto, es una buena regla evitar las variables globales en los programas. En el Programa 5.12 se muestra alguno de los peligros del uso de las variables globales.

Programa 5.12 `#include <stdio.h>`

```
void otra_funcion(void);
int contador;

main()
{
    contador = 0;
```

```

while (contador < 5)
{
    printf("La cuenta es %d\n", contador);
    contador++;
}
otra_funcion();
}

void otra_funcion(void)
{
    while (contador < 6)
    {
        printf("La cuenta en esta funcion es %d\n", contador);
        contador++;
    }
}

```

La ejecución del programa produce el siguiente resultado:

```

La cuenta es 0
La cuenta es 1
La cuenta es 2
La cuenta es 3
La cuenta es 4
La cuenta en esta funcion es 5

```

Observe que la salida producida por la segunda función podría sorprender al programador (o a cualquiera que intente utilizar el programa). Debido al uso de una variable global, la segunda función ve el valor almacenado por la función `main()` en la variable `contador`. Este tipo de situaciones puede tener efectos desastrosos, sobre todo cuando aparecen en programas más grandes donde varios programadores escriben las distintas funciones del programa.

Conclusión

En esta sección se han presentado los conceptos de ámbito y vida de una variable (o constante). Se ha mostrado también la diferencia entre variables locales y globales. Por último, se han podido ver los efectos negativos del uso de las variables globales y, por tanto, se ha expresado la recomendación de evitar el uso de las mismas. Compruebe su comprensión de esta sección mediante el siguiente repaso.

Repaso de la Sección 5.5

1. ¿A qué se refiere el término variable local?
2. Explique el significado del término «ámbito» cuando se aplica a las variables.
3. ¿Cómo se declara una variable global? ¿Y una local?
4. ¿Se considera una buena técnica de programación el uso de variables globales?
5. ¿Cómo se pueden pasar valores entre funciones si no se usan variables globales?

5.6. CLASES DE VARIABLES

Presentación

Esta sección presenta varias formas de clasificar los datos que utiliza un programa C. El material presentado en esta sección se basa en los conceptos presentados en la sección previa.

Constantes

Una **constante** es un valor que nunca cambia durante la ejecución de un programa. Se declaran usando la palabra reservada `const`, como se muestra en el Programa 5.13.

Programa 5.13

```
#include <stdio.h>

const float DOS_PI = 6.28;

main()
{
    printf("El valor de 2 multiplicado por pi es %f\n", DOS_PI);
}
```

En este programa, si se intenta modificar el valor de la constante global `DOS_PI`, el compilador producirá un aviso. Es aconsejable usar constantes globales puesto que su valor nunca cambia.

Variables automáticas

Las variables locales que se han estado utilizando hasta ahora se denominan **variables automáticas**. Por defecto, cualquier variable especificada dentro de una función se considera automática. Sin embargo, si se desea enfatizar este hecho, se puede utilizar la palabra reservada `auto`, como muestra el Programa 5.14.

Programa 5.14

```
#include <stdio.h>

main()
{
    auto int valor;

    valor = 5;
    printf("El valor es %d\n", valor);
}
```

En este programa se podría haber eliminado la palabra reservada `auto` sin modificar su comportamiento.

Variables externas

Una **variable externa** es una variable global cuya definición debe aparecer en otra parte del programa (posiblemente en otro archivo de código fuente del programa). La palabra reservada para esta clase de variables es `extern` (véase Capítulo 10).

Variables estáticas

Una **variable estática** es una variable cuyo ámbito es local pero cuya vida coincide con la del programa. Una variable estática declarada en una función, a diferencia de lo que ocurre con las variables automáticas, mantiene su valor entre diferentes invocaciones de la misma. El Programa 5.15 muestra esta característica.

Programa 5.15 `#include <stdio.h>`

```
void segunda_funcion(void);

main()
{
    segunda_funcion();
    segunda_funcion();
    segunda_funcion();
}

void segunda_funcion(void)
{
    static int numero;

    numero++;
    printf("Se ha llamado %d veces a esta funcion.\n", numero);
}
```

La ejecución del programa genera lo siguiente:

```
Se ha llamado 1 veces a esta funcion.
Se ha llamado 2 veces a esta funcion.
Se ha llamado 3 veces a esta funcion.
```

El punto importante de este programa es que la variable estática mantiene su valor a través de las sucesivas llamadas a la función. Sin embargo, sigue siendo una variable local sólo accesible desde la función donde se declaró.

Variables de registro

Una **variable de registro** informa al compilador del lenguaje que el programador desea que esta variable se almacene en un lugar de rápido acceso, normalmente en los registros del computador. Puesto que hay un número limitado de registros, cuando el programador especifica una variable de esta clase, el compilador intentará almacenarla en un registro pero, si no hay ninguno disponible, la almacenará en memoria.

La palabra reservada para esta clase es `register`. El Programa 5.16 muestra el uso de una variable de esta clase.

Programa 5.16 `#include <stdio.h>`

```
main()
{
    register contador;

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

Este programa debería ejecutar más rápido con una variable de este tipo ya que no sería necesario acceder a la memoria para incrementar el contador.

Visión general

La Tabla 5.3 muestra las diversas clases de variables disponibles en C.

Tabla 5.3. Clases de variables

Tipo	Palabra reservada	Dónde se declara	Ámbito	Vida
Global	Ninguna	Antes de cualquier función.	Todo el programa.	Hasta que termina el programa.
Global	<code>extern</code>	Antes de cualquier función dentro de un archivo.	Todos los archivos que incluyen a otros donde haya declaraciones <code>extern</code> .	Mientras cualquiera de los archivos anteriores está activo.
Local	Ninguna o <code>auto</code>	Dentro de la función.	Sólo dentro de la función donde es declarada.	Mientras la función esté activa.
Local	<code>register</code>	Dentro de la función.	Sólo dentro de la función donde es declarada.	Mientras la función esté activa.
Local	<code>static</code>	Dentro de la función.	Sólo dentro de la función donde es declarada.	Hasta que termina el programa.

Conclusión

Esta sección ha presentado las distintas clases de variables disponibles en C mostrando mediante ejemplos las diferencias entre ellas. Compruebe su comprensión de esta sección a través del siguiente repaso.

Repaso de la Sección 5.6

1. ¿Qué efecto tiene la palabra reservada `const` cuando se aplica a una variable?
2. ¿Qué es una variable automática?
3. ¿Qué es una variable estática?
4. ¿Qué es una variable de registro?

5.6. DEPURACIÓN E IMPLEMENTACIÓN DE PROGRAMAS

Problemas en la programación en C

Esta sección presenta algunos de los errores de programación más comunes en C. Esto ayudará al lector a evitar perder el tiempo depurando este tipo de errores.

Los punteros

Comprensiblemente, muchos de los programadores noveles encuentran dificultades con el uso de los punteros. En primer lugar, se debe estar seguro de que se comprenden las diferencias entre las distintas formas de variables de tipo puntero. La Tabla 5.4 resume estas diferencias.

El operador de indirección (*) accede indirectamente a un valor, a través de un puntero. Por lo tanto, el operando deberá ser un puntero.

Tabla 5.4. Nomenclatura de punteros

Dado: `int *ptr;`
`int datos`
 [Se sigue que:]

Operación	Ejemplo	Significado
Operador de dirección	<code>&datos</code>	Dirección de la variable <code>datos</code> .
Asignación	<code>ptr = &datos</code>	Coloca en la variable <code>ptr</code> la dirección de <code>datos</code> .
Variable	<code>ptr</code>	Valor de la variable <code>ptr</code> , que ahora contiene la dirección de <code>datos</code> .
Operador indirección	<code>*ptr</code>	Valor en la variable <code>datos</code> .

Es a menudo útil desarrollar pequeños programas con el fin de familiarizarse con los conceptos presentados en dicha tabla.

Punteros sin valor inicial

Un error común es olvidarse de dar valor inicial a un puntero. **Dar un valor inicial** significa establecer donde señalará el puntero antes de usarlo. Como ejemplo, el siguiente fragmento de programa no es correcto:

```
main()
{
    int *ptr;
    *ptr=5;
}
```

Aquí, el programador está intentando asignar un valor a la posición de memoria apuntada por el puntero. Sin embargo, nadie sabe dónde se almacenará dicho valor. Lo que se debe recordar es que siempre se debe saber dónde apunta un puntero antes de usarlo como tal.

Asignación e igualdad

Otro aspecto que causa problemas a los programadores noveles es olvidar que el símbolo = (operador de asignación) no significa «igual a». El = significa asignar el valor de la derecha a la variable de la izquierda. Si se desea comprobar la igualdad de dos expresiones, se debe usar el símbolo ==. Por ejemplo, el siguiente fragmento de programa compilará y ejecutará. Sin embargo, el programador no obtendrá lo que esperaba:

```
if (esto = eso)
    printf("esto es igual a eso");
else
    printf("esto no es igual a eso");
```

Lo que ocurre es que se está asignando el valor de la variable eso a la variable esto. No se está realizando ninguna comparación. Lo correcto sería:

```
if (esto == eso)
    printf("esto es igual a eso");
else
    printf("esto no es igual a eso");
```

Omisión del operador de dirección

Otro error común es olvidar pasar los valores por dirección. El siguiente fragmento muestra esta situación:

```
main()
{
```

```

int entrada;
printf("Deme un numero => ");
scanf("%d", entrada);
printf("Ha introducido el numero %d", entrada);
}

```

En este fragmento el valor introducido por el usuario no hará lo esperado. La función `scanf()` requiere el uso del operador de dirección `&` para poder devolver el dato leído. Lo correcto sería modificar la llamada a la función `scanf()` de la siguiente manera:

```
scanf("%d", &entrada);
```

Lo mismo ocurre con cualquier función definida en el programa que devuelva varios valores a la función que la invocó.

Conclusión

Esta sección ha presentado algunos de los errores de programación más comunes cuando se usa el lenguaje C. Éstos están relacionados con el uso de punteros, la confusión entre el operador de asignación y el de igualdad, y la forma de pasar valores por dirección. Como es habitual, es conveniente realizar el siguiente repaso.

Repaso de la Sección 5.7

1. ¿Qué es el operador de dirección? ¿Qué hace?
2. ¿Qué es el operador de indirección? ¿Qué hace?
3. Explica la diferencia entre el operador de asignación y el de igualdad.

5.8. PROGRAMAS DE APLICACIÓN

En esta sección se verán más ejemplos del uso de punteros para acceder a los datos.

El Programa 5.17 convierte un número decimal a binario usando la operación AND binaria junto con la operación de desplazamiento a la derecha. Se usa una máscara binaria para seleccionar un bit cada vez e imprimirlo. Esta máscara tiene el valor 80 en hexadecimal por lo que selecciona el bit más significativo. En cada vuelta del bucle se desplaza una posición el valor de la máscara. Así, después de ocho vueltas se habrán tratado todos los bits del número.

Programa 5.17 `#include <stdio.h>`

```

void abin(char *valor);

main()
{

```

```

    unsigned char numero;    /* Almacenamiento para el número leído */
    printf("Introduzca un numero del 0 al 255 => ");
    scanf("%d",&numero);
    printf("El numero %d en binario es ",numero);
    abin(&numero);
}

void abin(char *valor)
{
    unsigned char posicion = 0x80;
    char temp;

    while (posicion != 0)
    {
        temp = *valor & posicion;
        (temp == 0) ? printf("0 ") : printf("1 ");
        posicion = posicion >> 1;
    }
}

```

Un ejemplo de la ejecución del programa es el siguiente:

```

Introduzca un numero del 0 al 255 => 143
El numero 143 en binario es 1 0 0 0 1 1 1 1

```

El Programa 5.18 muestra cómo se pueden generar una secuencia de números pseudo-aleatorios.

Programa 5.18 #include <stdio.h>

```

void generador_aleatorios(unsigned char *aleatorio);

main()
{
    unsigned char aleatorio;    /* Almacenamiento para el número
                                aleatorio. */
    int contador;

    printf("Introduzca la semilla de aleatorios (del 0 a 255) => ");
    scanf("%d",&aleatorio);
    for (contador = 1; contador <= 10; contador++)
    {
        generador_aleatorios(&aleatorio);
        printf("El proximo numero aleatorio es %d\n",aleatorio);
    }
}

void generador_aleatorios(unsigned char *aleatorio)
{
    unsigned char temp1,temp2;    /* Utilizados en los cálculos. */

```

```

temp1 = *aleatorio &0x80;
temp2 = *aleatorio &0x04;
*aleatorio <<= 1;
if ((0 == temp1 | temp2) || (0x84 == temp1 | temp2))
    *aleatorio = *aleatorio | 0x01;
}

```

El Programa 5.18 intenta generar una secuencia de números aleatorios calculando el siguiente número de la secuencia modificando algunos de los bits del anterior. El usuario suministra una semilla inicial a partir de la cual se calculan diez números aleatorios usando la siguiente técnica:

1. Examina el BMS usando la máscara 0x80.
2. Examina el bit de la posición 2 usando la máscara 0x04.
3. Desplaza el número a la izquierda una posición.
4. Si son iguales los dos bits examinados, pone a 1 el bit menos significativo del número.

La ejecución del programa con una semilla inicial de 34 da los siguientes resultados:

```

Introduzca la semilla de aleatorios (del 0 a 255) => 34
El proximo numero aleatorio es 69
El proximo numero aleatorio es 139
El proximo numero aleatorio es 22
El proximo numero aleatorio es 45
El proximo numero aleatorio es 91
El proximo numero aleatorio es 183
El proximo numero aleatorio es 111
El proximo numero aleatorio es 223
El proximo numero aleatorio es 191
El proximo numero aleatorio es 127

```

La secuencia mostrada parece aleatoria, aunque realmente se genera de forma sistemática. Intente usar las semillas 73, 118 y 137, y observe los resultados obtenidos ¿Tiene limitaciones esta técnica?

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido aquí para permitirle adquirir una valiosa experiencia y, lo que es más importante, para tener una realimentación inmediata de lo que hacen los conceptos y mandatos presentados en este capítulo. Además son divertidos.

Ejercicios

1. ¿Cuáles de los valores mostrados por el Programa 5.19 son predecibles?

Programa 5.19 #include <stdio.h>

```

main()
{
    int dato = 15;
    int *apunta_a;

    apunta_a = &dato;
    printf("%d\n", apunta_a);
    printf("%d\n", &apunta_a);
    printf("%d\n", *apunta_a);
}

```

2. ¿Qué salida producirá el Programa 5.20?

Programa 5.20 #include <stdio.h>

```

const int un_valor = 15;
int funcion(void);

main()
{
    int un_valor = 12;

    printf("%d\n", un_valor);
    printf("%d\n", funcion());
}

int funcion(void)
{
    return(un_valor);
}

```

3. ¿Compilará sin errores el Programa 5.21? Si no, ¿qué cambios necesitaría?

Programa 5.21 #include <stdio.h>

```

    int a = 5;

void abc(int b);

main()
{
    int c = 10;

    abc(c);
}

void abc(int b)
{

```



```

printf("a igual a %d",a);
printf("b igual a %d",b);
printf("c igual a %d",c);
}

```

Autoevaluación

El Programa 5.22 se desarrolló para mostrar muchos de los conceptos fundamentales presentados en este capítulo. Las preguntas de esta sección se refieren a este programa.

Programa 5.22 #include <stdio.h>

```

void funcion_1(void);
void funcion_2(char *letra1, char *letra2);
int AND_binario(int este_octeto, int aquel_octeto);

const unsigned int numero_1 = 57532;
int *apunta_a;

main()
{
    int posicion_de_memoria_1;
    char este_valor;
    char aquel_valor;
    int resultado;

    apunta_a = &posicion_de_memoria_1;
    funcion_1();

    este_valor = 'a';
    aquel_valor = 'b';
    funcion_2(&este_valor, &aquel_valor);
    resultado = AND_binario(15,15);

    printf("La constante es %u.\n",numero_1);
    printf("El valor de posicion_de_memoria_1 es %d\n",
           posicion_de_memoria_1);
    printf("El contenido de este_valor = %c\n",este_valor);
    printf("El resultado es %X.\n",resultado);
}

extern char nuevo_valor;

void funcion_1(void)
{
    *apunta_a = 375;
}

void funcion_2(char *letra1, char *letra2)
{
    *letra1 = 'b';
}

```

```

        *letra2 = 'a';
    }

    int AND_binario(int este_octeto, int aquel_octeto)
    {
        return(este_octeto &aquel_octeto);
    }

```

Preguntas

1. ¿Qué salida producirá cada una de las llamadas a `printf()`?
2. ¿Qué datos son globales a todo el programa?
3. ¿Qué datos son globales a una parte del programa? ¿Cuál es su ámbito?
4. Explique cómo se almacena el valor 375 en `posicion_de_memoria_1`.
5. ¿Qué sentencia en `main()` almacena el valor 375 en `posicion_de_memoria_1`?
6. Explique por qué la salida del último `printf()` es F.
7. ¿Por qué `funcion_1` no tiene argumentos de tipo puntero y `funcion_2` sí los tiene?
8. ¿Necesita `funcion_1` ser de tipo `double`?
9. ¿Cómo `funcion_1` puede acceder al puntero `apunta_a` si no está declarado en dicha función?
10. ¿Por qué `funcion_1` modifica el valor de `posicion_de_memoria_1`?

Problemas de fin de capítulo

Conceptos generales

Sección 5.1

1. Explique una forma de visualizar la memoria de un computador.
2. ¿Cómo se distinguen las direcciones de memoria entre sí?
3. ¿Cómo se llama el proceso que realiza la UCP cuando lee una instrucción de memoria y luego la ejecuta?
4. Explique la diferencia entre el direccionamiento inmediato y el directo.

Sección 5.2

5. Enumere algunos de los tamaños más comunes de palabra usados por diversos computadores.
6. ¿Cuál es el propósito de la función `sizeof()`?
7. Determine la suma de los siguientes números binarios:
A. 0101 + 0001 B. 0111 + 0001
8. Calcule el complemento a 1 de los siguientes números binarios:
A. 0001 B. 1111 C. 1010
9. Calcule el complemento a 2 de los siguientes números binarios:
A. 0001 B. 1111 C. 1010

10. Determine los valores en base decimal de los siguientes números binarios representados en complemento a 2:

A. 0111 B. 1000 C. 1101

11. Convierta a decimal los siguientes valores hexadecimales que usan la notación complemento a 2:

A. A B. 3 C. 9C

Sección 5.3

12. ¿Qué hace el operador de dirección? ¿Cómo se usa?
13. ¿Qué hace el operador de indirección? ¿Cómo se usa?
14. ¿Qué es un puntero? ¿Cómo se usa?
15. Dadas las siguientes declaraciones, responda a estas cuestiones:

```

int dato;
int *puntero;
puntero = &dato;
*puntero = 5;

```

- A. ¿Cuál es el valor de `puntero`?
- B. ¿Cuál es el valor de `dato`?
- C. ¿Cuál es el valor de `*puntero`?

16. Explique cómo se puede pasar más de una variable desde una función hacia la función que la invocó.

Sección 5.4

17. Explique por qué se usan funciones separadas dentro de un programa.
18. ¿Cuál son las limitaciones de la sentencia `return`?
19. Explique el uso del operador `&` para devolver valores en los argumentos de una función.
20. ¿Cómo debe ser el argumento formal de una función que devuelve un valor en dicho argumento?

Sección 5.5

21. ¿De qué clase es una variable declarada en una función cuya vida está restringida a la activación de la función?
22. ¿Cuándo la vida de una variable automática coincide con la duración del programa?
23. ¿De qué clase es una variable que se declara al principio del programa antes de la función `main()`? ¿Cuál es su ámbito?
24. Explique qué se considera como una buena técnica de programación con respecto al uso de las variables.

Sección 5.6

25. ¿Qué palabra reservada se usa para asegurar que un valor no puede modificarse durante la ejecución del programa?
26. ¿De qué clase es una variable declarada en una función cuya vida coincide con la duración del programa?
27. ¿De qué clase es una variable que indica al compilador que mantenga dicha variable en un registro interno del procesador?
28. ¿De qué clase es una variable cuyo ámbito abarca más de una función?

Sección 5.7

29. Explique cómo se obtiene la dirección de la variable `dato`.
30. Muestre cómo se asigna la dirección de dato a un puntero llamado `ptr`.
31. ¿Qué símbolo se usa para la igualdad? ¿Y para la asignación?

32. ¿Qué terminología se usa cuando se utiliza el operador de dirección para asignar valores a los argumentos de una función?

Diseño de programas

Para el desarrollo de los siguientes programas, use el método desarrollado en el Programa 5.9, esto es, un diseño descendente y una estructura en bloques sin variables globales. La función `main()` se encargará básicamente de llamar al resto de las funciones. Cuando una función necesite devolver más de una variable, se usarán punteros. No olvide incluir toda la documentación en la versión final del programa.

33. Escriba un programa que permita al usuario seleccionar una de las siguientes seis operaciones binarias:

- A. COMPLEMENTO
- B. AND
- C. OR
- D. XOR
- E. Desplazamiento a la izquierda
- F. Desplazamiento a la derecha

Una vez seleccionada la operación, el usuario introducirá los valores con los que se operará.

34. Desarrolle un programa que convierta un número a la notación hexadecimal con dos dígitos. Por ejemplo, 100 se convierte en 64 y 255 en FF. Utilice operaciones binarias y de desplazamiento para realizar la conversión.
35. Diseñe su propio generador de números aleatorios y escriba un programa que genere 40 números a partir de una semilla. Imprima ocho números en cada línea.
36. Escriba un programa que genere un palíndromo de ocho bits a partir de un número de cuatro bits. Un palíndromo es un número que tiene el mismo valor tanto leído hacia adelante como hacia atrás. Por ejemplo, si el usuario introduce el valor 0011, el programa debe generar 11000011. Utilice operaciones binarias para duplicar los unos.

Cadenas de caracteres

Objetivos

Este capítulo le da la oportunidad de aprender lo siguiente:

1. La relación entre caracteres y punteros.
2. Formas de poner caracteres en memoria para formar cadenas.
3. El uso de las cadenas de caracteres en C.
4. Cómo manipulan las cadenas de caracteres las funciones de cadenas de caracteres.
5. Formas de ordenar datos en cadenas de caracteres.

Palabras clave

Cadenas de caracteres	Asignación de valor inicial a cadenas de caracteres
Matriz (array)	Matriz rectangular
Carácter nulo	Ordenación por burbuja (Bubble)
Elemento	Matriz irregular
Puntero	

Contenido

6.1. El entorno C	6.5. Funciones para manejar cadenas de caracteres
6.2. Inicio de cadenas de caracteres	6.6. Ordenación de cadenas de caracteres
6.3. Paso de cadenas de caracteres entre funciones	6.7. Programa de aplicación: formateador de texto
6.4. Trabajo con elementos de una cadena de caracteres	6.8. Programas de aplicación adicionales

Introducción

Este capítulo le dará una oportunidad de aplicar lo que ha aprendido acerca de los punteros de C. Verá la relación entre punteros y caracteres. Comprender esta relación le ayudará a desarrollar programas en C que sean capaces de realizar aplicaciones muy potentes, incluida la ordenación.

Con la información de este capítulo, tendrá un fundamento sólido para desarrollar en el futuro programas en C que resuelvan una amplia variedad de aplicaciones tecnológicas complejas.

6.1. CARACTERES Y CADENAS DE CARACTERES

Presentación

Esta sección le presenta la relación entre caracteres, punteros y cadenas de caracteres. Podrá ver cómo se usan los punteros para almacenar caracteres en memoria. Tendrá su primer contacto con vectores y matrices.

El uso de cadenas de caracteres es una parte importante de cualquier programa de tecnología. Comprender cómo usar las cadenas de caracteres en su programa le abre una nueva dimensión de la programación técnica. La habilidad para usar nombres de objetos, personas y datos es un añadido potente a sus habilidades de programación. Esta sección presenta esta importante herramienta.

Almacenamiento de cadenas de caracteres

Recuerde que, como vimos en el Capítulo 1, se puede pensar en un carácter como en una única posición de memoria que contiene un código ASCII. Una **cadena de caracteres** no es nada más que una formación de caracteres y, por tanto, se puede pensar en una cadena de caracteres como un vector de caracteres. En la Figura 6.1 puede verse cómo se almacena en memoria una cadena de caracteres.

Una formación secuencial de datos en memoria es lo que habitualmente se denomina un vector. Observe que el último carácter de la cadena de la Figura 6.1 es el **carácter nulo** de C (representado como `\0`). Todas las cadenas de caracteres en C necesitan tener el carácter nulo para indicarle donde termina la cadena. El carácter nulo se coloca automáticamente al final de la cadena por el lenguaje C cuando se define la cadena de caracteres.

La forma de decir en C que se quiere una cadena de caracteres (un vector de caracteres) es poner los corchetes `[]` inmediatamente después del identificador de la cadena de caracteres. En el Programa 6.1 se muestra cómo hacer esto y cómo recorrer la cadena del vector componente a componente.

'H'	'o'	't'	's'	'\0'

Figura 6.1. Almacenamiento de una cadena de caracteres en memoria.

Programa 6.1 `#include <stdio.h>`

```

    char cadena[] = "Hola";
main()
{
    printf("La cadena de caracteres es %s.",cadena);
    printf("Los caracteres son:\n");
    printf("%c\n",cadena[0]);
    printf("%c\n",cadena[1]);
    printf("%c\n",cadena[2]);
    printf("%c\n",cadena[3]);
    printf("%c\n",cadena[4]);
}

```

Cuando se ejecute el Programa 6.1 se obtendrá lo siguiente:

```

La cadena de caracteres es Hola
Los caracteres son:
H
o
l
a

```

Observe que el especificador de formato es `%s` para una variable de tipo cadena de caracteres y `%c` para una variable de tipo carácter.

Análisis del Programa

La sentencia `char cadena [] = "Hola";` le dice a C que reserve suficiente espacio consecutivo de memoria para almacenar la cadena: `H o l a`. La variable `cadena` es una variable de tipo vector porque representa una formación de información en memoria, no sólo la posición en memoria.

El especificador de formato `%s` le dice a C que debe imprimir la variable vector como una cadena de caracteres. El especificador `%c` le dice a C que debe imprimir un carácter individual, cosa que se hace en el programa con las sentencias de tipo:

```
printf("%c\n",cadena[0]);
```

Esta anterior, por ejemplo, hace que se imprima el primer elemento de la cadena (el elemento 0), es decir la letra mayúscula H. De igual forma, las sentencias siguientes imprimen el resto de los caracteres que componen la cadena.

Una mirada al interior

Un vector está compuesto de **elementos**. Por ejemplo, en el vector de caracteres «Hola», cada carácter es un elemento del vector. Los elementos en un vector de carac-

terres de C se numeran empezando desde 0, por tanto el elemento cero del vector anterior es la letra H. Para representar cualquier elemento del vector basta con poner el número del elemento entre los corchetes de la variable vector; en este caso `cadena[0]` es la letra H, como puede verse en el Programa 6.1.

Lo anterior denota algunos aspectos interesantes de las cadenas de caracteres en C. En primer lugar, todos comienzan con el elemento 0 del vector. En segundo lugar, todas las cadenas de caracteres deben tener un carácter nulo como último elemento del vector. Este terminador es necesario para que el programa sepa cuando termina la cadena de caracteres. Este concepto se muestra en la Figura 6.2.

¿Dónde están los elementos?

En el vector de una cadena de caracteres, cada elemento es un carácter y representa un posición de memoria única de un octeto. Como con otros tipos de datos, se puede usar un **puntero** para acceder a cualquier elemento del vector. Examine el Programa 6.2. Define la misma cadena de caracteres y además un puntero (`*ptr`, de tipo `char`) que es usado para acceder a cada elemento individual del vector (cada posición de memoria).

Programa 6.2 `#include <stdio.h>`

```

    char cadena[] = "Hola";
main()
{
    char *ptr;

    ptr = cadena;
    printf("La cadena es %s. \n", cadena);
    printf("%c\n", *ptr);
    printf("%c\n", *(ptr + 1));
    printf("%c\n", *(ptr + 2));
    printf("%c\n", *(ptr + 3));
    printf("%c\n", *(ptr + 4));
}

```

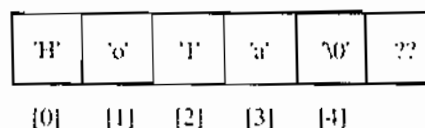


Figura 6.2. Cadena almacenada en memoria con elementos numerados.

La ejecución del Programa 6.2 imprimiría:

```
La cadena de caracteres es Hola
Los caracteres son:
H
o
l
a
```

Como puede ver, la salida del Programa 6.2 es idéntica a la del Programa 6.1. Lo que significa que lo siguiente es exactamente igual:

```
*ptr = cadena[0]
*(ptr + 1) = cadena [1]
*(ptr + 2) = cadena [2]
*(ptr + 3) = cadena [3]
*(ptr + 4) = cadena [4]
```

Esta igualdad se debe a que la variable `cadena` se declaró como una variable vector (tenía a continuación los corchetes `[]`). Cuando se usa la sentencia

```
ptr = cadena;
```

se coloca automáticamente la dirección del primer elemento de la cadena de caracteres (`cadena[0]`) en `ptr`. Por tanto, si se suma 1 al valor contenido en `ptr` (como en `*(ptr + 1)`), se está sumando 1 a la dirección contenida en él, con lo que se tiene la posición de memoria del siguiente carácter. Este concepto se ilustra en la Figura 6.3. Observe el uso de las direcciones de memoria del ejemplo (desde la 5321 hasta la 5325).

Es importante observar la diferencia entre

```
*(ptr + 1)
```

```
y
```

```
ptr + 1
```

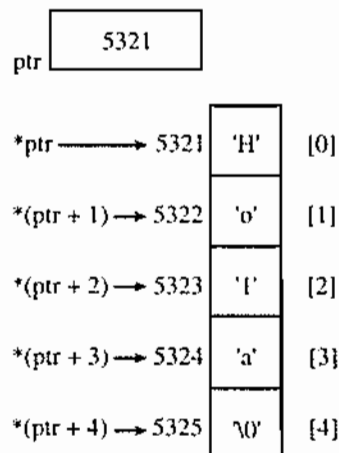


Figura 6.3. Uso de un puntero para acceder a una cadena de caracteres.

En el primer caso, el programa se dirigirá a la dirección siguiente a la almacenada en `ptr`. En el segundo caso, se sumará 1 al valor almacenado en `ptr`. Esto se aclara en el Programa 6.3.

Programa 6.3 `#include <stdio.h>`

```

    char cadena[] = "Hola";
main()
{
    char *ptr;

    ptr = cadena;
    printf("La cadena es %s. \n", cadena);
    printf("Direccion => %d | %c |\n", ptr, *ptr);
    printf("Direccion => %d | %c |\n", ptr + 1, *(ptr + 1));
    printf("Direccion => %d | %c |\n", ptr + 2, *(ptr + 2));
    printf("Direccion => %d | %c |\n", ptr + 3, *(ptr + 3));
    printf("Direccion => %d | %c |\n", ptr + 4, *(ptr + 4));
}

```

Cuando se ejecuta el Programa 6.3, la salida que se obtiene es:

```

La cadena es Hola
Los caracteres son:
Direccion => 5321 | H |
Direccion => 5322 | o |
Direccion => 5323 | l |
Direccion => 5324 | a |
Direccion => 5325 |  |

```

Este ejemplo demuestra tres cosas. Primero, las direcciones del vector de caracteres son contiguas. Segundo, `ptr + 1` es un valor mayor en 1 que el contenido en `ptr`. Tercero, `*(ptr + 1)` representa el carácter almacenado en la posición de memoria cuya dirección es mayor en 1 que la dirección almacenada en `ptr`.

Conclusión

En esta sección se ha presentado el uso de cadenas de caracteres en C. Se ha presentado también el concepto de vector como un conjunto de datos contiguos en memoria. Se ha visto que el primer elemento de un vector en C es siempre el 0. Se ha visto también que una cadena de caracteres debe terminar con el carácter nulo de C `'\0'`.

Se ha demostrado la relación entre vectores y punteros. Compruebe lo que ha comprendido de esta sección intentando hacer los ejercicios de repaso.

Repaso de la Sección 6.1

1. Defina qué se entiende por una cadena de caracteres.
2. Explique cómo se especifica una cadena de caracteres (`char`) en C.
3. ¿Cuántos elementos de un vector son necesarios para una cadena de caracteres de cinco caracteres? Explíquelo.
4. ¿Cuál es el número de elemento del primer carácter de una cadena en C?
5. Explique la relación entre punteros y elementos de una cadena de caracteres.

6.2. INICIO DE CADENAS DE CARACTERES

En esta sección se examinarán varias formas de **iniciar** una cadena de caracteres. Recuerde, como se dijo en la Sección 6.1, que la sentencia

```
char cadena [] = "Hola";
```

reserva automáticamente el número correcto de posiciones de memoria para almacenar cada elemento de la cadena, incluyendo el carácter nulo `'\0'` del final. Lo mismo puede conseguirse con la siguiente sentencia

```
char cadena [] = {'H', 'o', 'l', 'a', '\0'};
```

aunque, estará de acuerdo, es más trabajoso que usar pares de dobles comillas.

Cuando es necesario especificar la longitud de la cadena de caracteres, se debe poner un valor de tipo entero entre los corchetes. Por ejemplo, en la sentencia:

```
char cadena [4] = "Hola";
```

la longitud real de la cadena (4) se especifica entre los corchetes. Observe que el carácter nulo no cuenta como parte de la longitud de la cadena. Se puede crear una cadena de longitud cero escribiendo el carácter nulo en la posición del elemento cero, de la siguiente manera:

```
cadena [0] = '\0';
```

Observe que es frecuente especificar una longitud de la cadena de longitud mayor de la necesitada habitualmente, lo que deja al programador la carga de contar los caracteres de la cadena mientras escribe el programa. Por ejemplo, la sentencia

```
char cadena [80] = "Hola";
```

es aceptable y permite añadir más caracteres a la cadena, si se desea, posteriormente. Tenga en cuenta que las posiciones extra pueden tener valores aleatorios que pueden ser incorrectamente interpretados como caracteres válidos si no son previamente iniciados.

En algunos casos, al principio del programa sólo se desea asignar memoria para la cadena de caracteres, rellenando posteriormente los caracteres de cada posición de la cadena, tal y como se muestra en el código siguiente:

```
char cadena [4] = "Hola";
```

```
cadena [0] = 'H';
cadena [1] = 'o';
cadena [2] = 'l';
cadena [3] = 'a';
cadena [4] = '\0';
```

Observe que también es necesario poner el carácter nulo en memoria para que C sepa dónde termina la cadena de caracteres. Si se olvidara hacer esto, la cadena no sería simplemente "Hola" sino "Hola" seguido por todos los caracteres de memoria que haya detrás de la 'a' hasta que se encuentre un carácter nulo. Observe también la diferencia entre usar las comillas simples (como en 'x') o las dobles (como en "x"). Las comillas simples definen un carácter mientras que las dobles definen una cadena de caracteres. Por tanto, 'x' representa el carácter ASCII 'x' y "x" representa una cadena de caracteres que contiene dos caracteres 'x' y '\0'.

Otra forma de iniciar una cadena de caracteres es reservar memoria para ella y usar luego `scanf` para poner la cadena dentro de esa memoria. Examine el siguiente código:

```
char cadena [20];

printf ("Introduzca una cadena de caracteres => ");
scanf ("%s", cadena);
```

Observe que no se pone el carácter & delante de la variable (por ejemplo, &cadena) como se hace habitualmente en `scanf` cuando se lee un número. Esto se debe a que el nombre de la variable `cadena` representa de forma automática la dirección de comienzo de la cadena de caracteres, que es lo que necesita `scanf`.

Suponga que, durante la ejecución, el usuario escribe:

```
Introduzca una cadena de caracteres => Hola
```

Los caracteres almacenados en memoria por `scanf` serán 'H', 'o', 'l' y 'a' seguidos por el carácter nulo. Por ejemplo, si el usuario escribe

```
Introduzca una cadena de caracteres => Hola a todos.
```

`scanf` cargará únicamente los mismos cinco caracteres porque deja de buscar la entrada cuando encuentra un carácter blanco entre "Hola" y "a todos". Esto no es muy afortunado porque al usuario le gustaría ser capaz de introducir cadenas de caracteres que tengan blancos (como ocurre en los procesadores de texto). Esto puede lograrse usando la función `gets()`. La función `gets()` leerá la línea completa de texto escrita por el usuario hasta que encuentre el retorno de carro. Así pues, una técnica mejor para leer cadenas de caracteres sería la siguiente:

```
char cadena [20];

printf ("Introduzca una cadena de caracteres => ");
gets (cadena);
```

La función `gets()` sólo necesita que se le suministre el nombre de la cadena de caracteres y está incluida en el archivo `stdio.h` como una función estándar de entrada. Si se teclea:

Introduzca una cadena de caracteres => !Oh, la tengo!

la función `gets()` almacenará en memoria todos los caracteres tecleados (14, incluyendo los blancos), más un carácter nulo para terminar la cadena.

Como puede verse, hay varias opciones disponibles cuando un programador de C quiere iniciar una cadena de caracteres. Hay un último punto que merece la pena mencionar. A menudo es necesario rellenar una cadena de caracteres completa con blancos (o algún otro carácter). Esto puede hacerse fácilmente de la siguiente manera:

```
char cadena [80];
int i;

for (i=0; i<80; i++)
    cadena [i] = ' ';
cadena [80] = '\0'
```

Una vez más, observe la importancia de escribir el carácter nulo en memoria para completar la definición de la cadena de caracteres.

Compruebe lo que ha comprendido de la sección haciendo los ejercicios de repaso.

Repaso de la Sección 6.2

1. ¿Cuántos caracteres pone en memoria la siguiente sentencia?

```
char cadena [] = "abcdefghijklmnpqrstuvwxy";
```

2. ¿Qué sería almacenado en memoria por `scanf` si se introdujera el siguiente texto?

```
"Oh, lo tengo!"
```

3. ¿Cuál es la diferencia entre `"\0"` y `'\0'`?

4. ¿Por qué se usaría la sentencia `char cadena [80] = "Hola"` en lugar de `char cadena [] = "Hola"`?

6.3. PASO DE CADENAS DE CARACTERES ENTRE FUNCIONES

Debido a que las cadenas de caracteres no son nada más que vectores de caracteres, se pueden pasar cadenas de caracteres entre funciones pasando únicamente un puntero al primer elemento de la cadena. Este concepto se muestra en los Programas 6.4 y 6.5.

Programa 6.4 #include <stdio.h>

```

void funcion1(char nombre[]);
main()
{
    char cadena[20];

    printf("Cual es su nombre? => ");
    gets(cadena);
}
void funcion1(char nombre[])
{
    printf("Hola %s!\n", nombre);
}

```

El Programa 6.4 muestra cómo se pasa una cadena de caracteres a una función. La ejecución del programa produce:

```

Cual es su nombre? => Juan Estudiante
Hola Juan Estudiante!

```

El Programa 6.5 muestra cómo se recibe una cadena de caracteres desde una función.

Programa 6.5 #include <stdio.h>

```

void funcion1(char nombre[]);
main()
{
    char cadena[20];

    funcion1(cadena);
    printf("Hola %s!\n\n", cadena);
}
void funcion1(char nombre[])
{
    printf("Introduzca su nombre => ");
    gets(nombre);
}

```

La salida del Programa 6.5 es idéntica a la del Programa 6.4. Observe que entre `funcion1` y `main` sólo se pasa la dirección de la cadena de caracteres.

Para pasar múltiples cadenas de caracteres entre funciones sólo hay que incluir todos los nombres de las cadenas en la cabecera de la función. Por ejemplo, el Programa 6.6 tiene una función llamada `mismo_tamaño` que compara las longitudes de dos cadenas de caracteres que ha recibido como argumentos. Si ambas cadenas tiene el

mismo número de caracteres antes del carácter nulo, tienen la misma longitud. A la función `mismo_tamaño` se le pasan las direcciones de comienzo de ambas cadenas y devuelve 1 si ambas cadenas tienen la misma longitud y 0 si tienen distinta longitud.

Programa 6.6 `#include <stdio.h>`

```

int mismo_tamaño(char s1[], char s2[]);
main()
{
    char tira1[] = "Uno";
    char tira2[] = "Dos";
    char tira3[] = "Tres";

    if(1 == mismo_tamaño(tira1,tira2))
        printf("\"%s\" y \"%s\" son de la misma longitud.\n",
            tira1,tira2);
    else
        printf("\"%s\" y \"%s\" son de distinta longitud.\n",
            tira1,tira2);
    if(1 == mismo_tamaño(tira1,tira3))
        printf("\"%s\" y \"%s\" son de la misma longitud.\n",
            tira1,tira3);
    else
        printf("\"%s\" y \"%s\" son de distinta longitud.\n",
            tira1,tira3);
}

int mismo_tamaño(char t1[], char t2[])
{
    int i = 0, j = 0;
    while('\0' != t1[i])
        i++;
    while('\0' != t2[j])
        j++;
    if(i == j)
        return(1);
    else
        return(0);
}

```

Un punto importante a recordar tiene que ver con el paso de una cadena de caracteres como parámetro de salida de una función. Es necesario que el espacio de almacenamiento para la cadena de salida esté ya incluido en la llamada a dicha función. De otra forma, el contenido de la memoria de la cadena podría perderse cuando termine la función y su almacenamiento local sea devuelto a la memoria libre. Por esta razón, muchos programadores definen las cadenas de caracteres como variables globales cuando se van a pasar entre funciones.

Compruebe su nivel de comprensión de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 6.3

1. ¿Harían los Programas 6.4 y 6.5 lo mismo si se usara `scanf()` en lugar de `gets()`?
2. ¿Por qué no es necesario especificar el tamaño de una cadena de caracteres cuando se pasa una cadena como argumento a una función?
3. En el Programa 6.6, ¿qué ocurriría durante la ejecución de `misimo_tamaño` si alguna de las cadenas no tuviera el carácter terminador nulo?

6.4. TRABAJO CON ELEMENTOS DE CADENAS DE CARACTERES

En esta sección se examinan varias funciones de biblioteca para leer elementos de cadenas de caracteres, examinarlos e incluso convertirlos de una forma a otra. Todas ellas son útiles en programas que aceptan texto de los usuarios.

Un carácter cada vez

Una función útil en C, es la denominada `getchar()`, que permite leer un carácter cada vez desde la entrada. Como verá, la ventaja de usar esta función es que se puede comprobar cada carácter leído para determinar si un carácter es una letra del alfabeto, un número o cualquier otro tipo de entrada como un signo de puntuación. El uso de esta función puede verse en el Programa 6.7.

Programa 6.7 `#include <stdio.h>`

```
main()
{
    char car;

    printf("Deme un numero => ");
    while((car = getchar()) != '\n');
}
```

La función `getchar()` lee caracteres de la entrada continuamente hasta que se tecllea el retorno de carro. Cuando ocurre esto, el bucle `while` termina. Por ejemplo, el usuario del programa puede introducir una cadena de caracteres hasta que se encuentre un marcador de nueva línea.

Comprobación de los caracteres

Hay varios tipos de funciones de biblioteca en C que permiten comprobar el tipo de carácter que ha recibido un programa C como entrada. La Tabla 6.1 muestra dichas funciones.

Tabla 6.1. Clasificación de caracteres en C en <ctype.h>

Función	Significado (Devuelve un valor no nulo si el carácter satisface la prueba, si no devuelve cero)	Ejemplo (ch es el carácter siendo probado)
isalnum()	Alfanumérico	if (isalnum (ch) != 0) printf ("%c es alfanumerico \n", ch);
isalpha()	Alfabético	if (isalpha (ch) != 0) printf ("%c es alfabetico \n", ch);
isctrl()	Carácter de control	if (isctrl (ch) != 0) printf ("%c es un ch de control \n", ch);
isdigit()	Dígito	if (isdigit (ch) != 0) printf ("%c es un dígito \n", ch);
isgraph()	Dice si el carácter es imprimible Excluye el caracter espacio	if (isgraph (ch) != 0) printf ("%c es imprimible \n", ch);
islower()	Dice si es un carácter de las minúsculas	if (islower (ch) != 0) printf ("%c es minuscula \n", ch);
isprint()	Dice si el carácter es imprimible Incluye el carácter espacio	if (isprint (ch) != 0) printf ("%c es imprimible \n", ch);
ispunct()	Dice si el carácter está entre los de puntuación	if (ispunct (ch) != 0) printf ("%c es de puntuacion \n", ch);
isspace()	Espacio	if (isspace (ch) != 0) printf ("%c es un espacio \n", ch);
isupper()	Dice si es un carácter de las mayúsculas	if (isupper (ch) != 0) printf ("%c es mayuscula \n", ch);
isxdigit()	Dígito hexadecimal	if (isxdigit (ch) != 0) printf ("%c es un dígito hexadecimal \n", ch);

El Programa 6.8 muestra la aplicación de una de las funciones de comprobación de caracteres.

```

Programa 6.8 #include <stdio.h>
#include <ctype.h>

main()
{
    char car;

    printf("Introduzca una sentencia => \n");
    while ((car = getchar()) != '\r')
        printf("%d", isalpha(car));
}

```

La ejecución del Programa 6.8 daría lo siguiente:

```

Introduzca una sentencia =>
<Aprenda C> -- tecleado por el usuario pero no impreso
122222201 -- impreso a medida que el usuario introduce caracteres

```

Como puede ver en la salida, la función `isalpha()` devuelve 0 cuando el carácter no es una letra del alfabeto (como el espacio en blanco), 1 para cualquier carácter en mayúsculas y 2 para cualquier carácter en minúsculas.

Más sobre comprobación de caracteres

Lo mismo que el Programa 6.8 comprueba que los caracteres estén en el alfabeto, se puede comprobar que sean números o cualquier otro tipo de caracteres de los presentados en la Tabla 6.1.

El Programa 6.9 pide al usuario una cadena de caracteres e imprime la cadena sin los signos de puntuación ni los números. Además, los caracteres alfabéticos en minúscula se convierten en mayúsculas. Esta técnica es útil en una aplicación de procesamiento de textos donde se hagan comprobaciones ortográficas eliminando previamente los caracteres no alfabéticos. Poner todos los caracteres en mayúsculas permite que la búsqueda y la comparación con el diccionario de referencia sea más sencilla.

```

Programa 6.8 #include <stdio.h>
              #include <ctype.h>

main()
{
    char cadena[80];
    int i = 0;

    printf("Introduzca una cadena de texto que incluya dígitos");
    printf(" y signos de puntuacion: \n");
    gets(cadena);
    while(cadena[i] != '\0')
    {
        if ((0 == ispunct(cadena[i])) && (0 == isdigit(cadena[i])))
        {
            if(0 != islower(cadena[i]))
                cadena[i] = cadena[i] & 0xdf;
            printf("%c", cadena[i]);
        }
        i++;
    }
}

```

Una ejecución de muestra del Programa 6.9 es como sigue:

```

Introduzca una cadena de texto que incluya dígitos y signos de
puntuacion:
Hola agente 99, soy Max! Donde está el jefe?
HOLA AGENTE SOY MAX DONDE ESTA EL JEFE

```

Busqueda de números

Cuando el usuario introduce una cadena numérica desde el teclado, los dígitos del número se almacenan como códigos de caracteres ASCII, no como dígitos numéricos reales. Esto hace necesario recorrer la cadena y convertir cada código ASCII numérico en su número decimal asociado y combinar todos los dígitos en un único número. El Programa 6.10 hace esto para enteros con signo. El formato de un entero con signo es el siguiente:

[+ o -] [dígitos]

donde el signo + o - es opcional (la no existencia de signo indica un entero positivo). Debe haber al menos un dígito en la parte de [dígitos]. Otros tipos de números tienen formatos similares, aunque más complicados, como por ejemplo:

[+ o -][dígitos].[dígitos] (números reales)
 [+ o -][dígitos].[dígitos][e o E][+ o -] [dígitos] (números científicos)

El Programa 6.10 pide al usuario que introduzca un entero con signo, examina la cadena de caracteres de entrada y construye en valor entero resultante a partir de los códigos ASCII de los caracteres de la cadena recibida. Observe que no hay ningún tipo de comprobación de error para asegurar que los datos leídos se ajustan al formato del número.

```
Programa 6.10 #include <stdio.h>
#include <ctype.h>

main()
{
    char cadena[10];
    int i = 0;
    int signo = 1, numero = 0;

    printf("Introduzca un entero con signo => ");
    gets(cadena);
    if('-' == cadena[0])
    {
        signo = -1;
        i++;
    }
    if('+' == cadena[0])
        i++;
    while(cadena[i] != '\0')
    {
        numero *= 10;
        numero += cadena[i] - 0x30;
        i++;
    }
}
```

```

    numero *= signo;
    printf("Introdujo el numero %d.\n", numero);
}

```

Una forma alternativa de convertir a número entero una cadena de caracteres es usar la función `atoi()` (ASCII a entero) de la biblioteca `<stdlib.h>`. Esta función proporciona una forma sencilla de convertir códigos ASCII a enteros. El Programa 6.11 muestra el uso de `atoi()`.

Programa 6.11

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    char cadena[10];
    int numero;

    printf("Introduzca un entero con signo => ");
    gets(cadena);
    numero = atoi(cadena);
    printf("Introdujo el numero %d.\n", numero);
}

```

Otras funciones de `<stdlib.h>` relacionadas son:

- `atof()` ASCII a real.
- `atol()` ASCII a entero largo.

Aunque estas funciones de biblioteca permiten realizar conversiones de forma sencilla, escribir sus propias funciones de conversión es un buen reto de programación (con el beneficio adicional de poder incluir comprobación de errores).

Compruebe su nivel de comprensión de esta sección intentando hacer los ejercicios de repaso.

Repaso de la Sección 6.4

1. Explique qué hace `getchar()`.
2. ¿Cómo se clasifican los caracteres de C? Ponga ejemplos.
3. ¿Imprimirá el Programa 6.9 algún carácter además de los alfabéticos?
4. ¿Puede reescribirse el Programa 6.9 con una función de `<ctype.h>` distinta y seguir mostrando sólo caracteres alfabéticos?
5. ¿Qué ocurre si el usuario introduce 40000 durante la ejecución del Programa 6.10?

6.5. FUNCIONES PARA MANEJAR CADENAS DE CARACTERES

Los ejemplos de programación vistos hasta ahora en este capítulo están relacionados con operaciones sobre cadenas de caracteres hechas carácter a carácter. Desde un punto de vista distinto cabe preguntarse, ¿qué operaciones para manejar cadenas de caracteres *completas* se podrían necesitar? Algunos ejemplos podrían ser:

- Hallar la longitud de una cadena de caracteres.
- Combinar dos cadenas de caracteres.
- Comparar dos cadenas de caracteres.
- Buscar una cadena (o subcadena) dentro de una cadena de caracteres.

C proporciona estas operaciones, y muchas más, a través de las funciones definidas en el archivo `include string.h`. Un subconjunto de dichas funciones se muestra en la Tabla 6.2. En esta sección se examinarán el funcionamiento y el uso de todas las funciones mostradas en dicha tabla. En algunos casos, se usarán dos programas de ejemplo para la misma función, sirviendo el segundo programa para mostrar cómo se puede implementar en C la función descrita.

`strlen()`

La función `strlen()` calcula la longitud de una cadena de caracteres. La longitud de una cadena de caracteres es un entero que indica el número de caracteres que forman parte de la cadena, excluyendo el carácter nulo del final. Por ejemplo, para una cadena definida como `char alpha[] = "abcdefghijklmnopqrstuvwxyz"` hará `strlen(alpha)` y devolverá un valor de 26. El Programa 6.12 ilustra el uso de `strlen()`.

Tabla 6.2. Funciones de cadenas de caracteres disponibles en `<string.h>`

Función	Significado
<code>strlen()</code>	Longitud de la cadena de caracteres.
<code>strcat()</code>	Concatenación de cadenas de caracteres.
<code>strncat()</code>	Concatenación de <i>n</i> caracteres.
<code>strcmp()</code>	Comparación de cadenas de caracteres.
<code>strncmp()</code>	Comparación de <i>n</i> caracteres.
<code>strchr()</code>	Busca un carácter dentro de una cadena de caracteres.
<code>strrchr()</code>	Busca un carácter dentro de una cadena de caracteres (desde el final).
<code>strstr()</code>	Busca una subcadena dentro de una cadena de caracteres.
<code>strpbrk()</code>	Busca la primera ocurrencia de cualquier carácter de una subcadena dentro de una cadena de caracteres.
<code>strtod()</code>	Convierte una cadena de caracteres a un real.
<code>strtol()</code>	Convierte una cadena de caracteres a un entero largo.
<code>strtoul()</code>	Convierte una cadena de caracteres a un entero largo sin signo.

Programa 6.12

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

int long_tira(char texto[]);
main()
{
    char alfa[] = "abcdefghijklmnopqrstuvwxyz";

    printf("La cadena \"%s\" contiene %d caracteres."
           , alfa, strlen(alfa));
    printf("La cadena \"%s\" contiene %d caracteres."
           , alfa, long_tira(alfa));
}
int long_tira(char texto[])
{
    int numcar = 0;

    while (texto[numcar] != '\0')
        numcar++;
    return(numcar);
}
```

Es importante usar la directiva `#include <string.h>` para que la función `strlen()` esté disponible durante la compilación. El Programa 6.12 muestra el siguiente mensaje:

```
La cadena "abcdefghijklmnopqrstuvwxyz" contiene 26 caracteres.
La cadena "abcdefghijklmnopqrstuvwxyz" contiene 26 caracteres.
```

Tenga en cuenta que las comillas dobles se imprimen porque se especificaron como `\` en la sentencia `printf()`, pero no por sí mismas.

Además en el Programa 6.12 se implementa una función `long_tira` que hace las operaciones equivalentes a las de la función `strlen()`. Para ello cuenta el número de caracteres de la cadena, empezando en el elemento `[0]`, hasta que se encuentra el carácter nulo `'\0'`. Si el elemento `[0]` contiene `'\0'`, la longitud de la cadena es cero.

Observe que se usa el nombre `long_tira` para la función alternativa para evitar confusiones con el nombre de la función de manejo de caracteres predefinida.

strcat() y strncat()

Las funciones `strcat()` y `strncat()` se usan para *concatenar* dos cadenas de caracteres. Cuando se concatenan dos cadenas de caracteres, los contenidos de la segunda cadena se copian al final de la primera, como puede verse en la Figura 6.4. En este caso, se usa `strcat()` para combinar las dos cadenas. Es importante que la primera cadena tenga definida una longitud suficiente como para almacenar la cadena concate-

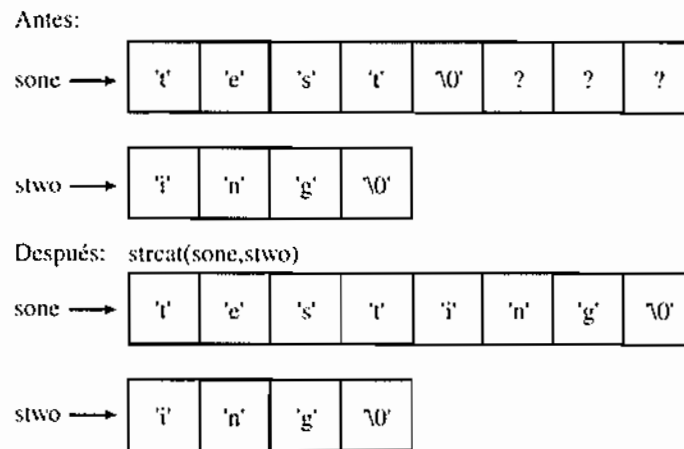


Figura 6.4. Concatenando dos cadenas de caracteres.

nada. Como puede verse en la Figura 6.4, la función `strcat()` necesita dos argumentos: los nombres de las cadenas a concatenar. El primer argumento será el destino de la nueva cadena.

Cuando no se necesite concatenar la segunda cadena completa, se debería usar la función `strncat()`. Esta función concatena a la primera cadena sólo los n primeros caracteres de la segunda cadena. Como la función necesita el valor de n , `strncat()` tiene tres argumentos:

```
strncat(str1, str2, n)
```

Suponga que `str1` y `str2` se definen de la siguiente manera:

```
char str1[20] = "¿Dónde esta";
char str2[] = "Juan?";
```

¿Qué tiene `str1` después de ejecutar `strncat(str1, str2, 3)`? Como el valor de n es 3, sólo se concatenarán los tres primeros caracteres de `str2`, dando como cadena resultante «¿Dónde esta Jua».

El Programa 6.13 muestra como se usa la función `strcat()`.

Programa 6.13

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

main()
{
    char cadena_a[80] = "La programacion en C";
    char cadena_b[80] = " es divertida";

    strcat(cadena_a,cadena_b);
    printf("La nueva cadena de caracteres es \"%s\\\".",cadena_a);
}
```

Observe que el tamaño definido para las cadenas en el Programa 6.13 es de 80 caracteres, aunque las cadenas realmente usadas tienen muchos menos caracteres. Esta planificación adelantada ayuda a evitar futuros problemas cuando múltiples concatenaciones podrían originar cadenas de longitudes mayores. La salida del Programa 6.13 es el mensaje siguiente:

La nueva cadena de caracteres es "La programacion en C es divertida".

El Programa 6.14 muestra la implementación de la función `strcat()`.

Programa 6.14

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

int concat_cadenas(char tiral[], char tira2[]);
main()
{
    char cadena_a[80] = "La programacion en C";
    char cadena_b[80] = " es divertida";

    concat_cadenas(cadena_a,cadena_b);
    printf("La nueva cadena de caracteres es \"%s\".",cadena_a);
}
int concat_cadenas(char tiral[], char tira2[])
{
    int pos,longitud,j;

    pos = tiralen(tiral);
    longitud = tiralen(tira2);
    for(j = 0; j < longitud; j++)
    {
        tiral[pos] = tira2[j];
        pos++;
    }
    tiral[pos] = '\0';
}
```

Una vez más, se ha cambiado el nombre a `concat_cadenas` para evitar confusiones. La función `concat_cadenas` usa `strlen()` para determinar la posición final de la primera cadena. Luego vuelve a usar `strlen()` para obtener la longitud de la segunda cadena y proceder a copiar los caracteres de la misma al final de la primera cadena. Es necesario poner el carácter nulo al final de la primera cadena para garantizar su posición final.

strcmp() y strncmp()

Las funciones `strcmp()` y `strncmp()` se usan para comparar dos cadenas de caracteres. La comparación se hace carácter a carácter. Cada función devuelve un valor que depende del resultado de la comparación. El valor entero devuelto es:

- 0 si las cadenas son idénticas (iguales caracteres y longitud).
- > 0 Si la primera cadena precede alfabéticamente a la segunda.
- < 0 si la segunda cadena precede alfabéticamente a la primera.

Por ejemplo, considere esta corta lista de nombres:

"Doe"
 "Morris"
 "Morrison"
 "Smith"

Los cuatro nombres están ordenados en orden alfabético. Por tanto, "D" está antes que "M" y "M" está antes que "S". Se devolverán valores positivos para `strcmp("Doe", "Morris")` y `strcmp("Morris", "Smith")`. En el caso de "Morris" y "Morrison", ambos nombres coinciden en los seis primeros caracteres. "Morrison" estaría después porque tiene más caracteres que "Morris". La comparación `strcmp("Morris", "Morrison")` devuelve un valor positivo y `strcmp("Morrison", "Morris")` devuelve un valor negativo. Cuando las cadenas están ordenadas de esta forma se dice que están en *orden lexicográfico*. Por tanto, `strcmp()` devuelve un valor relacionado con el orden lexicográfico de sus dos argumentos.

La función `strncmp()` necesita un tercer argumento, *n*, que especifica el número de caracteres a comparar. Así, "Morris" y "Morrison" parecen idénticos cuando se usa `strncmp("Morris", "Morrison", 6)`.

El Programa 6.15 muestra cómo se usa la función `strcmp()`. Para ello, define cuatro cadenas de caracteres que son comparadas de tres formas distintas para ilustrar lo que hace `strcmp()`.

Programa 6.15 `#include <stdio.h>`
`#include <string.h>`

```

void comparar(char tira1[], char tira2[]);
main()
{
    char cadena_a[] = "comprador";
    char cadena_b[] = "compras";
    char cadena_c[] = "comprador";
    char cadena_d[] = "bola";

```

```

        comparar(cadena_a,cadena_b);
        comparar(cadena_a,cadena_c);
        comparar(cadena_a,cadena_d);
    }
void comparar(char tira1[], char tira2[])
{
    int valor;

    valor = strcmp(tira1,tira2);
    if (valor == 0)
        printf("\"%s\" es 'igual que' \"%s\".\n",tira1,tira2);
    else
        if (valor < 0)
            printf("\"%s\" es 'menor que' \"%s\".\n",tira1,tira2);
        else
            printf("\"%s\" es 'mayor que' \"%s\".\n",tira1,tira2);
}

```

La salida del Programa 6.15 es la siguiente:

```

"comprador" es menor que "compras"
"comprador" es igual que "comprador"
"comprador" es mayor que "bola"

```

Se puede hacer una función que emule el comportamiento de `strcmp()`. Si las dos cadenas tienen distinta longitud, la más larga se usa como muestra en el bucle de comparación. La comparación de elementos continúa mientras las dos cadenas tengan caracteres idénticos en cada posición.

strchr() y strrchr()

Ambas funciones, `strchr()` y `strrchr()`, buscan un carácter específico dentro de una cadena de caracteres. Necesitan dos argumentos. El primero es la cadena dentro de la cuál se va a buscar. El segundo es el carácter a buscar. Ambas funciones devuelven un puntero a la posición del carácter dentro de la cadena, si lo han encontrado, o un puntero nulo si no lo han encontrado. `strchr()` devuelve la primera ocurrencia del carácter dentro de la cadena. `strrchr()` devuelve la última ocurrencia del carácter dentro de la cadena. El Programa 6.16 muestra cómo se usa `strchr()`. La función `search()` usa `strchr()` para determinar la posición del carácter de búsqueda y luego comprueba la posición para ver si es nula o hubo éxito en la búsqueda.

Programa 6.16 `#include <stdio.h>`
`#include <string.h>`
`#include <ctype.h>`

```

void buscar(char texto[], char letra);
main()
{
    char alfa[] = "abcdefghijklmnopqrstuvwxyz";

    buscar(alfa, 'e');
    buscar(alfa, 'E');
}
void buscar(char texto[], char letra)
{
    int posicion;

    posicion = strchr(texto, letra);
    if (posicion == '\0')
        printf("La letra %c no esta en la cadena.\n", letra);
    else
        printf("La letra %c esta almacenada en la direccion %X de
            memoria.\n", letra, posicion);
}

```

La ejecución del Programa 6.16 da la siguiente salida:

```

La letra e esta almacenada en la direccion E0E de memoria.
La letra E no esta en la cadena.

```

La prueba anterior muestra que `strchr()` es *sensible a mayúsculas y minúsculas*. Por tanto, los caracteres en mayúsculas y en minúscula son distintos entre sí. Esto tiene sentido, porque ambos tipos de caracteres tienen códigos ASCII únicos asignados.

strstr()

La función `strstr()` busca la primera ocurrencia de una subcadena dentro de una cadena. Si se encuentra la subcadena, la función `strstr()` devuelve un puntero al principio de la subcadena. Si no se encuentra, devuelve un puntero nulo. Por ejemplo, `strstr("si, por supuesto", "por")` devuelve un puntero a la subcadena "por" dentro de "si, por supuesto", mientras que `strstr("si, por supuesto", "¿cómo?")` devuelve un puntero nulo, porque "¿cómo?" no se encuentra en ninguna parte.

El Programa 6.17 usa `strstr()` para buscar todas las apariciones de la palabra "el" en un texto de muestra.

Programa 6.17 `#include <stdio.h>`
`#include <string.h>`

```

main()
{
    char texto[] = "Es importante que este bloque de texto "
        "contenga la palabra 'el' tantas veces como sea "
        "posible, ya que la funcion strstr() contara "
        "el numero de veces que aparece 'el'. "
        "Se trata de una simple operacion matematica. "
        "Cuantas veces aparece el termino 'el'?";
    char *puntero;
    int contador = 0;
    puntero = texto;
    do
    {
        puntero = strstr(puntero, "el");
        if(puntero != '\0')
        {
            contador++;
            puntero++;
        }
    } while(puntero != '\0');
    printf("La palabra 'el' aparece %d veces.\n", contador);
}

```

Cada vez que se llama a la función `strstr()` y se encuentra una subcadena, se usa el puntero devuelto para indicar dónde debería empezar la búsqueda en la siguiente llamada a `strstr()`. Este proceso continúa hasta que se devuelve un puntero nulo.

¿Puede pensar en qué forma podría ser útil la función `strstr()` en un corrector ortográfico de un procesador de textos?

strpbrk()

La función `strpbrk()` busca la primera aparición de cualquier carácter de una subcadena dentro de una cadena. Por ejemplo, `strpbrk("buenos días", "temporizador")` devolverá un puntero a la primera 'e' en "buenos días", porque ese es el primer carácter de "temporizador" que está también en "buenos días". Esta función es útil cuando es necesario saber rápidamente si una cadena de caracteres contiene un carácter determinado. Por ejemplo, si la cadena a examinar representa una expresión matemática como "5*(2-3)" puede ser necesario encontrar la posición de todos los números dentro de la misma. La sentencia

```
ptr = strpbrk(expr, "0123456789")
```

es una forma sencilla de hacerlo. Realizar el mismo trabajo con `strchr()` necesitaría múltiples sentencias C.

El Programa 6.18 muestra cómo se usa `strpbrk()` para quitar todos los signos de puntuación estándares de una cadena de caracteres.

Programa 6.18 `#include <stdio.h>`
`#include <string.h>`

```
main()
{
    char texto[80];
    char *puntero;

    printf("Introduzca una cadena con signos de puntuacion:\n");
    gets(texto);
    puntero = texto;
    while(puntero != '\0')
    {
        puntero = strpbrk(puntero,".,!;'\"?~-");
        if(puntero != '\0')
            *puntero = ' ';
    }
    printf("\n%s", texto);
}
```

Los signos de puntuación buscados en el Programa 6.18 son ". , ! ; ' \" ? ~ - ". Cuando se encuentra alguno se sustituye por un blanco. Un ejemplo de ejecución haría lo siguiente:

Introduzca una cadena de caracteres con signos de puntuacion:
 !Hace mucho sol hoy! ?No lo cree?

Hace mucho sol hoy No lo cree

Esta técnica es útil cuando se desea eliminar información sin sentido, poco importante o datos que no se quieren procesar en un momento determinado. Si se estuviera cifrando un texto, la puntuación sería probablemente eliminada porque podría dar pistas acerca de la longitud de las palabras o las frases.

strtod(), strtol() y strtoul()

Las funciones `strtod()`, `strtol()` y `strtoul()` permiten convertir cadenas de caracteres a números de tipo `double` y `long`. Esta funcionalidad es similar a la de las funciones `atoi()` y `atof()` vistas en la sección anterior. Estas funciones permiten la conversión de números más largos.

Compruebe su nivel de comprensión de esta sección haciendo los siguientes ejercicios de repaso.

Repaso de la Sección 6.5

1. ¿Qué sentencia `include` se necesita para tener disponibles las funciones de cadenas de caracteres?
2. Enumere los operadores de cadenas de caracteres más frecuentes.
3. ¿Por qué es necesario para `strlen()` encontrar el carácter nulo?
4. ¿Qué requeriría una función de sustitución de cadenas de caracteres? Por ejemplo, `strsub("abcde", "bc", "hola")` daría "aholade".
5. ¿Qué requisito debe cumplir la longitud de la primera cadena recibe por argumento `strcat()`?
6. ¿Qué significa que `strchr()` sea sensible a mayúsculas y minúsculas?
7. ¿Que funciones de cadenas de caracteres son sensibles a mayúsculas y minúsculas? Puede intentar hacer algunos ejemplos antes de responder a esta pregunta.

6.6. ORDENACIÓN DE CADENAS DE CARACTERES

En esta sección se examina una forma de ordenar un conjunto de cadenas de caracteres en orden alfabético. El conjunto de cadenas está almacenado en memoria como una matriz *bidimensional* de caracteres. Una matriz bidimensional con 5 nombres de una longitud máxima de siete caracteres cada uno (la posición octava se reserva para el carácter nulo), se define como sigue:

```
char nombres [5][7];
```

Este tipo de matriz también se conoce como **matriz rectangular**.

Iniciar la matriz bidimensional es muy sencillo. Puede hacerse de la siguiente manera:

```
char nombres[5][7] = { {'s', 'u', 's', 'a', 'n', 'a'},
                        {'m', 'i', 'g', 'u', 'e', 'l'},
                        {'j', 'a', 'v', 'i', 'e', 'r'},
                        {'j', 'u', 'a', 'n'},
                        {'j', 'a', 'v', 'i' } };
```

Observe que cada cadena de la matriz está situada entre un par de llaves `{ }` y que la matriz completa está también delimitada por llaves. Cuando un nombre tiene menos de siete caracteres, las posiciones restantes se rellenan automáticamente con caracteres nulos cuando se usa este método de inicialización, como puede verse en la Figura 6.5.

Como puede verse en el Programa 6.19, cada nombre dentro de la matriz puede referenciarse usando como índice el número de la fila que ocupa, como `nombres[j]` o `nombres[k]`. Por ejemplo, la función `strcpy()` necesita sólo un subíndice como los anteriores para acceder al nombre completo.

Programa 6.19 `#include <stdio.h>`
`#include <string.h>`
`#define NUM 5`

```

main()
{
    char nombres[5][7] = { {'s', 'u', 's', 'a', 'n', 'a'},
                           {'m', 'i', 'g', 'u', 'e', 'l'},
                           {'j', 'a', 'v', 'i', 'e', 'r'},
                           {'j', 'u', 'a', 'n'},
                           {'j', 'a', 'v', 'i'} };
    char nombre_aux[7];
    int j,k;

    printf("La lista original es:\n");
    for(j = 0; j < NUM; j++)
        printf("%s\n",nombres[j]);
    for(j = 0; j < NUM - 1; j++)
        for(k = j + 1; k < NUM; k++)
            if(strcmp(nombres[j],nombres[k]) > 0)
            {
                strcpy(nombre_aux,nombres[j]);
                strcpy(nombres[j],nombres[k]);
                strcpy(nombres[k],nombre_aux);
            }
    printf("\nLa lista ordenada es:\n");
    for(j = 0; j < NUM; j++)
        printf("%s\n",nombres[j]);
}

```

La técnica usada en el Programa 6.19 para ordenar la lista de nombres se llama *ordenación de burbuja*. Una **ordenación de burbuja** es realmente un bucle anidado que hace $n - 1$ pasadas sobre una lista de n elementos, cambiando el elemento X por el $X + 1$ durante cada pasada si el elemento X es «mayor» que el elemento $X + 1$. Por ejemplo, la lista de nombres definida en el Programa 6.19 comienza por "susana". Como "susana" es lexicográficamente mayor que los otros cuatro nombres de la lista, al final de la primera pasada "susana" estará en la última posición de la nueva lista reordenada. Una ejecución de prueba del Programa 6.19 daría lo siguiente:

La lista original es:

```

susana
miguel
javier
juan
javi

```

La lista ordenada es:

```

javi
javier
juan
miguel
susana

```

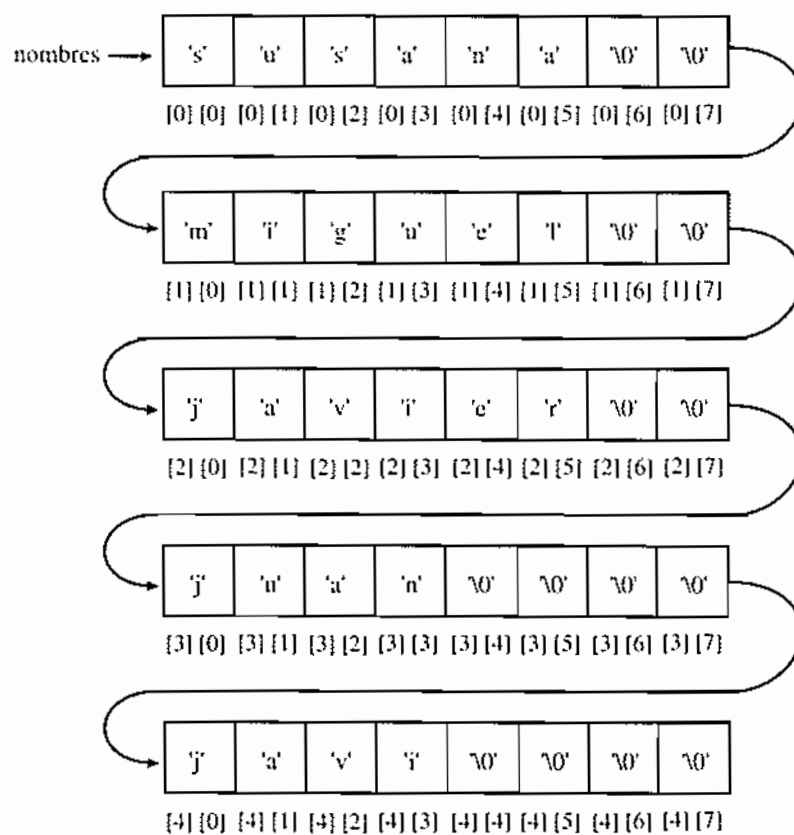


Figura 6.5. Almacenando una cadena de caracteres bidimensional en memoria.

La ordenación de burbuja es una de las técnicas más sencilla de ordenación y también una de las más ineficientes. Para una lista de 100 nombres, este método necesitará, desafortunadamente, casi 10.000 comparaciones. En el Capítulo 7 veremos otras técnicas de ordenación más eficientes.

Una forma alternativa de definir una matriz bidimensional de caracteres para el programa de ordenación sería la siguiente:

```
char *nombres[5] = { ("susana "),
                     ("miguel "),
                     ("javier "),
                     ("juan  "),
                     ("javi  ") };
```

Este método de inicialización define también una matriz rectangular, pero necesita que el usuario introduzca el número apropiado de blancos detrás de cada nombre para garantizar que cada cadena ocupa la misma cantidad de memoria. Si no fuera este el caso, las funciones de biblioteca (`strcpy`, `strcmp`) darían resultados impredecibles. Este mismo método de inicialización puede usarse en el Programa 6.19 sin que cambie en nada el resto del programa.

Cuando no se usan los blancos del final, como en este ejemplo:

```
char *nombres[5] = { {"susana"},
                     {"miguel"},
                     {"javier"},
                     {"juan"},
                     {"javi"} };
```

se obtiene como resultado un **vector irregular**, como puede verse en la Figura 6.6. El vector irregular no es rectangular. Necesita solamente 28 posiciones de memoria, mientras que el vector rectangular definido como `nombres[5][8]` necesita 40. Por lo tanto, los vectores irregulares son un método de almacenamiento más eficiente que los rectangulares, pero también es más difícil trabajar con ellos. Considere el código de intercambio del Programa 6.19:

```
strcpy(nombre_aux, nombres[j]);
strcpy(nombres[j], nombres[k]);
strcpy(nombres[k], nombre_aux);
```

Durante el intercambio de "javier" con "juan", se obtendría este desafortunado resultado:

```
nombres[3] = "juaner"
nombres[4] = "javi"
```

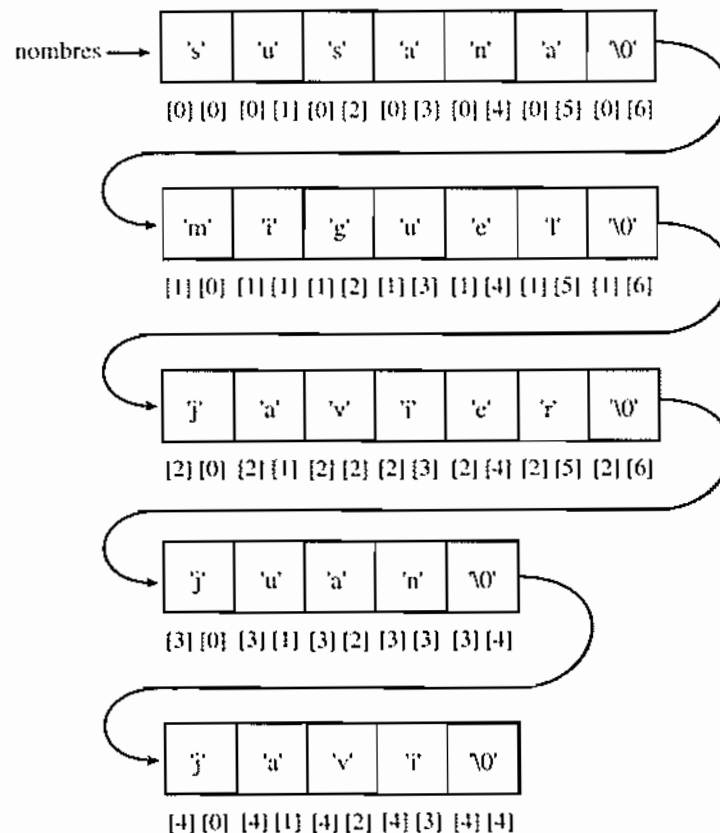


Figura 6.6. Una matriz irregular de caracteres.

Este resultado se debe a que los punteros internos a los componentes de nombres se definen cuando se inicia el vector. Por tanto, las longitudes individuales de cada cadena de caracteres se fijan para valores específicos (7 para "susana" incluyendo el carácter nulo, 6 para "miguel" y así para el resto). Sobre escribir cualquiera de estas cadenas de caracteres puede causar que una cadena se solape con las posiciones de memoria de la siguiente, como puede verse en `nombres[4]`. Un buen programador debería recordar las fortalezas y debilidades de usar la definición de un vector encima de la de otro.

Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 6.6

1. ¿Cuántas posiciones de memoria se necesitan para una matriz rectangular definida como sigue: `matriz[7][10]`?
2. ¿Cuántas comparaciones son necesarias en la ordenación por burbuja del Programa 6.19?
3. Especifique todas las matrices de nombres para cada pasada del bucle exterior del Programa 6.19.
4. ¿Por qué hace el método de ordenación por burbuja n^2 comparaciones aproximadamente para n elementos?
5. ¿Cuáles son las fortalezas y debilidades de las matrices rectangulares e irregulares?
6. Explique cómo puede producir la función `strcpy()` la cadena «javiju».

6.7. PROGRAMA DE APLICACIÓN: FORMATEADOR DE TEXTO

Presentación

El manejo de cadenas de caracteres es una parte tan importante de muchas aplicaciones que se ha decidido dedicar esta sección a una aplicación basada en el manejo de dichas cadenas. La aplicación que se presenta es un formateador de texto. Los formateadores de texto están habitualmente disponibles en la mayoría de los procesadores de texto. El propósito de este formateador de texto es ajustar un bloque de texto, para determinar la forma en que será mostrado por pantalla o impreso, mediante la inserción del número apropiado de blancos entre palabras en cualquiera de las líneas de forma que todas ellas se ajusten perfectamente a los márgenes derecho e izquierdo predefinidos. Por ejemplo, considere este bloque de texto de entrada:

El microcomputador es una parte importante de cualquier curriculum de ingeniería. La maquina en si misma puede usarse para enseñar muchos temas del curso relacionados con el hardware, tales como interfaz de usuario, diseño de memoria y fundamentos de microprocesadores. Los paquetes de software disponibles actualmente ofrecen a los estudiantes una amplia variedad de aplicaciones, incluyendo graficos, programacion, y diseño analogico digital.

Con todo ello, el microcomputador representa una ayuda apreciable para la enseñanza y la productividad.

Puede ser necesario reformatear el texto para que cada línea contenga un máximo de 45 caracteres —posiblemente debido a requisitos de la impresora (ancho del papel especial cargado en la impresora) o a alguna otra preferencia personal. Cuando se reformatea el bloque de texto en líneas de 45 caracteres, se obtiene lo siguiente:

El microcomputador es una parte importante de cualquier curriculum de ingeniería. La maquina en si misma puede usarse para enseñar muchos temas del curso relacionados con el hardware, tales como interfaz de usuario, diseño de memoria y fundamentos de microprocesadores. Los paquetes de software disponibles actualmente ofrecen a los estudiantes una amplia variedad de aplicaciones, incluyendo graficos, programacion, y diseño analogico digital. Con todo ello, el microcomputador representa una ayuda apreciable para la enseñanza y la productividad.

Observe como algunas líneas (como la quinta empezando por abajo) que contienen menos de 45 caracteres se expanden con varios blancos insertos entre cada dos palabras. Este bloque de texto formateado parece más bonito que el anterior, donde cada línea terminaba con una longitud distinta.

El programa de aplicación que se muestra en esta sección crea este tipo de formato de salida.

El problema

El problema de este tipo de formato es doble. Primero, cómo determinar cuántas palabras caben en una línea de texto y, segundo, cada línea mostrada puede requerir un número distinto de caracteres para rellenar completamente los márgenes. Por ejemplo, considere la sexta línea empezando por arriba. En su forma original, antes de la expansión, tiene este aspecto:

```
-->microprocesadores. Los paquetes de software <--
```

¿Por qué no se incluyó en la línea formateada la palabra disponibles? La respuesta es sencilla. Los marcadores --> y <-- indican los márgenes dentro de los cuáles se admiten 45 caracteres. La línea de texto contiene 43 caracteres, por lo que faltan 2 blancos para su expansión. La siguiente palabra «disponibles» necesita 10 caracteres y sólo quedan 2 disponibles. Por tanto, es necesario expandir la línea como está y poner la palabra «disponibles» al principio de la línea siguiente. El problema ahora es el siguiente: ¿cómo insertar blancos entre palabras de una línea de texto? Una solución es avanzar por el texto hasta que se encuentra un blanco e insertar un nuevo blanco,

luego se avanza hasta el final de la siguiente palabra y se inserta un nuevo blanco. Este proceso se repite hasta que se ha insertado el número de blancos deseado. La sexta línea es pues expandida de la siguiente manera:

```
-->microprocesadores. Los paquetes de software <--
-->microprocesadores. Los paquetes de software <--
-->microprocesadores. Los paquetes de software<--
```

Aunque puede haber otras técnicas para hacer la expansión, ésta es la que se usa en el programa de ejemplo.

Desarrollo del algoritmo

Los pasos a seguir en la aplicación de formateo de texto son los siguientes:

1. Cargar nuevas palabras en el almacén intermedio de salida hasta que no quepan más palabras.
2. Expandir el almacén de salida hasta los márgenes.
3. Imprimir el almacén de salida.
4. Repetir los pasos 1 a 3 hasta que no queden más palabras.
5. No expandir la última línea de texto almacenada en el almacén de salida

En cada paso es necesario examinar el texto de entrada y el de salida. Se utilizarán todas las funciones de manejo de cadena de caracteres que sea necesario.

El proceso global

El formateador de texto debe hacer los siguientes pasos para cada línea de texto a formatear:

1. Empezar con un almacén intermedio de salida vacío (relleno de nulos).
2. Determinar la longitud de la siguiente palabra del texto.
3. Si ha sitio en el almacén de salida para la siguiente palabra, copiarla y volver al paso 2.
4. Si no hay espacio suficiente para la siguiente palabra, expandir el almacén de salida, imprimirlo y rellenar un nuevo almacén de salida con la palabra siguiente.
5. Repetir los pasos 1 a 4 hasta que no queden más palabras disponibles.
6. Si la memoria intermedia de salida no está vacía cuando termina el texto de entrada, imprimir el almacén de salida sin expansión.

Este proceso se implementa mediante un cierto número de las siguientes funciones, que son llamadas desde `main()`:

```
void inicalmacen(char almacen[]);
int obtener(char datos[], int ptr, char palabra[]);
void cargapalabra(char almacen[], char palabra[]);
void expandir_linea(char almacen[], int como);
```

El código de `main()` es el siguiente:

```
main()
{
    espacio_restante = ANCHO;
    donde = 0;
    inicalmacen(almacener);
    do
    {
        siguiente = obtener(text, donde, siguientepalabra);
        if(espacio_restante >= strlen(siguientepalabra))
        {
            cargapalabra(almacener, siguientepalabra);
            donde = siguiente;
        }
        else
        {
            expandir_linea(almacener, EXP);
            inicalmacen(almacener);
            espacio_restante = ANCHO;
        }
    } while(0 != strlen(siguientepalabra));
    if(0 != strlen(almacener))
        expandir_linea(almacener, NOEXP);
}
```

El valor de `ANCHO` se predefine al número de caracteres entre los márgenes. La variable `donde` apunta a la posición actual del texto de entrada, que está almacenado en la cadena `text`. El almacén de salida se llama `almacen_salida` y es usado por `inicalmacen()`, `obtener()`, `cargapalabra()` y `expandir_linea()`. `EXP` y `NOEXP` significan expandir y no-expandir y son usados para controlar lo que hace `expandir_linea()` con el almacén de salida.

El Programa 6.20 muestra cómo se implementa el formateador de texto.

Programa 6.20

```
#include <stdio.h>
#include <string.h>
#define EXP 1
#define NOEXP 0
#define ANCHO 45

void inicalmacen(char almacen[]);
int obtener(char datos[], int ptr, char palabra[]);
void cargapalabra(char almacen[], char palabra[]);
void expandir_linea(char almacen[], int como);
```

```

char text[] =
    "El microcomputador es una parte importante de cualquier"
    " curriculum de ingenieria. La maquina en si misma"
    " puede usarse para enseñar muchos temas del curso"
    " relacionados con el hardware, tales como interfaz de"
    " usuario, diseño de memoria y fundamentos de"
    " microprocesadores. Los paquetes de software disponibles"
    " actualmente ofrecen a los estudiantes una amplia"
    " variedad de aplicaciones, incluyendo graficos,"
    " programacion, y diseño analogico digital."
    " Con todo ello, el microcomputador representa una"
    " ayuda apreciable para la enseñanza y la"
    " productividad.";
char alm_salida[80];
char siglientepalabra[80];
int espacio_restante, donde, siguiente;
main()
{
    espacio_restante = ANCHO;
    donde = 0;
    inicalmacen(alm_salida);
    do
    {
        siguiente = obtener(text, donde, siglientepalabra);
        if(espacio_restante >= strlen(siglientepalabra))
        {
            cargapalabra(alm_salida, siglientepalabra);
            donde = siguiente;
        }
        else
        {
            expandir_linea(alm_salida, EXP);
            inicalmacen(alm_salida);
            espacio_restante = ANCHO;
        }
    } while(0 != strlen(siglientepalabra));
    if(0 != strlen(alm_salida))
        expandir_linea(alm_salida, NOEXP);
}
void inicalmacen(char almacen[])
{
    int z;

    for(z = 0; z <= ANCHO; z++)
        almacen[z] = '\0';
}
int obtener(char datos[], int ptr, char palabra[])
{
    int i = 0;

```

```

/* Saltar blancos entre palabras */
while((datos[ptr] == ' ') && (datos[ptr] != '\0'))
    ptr++;

/* Copiar caracteres hasta el blanco o el NULO */
while((datos[ptr] != ' ') && (datos[ptr] != '\0'))
{
    palabra[i] = datos[ptr];
    i++;
    ptr++;
}
palabra[i] = '\0';      /* Marcar el fin de la cadena */
return ptr;
}

void cargapalabra(char almacen[], char palabra[])
{
    strcat(almacen,palabra);    /* Copiar la palabra a la salida */
    espacio_restante -= strlen(palabra); /* Ajustar el espacio
                                     restante del almacen de salida */
    if(ANCHO > strlen(almacen)) /* Está el almacén salida lleno? */
    {
        strcat(almacen," ");    /* Añadir blanco después de palabra */
        espacio_restante--;
    }
}

void expandir_linea(char almacen[], int como)
{
    int n,k,exp;

    if(como == 0)                /* ¿No expandir alm-salida? */
        printf("%s\n",almacen);
    {
        /* Llenar fin de alm_salida con NULOS */
        n = ANCHO;
        while((almacen[n] == '\0') || (almacen[n] == ' '))
        {
            almacen[n] = '\0';
            n -= 1;
        }

        /* Determinar el número de blancos a expandir */
        exp = ANCHO - strlen(almacen);

        n = 0;
        while(exp > 0) /* Mientras haya blancos que insertar... */
        {
            do          /* Saltar palabra actual*/
            {
                n++;
            } while((almacen[n] != ' ') && (almacen[n] != '\0'));
        }
    }
}

```

```

    /* A cero el índice de alm_salida si está al final */
    if(almacen[n] == '\0')
        n = 0;
    else
    {
        /* Si no, insertar un */
        do /* blanco en alm_salida */
        {
            n++;
        } while(almacen[n] == ' ');
        for(k = ANCHO - 1; k > n; k--)
            almacen[k] = almacen[k-1];
        almacen[n] = ' ';
        n++;
        exp--; /* Un blanco menos a insertar */
    }
}
printf("%s\n",almacen); /* Imprimir la salida expandida */
}

```

Conclusión

La aplicación para formatear textos necesita que el programador entienda cómo usar y manipular cadenas de caracteres. Sería muy conveniente que usted pensara en otras formas de conseguir el mismo propósito. Por ejemplo, en lugar de cargar una nueva palabra cada vez, podría elegir leer la línea de entrada hasta que se haya alcanzado el ancho del formato y cargar entonces el bloque completo de texto. También se puede hacer la expansión de otra forma, teniendo en cuenta el número de palabras de cualquier línea y usando el contador de palabras para calcular matemáticamente el número de blancos necesarios entre palabras.

Compruebe su nivel de comprensión de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 6.7

1. ¿Por qué es necesario el formateo de texto?
2. ¿Cómo se determina la longitud de la próxima palabra del texto de entrada?
3. ¿Qué ocurre si el almacén de salida está completamente lleno de texto?
4. ¿Qué ocurre cuando se cambia el valor ANCHO?
5. ¿Qué ocurre cuando se cambia el valor ANCHO a 10?
6. ¿Debería haber alguna restricción para el valor de ANCHO?
7. ¿Cómo se insertan los blancos en expandir_línea?
8. ¿Se puede usar vectores irregulares para hacer el formateo de texto más fácil?

6.8. PROGRAMAS DE APLICACIÓN ADICIONALES

Los programas de ejemplo que se muestran en esta sección enseñan cómo hacer muchas cosas interesantes y útiles con cadenas de caracteres. Se le recomienda que los estudie detalladamente para que pueda comprender completamente su funcionamiento.

Comprobador de ISBN

El Programa 6.21 comprueba un código de ISBN (International Standard Book Code) introducido por el usuario y determina si es una secuencia válida.

Programa 6.21

```
#include <stdio.h>
main()
{
    char entrada[10];
    int total = 0;
    int i, rem;

    printf("Introduzca los 9 primeros digitos del codigo ISBN => ");
    scanf("%s", entrada);

    for(i = 0; i < 9; i++)
        total += (entrada[i] - 0x30) * (i + 1);

    rem = total % 11;
    printf("El ultimo caracter deberia ser ");
    if(rem != 10)
        printf("%c", (rem + 0x30));
    else
        printf("X");
}
```

El formato de un código ISBN es el siguiente:

- Código de grupo (1 dígito).
- Código del editor (4 dígitos).
- Código del libro (4 dígitos).
- Carácter/dígito de control (1 carácter/dígito).

Por ejemplo, el código ISBN para un libro de ingeniería es:

0 6 7 5 2 0 9 9 3 5

El carácter o dígito de control se obtiene en dos pasos:

$$\begin{array}{r}
 0 \quad 6 \quad 7 \quad 5 \quad 2 \quad 0 \quad 9 \quad 9 \quad 3 \\
 \times 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
 \hline
 0 + 12 + 21 + 20 + 10 + 0 + 63 + 72 + 27 = 225
 \end{array}$$

En el paso 2, se divide la suma por 11, y se conserva el entero del resto. La división de 225 por 11 da un resto de 5. Este es el dígito de control calculado, que casa con el dígito de control en el código ISBN original. Si el resto es 10, se usa la letra X en su lugar. Por tanto:

0 6 7 5 2 0 9 9 3 X

es también un código de ISBN válido.

Una ejecución de prueba del Programa 6.21 daría:

```
Introduzca los 9 primeros dígitos del código ISBN => 067520772
El último carácter debería ser X
```

Observe que el Programa 6.21 convierte cada carácter ASCII introducido en su valor numérico asociado restándole 0x30 (30 en hexadecimal). Esto es posible porque los códigos ASCII de «0» a «9» van de 0x30 a 0x39.

Ejecute el Programa 6.21 con los códigos de ISBN de sus restantes libros de texto.

Contador de Vocales

El Programa 6.22 cuenta las vocales (a, e, i, o, u) de un texto. Deben contarse las vocales mayúsculas y minúsculas, puesto que son idénticas. Se usa un vector entero de cinco elementos para mantener los contadores de las vocales. Para reducir el número de comparaciones necesarias, cada carácter de la cadena de caracteres se convierte a mayúsculas antes de hacer dichas comparaciones. Los caracteres ASCII desde la «a» a la «z» se convierten fácilmente a mayúsculas haciendo un AND a nivel de bit de sus códigos con 0xdf.

```
Programa 6.22 #include <stdio.h>
#include <string.h>
#include <ctype.h>

main()
{
    char texto[] = "NUESTRO profesor EXPLICA cada CLASE con CLARIDAD.";
    char vstr[] = "AEIOU";
    static int vocales;
    static int vcount[5];
    int j;

    printf("El texto de entrada es => \"%s\"", texto);
    for(j = 0; j < strlen(texto); j++)
    {
```

```

        if (isalpha(texto[j]) != 0)
            texto[j] = texto[j] & 0xdf;
        if (strchr(vstr, texto[j]) != '\0')
            vocales++;
        switch(texto[j])
        {
            case 'A' : vcount[0]++; break;
            case 'E' : vcount[1]++; break;
            case 'I' : vcount[2]++; break;
            case 'O' : vcount[3]++; break;
            case 'U' : vcount[4]++; break;
        }
    }
    printf("\nEl texto de entrada contiene %d vocales.\n", vocales);
    printf("Hay %d As, %d Es, %d Is, %d Os, and %d Us",
        vcount[0], vcount[1], vcount[2], vcount[3], vcount[4]);
}

```

La ejecución del Programa 6.22 da lo siguiente:

El texto de entrada es => "NUESTRO profesor EXPLICA cada CLASE
con CLARIDAD."

El texto de entrada contiene 13 vocales.

Hay 4 As, 3 Es, 2 Is, 2 Os y 2 Us

Cuenta las vocales para verificar el correcto funcionamiento del programa.

Comprobación de palíndromos

Un **palíndromo** es una cadena de símbolos que se lee igual al derecho que al revés. Los palíndromos juegan un papel importante en el estudio de los lenguajes. Algunos palíndromos son:

mon radar otto 11011011

Si se ignoran los signos de puntuación, algunas expresiones completas pueden ser palíndromos:

un ojo un ojo un ojo un

es también palíndromo.

El Programa 6.23 permite al usuario introducir una cadena completa de símbolos. A continuación, el programa determina si la cadena es un palíndromo válido comprobando la igualdad de los símbolos comenzando por ambos extremos de la cadena.

Programa 6.23 #include <stdio.h>

main()

```

{
    char palstr[80];
    int lchar,rchar,stopped;

    printf("Introduzca un texto => ");
    gets(palstr);
    lchar = 0;
    rchar = strlen(palstr) - 1;
    stopped = 0;
    while((lchar <= rchar) &&!stopped)
    {
        if(palstr[lchar] != palstr[rchar])
            stopped = 1;
        lchar++;
        rchar--;
    }
    if(!stopped)
        printf("\"%s\" es palindromo.",palstr);
    else
        printf("\"%s\" no es palindromo.",palstr);
}

```

Para demostrar cómo funciona el Programa 6.23, se muestran a continuación varias ejecuciones de prueba:

```

Introduzca un texto => radar
"radar" es palindromo.

```

```

Introduzca un texto => 11011011
11011011 es palindromo.

```

```

Introduzca un texto => hola
hola no es palindromo.

```

Observe que el Programa 6.23 *no* ignora los signos de puntuación.

Un analizador léxico

En la Figura 6.7 se muestra la estructura de un *compilador*. Un compilador lee un fichero fuente (cualquier archivo en C por ejemplo), rompe cada línea del archivo fuente en *componentes léxicos*, pasa los componentes al *analizador sintáctico*, que comprueba que la gramática es correcta, y, finalmente, construye un archivo de código objeto que representa al archivo fuente original. Puesto que el estudio de los compiladores es un

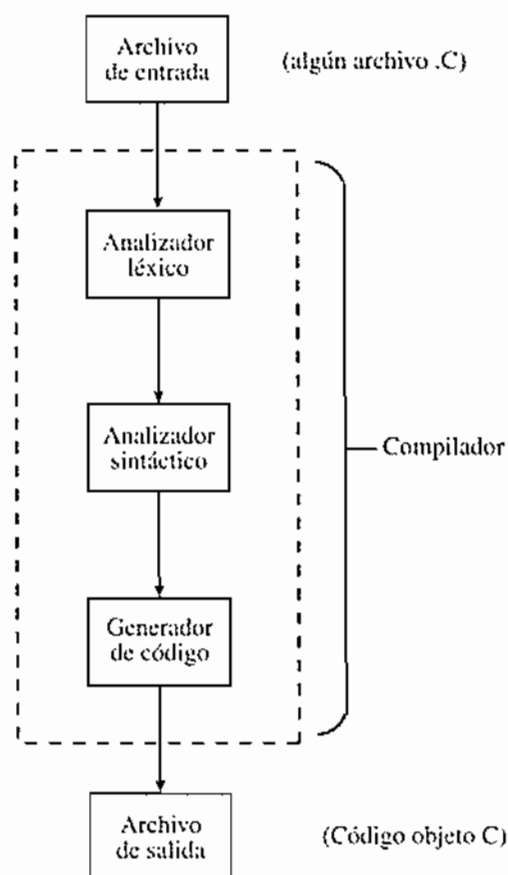


Figura 6.7. Estructura de un compilador.

tema avanzado, no se entrará en los detalles de las operaciones. En cambio, se examinarán las operaciones básicas de la primera parte del compilador, el analizador léxico.

Estudie la siguiente sentencia en C:

```
char cadena[] = "hola";
```

Esta sentencia se descompone en los siguientes componentes léxicos:

1. char
2. cadena
3. [
4.]
5. =
6. "hola"
7. ;

Como puede observar, la finalidad del analizador léxico es romper la sentencia de entrada para extraer los componentes más pequeños admitidos en el lenguaje (en este caso, el lenguaje C).

El Programa 6.24 muestra un analizador léxico muy limitado. No analizará correctamente cualquier sentencia de C que usted introduzca, solamente aquellas que contengan los símbolos definidos en los vectores *sencillos* y *dobles*.

```

Programa 6.24 #include <stdio.h>
               #include <string.h>
               #include <ctype.h>

main()
{
    char sencillos[] = " ; { } , = : ( ) [ ] * ? ' \" \\ ;";
    char dobles[] = " * = / = % = + = - = | | & & == != < = > = < < > > ++ -- ";
    char ch, entrada[80], ds[3];
    int pos = 0;

    printf("Introduzca una sentencia en C => ");
    gets(entrada);
    while (pos < strlen(entrada))
    {
        printf("Componente léxico: ");
        do
        {
            ch = entrada[pos];
            pos++;
        } while (ch == ' ');          /* Saltar blancos */

        /* ¿Es el nombre del símbolo? */
        if (isalpha(ch) != 0)
        {
            printf("%c", ch);          /* Imprimir la primera letra */

            /* Imprimir la letras/dígitos restantes */
            while (isalnum(entrada[pos]))
            {
                printf("%c", entrada[pos]);
                pos++;
            }
        }
        else

        /* ¿Es un número? */
        if (isdigit(ch))
        {
            printf("%c", ch);          /* Imprimir el primer dígito */

            /* Imprimir los dígitos restantes */
            while (isdigit(entrada[pos]))
            {
                printf("%c", entrada[pos]);
                pos++;
            }
        }
        else
    }
}

```

```

/* Comprobar que es un Componente léxico sencillo*/
if(strchr(sencillos,ch) != '\0')
{
    printf("%c",ch);          /* Imprimir Componente léxico */

    /* ¿Componente léxico de tipo cadena o carácter? */
    if((ch == '\"') || (ch == '\'))
    {
        do          /*Imprimir el texto del componente */
        {
            ch = entrada[pos];
            printf("%c",ch);
            pos++;
        } while((ch != ';' ) &&(ch != '\'));
    }
    else

    /* Comprobar el componente léxico == */
    if((ch == '=') && (entrada[pos] == '='))
    {
        printf("=");
        pos++;
    }
}
else

/* Comprobar que es un componente léxico doble */
{
    ds[0] = ch;          /* cargarlo */
    ds[1] = entrada[pos];
    ds[2] = '\0';
    if(strstr(dobles,ds) != '\0')
    {
        printf("%s",ds);          /* Imprimirlo */
        pos++;
    }
    else
        printf("%c",ch);
}
printf("\n");
}
}

```

El funcionamiento básico del Programa 6.24 es el siguiente: si un carácter de entrada es alfabético, se supone que es el principio del nombre de una variable o función. En otro caso, se busca en los vectores *sencillo* y *doble* alguna equivalencia. Si no se encuentra, el texto de entrada se imprime en la salida indicando que es un componente léxico indefinido.

Una ejecución de prueba podría ser la siguiente:

```
Introduzca una sentencia en C --> a = max(valores, n) * 25;
Componente léxico: a
Componente léxico: =
Componente léxico: max
Componente léxico: (
Componente léxico: valores
Componente léxico: ,
Componente léxico: n
Componente léxico: )
Componente léxico: *
Componente léxico: 25
Componente léxico: ;
```

Como puede ver, incluso las sentencias más sencillas de C están compuestas por múltiples componentes léxicos individuales. Se le anima a pensar en distintas formas de mejorar el analizador léxico. Por ejemplo, ¿cómo se harían componentes léxicos para los números hexadecimales o reales?

Codificación de texto

En aras de la seguridad, muchas organizaciones codifican actualmente sus datos de los computadores para evitar que los espías puedan obtener información. Existen muchas técnicas para codificar (o cifrar) un texto, algunas de las cuáles tienen cientos de años.

Una de las técnicas más simples es la codificación por *transposición*. En esta técnica, el texto de entrada se escribe como una matriz de caracteres bidimensional. Luego se transpone la matriz (filas y columnas invertidas) y se extraen los caracteres. Considere esta entrada tan sencilla:

```
programar en c es divertido
```

Sin incluir los caracteres blancos entre palabras, hay 23 caracteres. Si escribimos el texto de entrada en forma de matriz, tenemos:

```
p r o g r
a m a r e
n c e s d
i v e r t
i d o
```

Si ahora se leen los caracteres de la matriz por columnas (ignorando los blancos), se obtiene:

```
paniirmcvdoaceogrsrredt
```

que es el texto codificado por transposición. Los blancos se ignoran porque su posición podría dar pistas sobre donde comienzan o terminan las palabras.

El Programa 6.25 lleva a cabo una codificación por transposición creando una matriz cuadrada cuyas dimensiones se basan en la longitud del texto de entrada. Los ele-

mentos no usados de la matriz se rellenan con blancos para eliminar la posibilidad de que aparezcan caracteres extraños en la salida.

Programa 6.25

```

#include <stdio.h>
#include <string.h>
#include <math.h>

main()
{
    char entrada[80];
    char codificador[9][9];
    int r,c,p,n,i;

    printf("Introduzca el texto a codificar => ");
    gets(entrada);
    i = 0;
    for(n = 0; n < strlen(entrada); n++)
        if(entrada[n] == ' ')
            i++;
    n = 1+sqrt(strlen(entrada) - i);
    for(r = 0; r < 9; r++)
        for(c = 0; c < 9; c++)
            codificador[r][c] = ' ';

    p = 0;
    r = 0;
    c = 0;
    while(entrada[p] != '\0')
    {
        if(entrada[p] != ' ')
        {
            codificador[r][c] = entrada[p];
            r++;
            if(r == n)
            {
                r = 0;
                c++;
            }
        }
        p++;
    }

    printf("Codificación por transposición: ");
    for(r = 0; r < 9; r++)
        for(c = 0; c < 9; c++)
            if(codificador[r][c] != ' ')
                printf("%c",codificador[r][c]);
}

```

Si se ejecuta el Programa 6.25 con el texto sencillo que vimos antes se obtiene el siguiente resultado:

```
Introduzca el texto a codificar => programar en c es divertido
Codificación por transposición: animecvdaeeorsredt
```

Si se deshace la transposición se obtiene el texto original. Se deja como tarea suya adivinar cómo se hace.

Conclusión

En esta sección se han presentado varias aplicaciones sencillas de sobre cadenas de caracteres. Compruebe sus conocimientos de la materia haciendo los ejercicios de repaso.

Repaso de la Sección 6.8

1. ¿Por qué es necesario restar 0x30 de cada dígito en el comprobador del ISBN?
2. ¿Cómo se convierten las cadenas de caracteres a mayúsculas?
3. ¿Qué técnica se usa para buscar los palíndromos en el Programa 6.23?
4. ¿Qué hace un analizador léxico? Dé un ejemplo.
5. Codifique la cadena de caracteres `juan esta aqui` usando la codificación por transposición.

Ejercicios interactivos

DIRECTRICES

Ejecute cada programa en su computador y observe los resultados de su ejecución. En muchos casos se le pide que prediga los que hará el programa. Compare sus predicciones con lo que hace el programa de verdad.

Ejercicios

1. El Programa 6.26 contiene una cadena de caracteres sin valor inicial. ¿Qué se verá cuando se ejecute el programa?

Programa 6.26

```
#include <stdio.h>
main()
{
    char string[10];
    int j;

    printf("%s",string);

    for (j=0; j<10; j++)
        string[j] = j + 65;
    printf("%s",string);
}
```

2. ¿Qué hace el Programa 6.27 cuando se ejecuta?

Programa 6.27

```
#include <stdio.h>
void xyz(char abc[]);

main()
{
    char *string;

    xyz(string);
    printf("%s",string);
}
void xyz(char abc[])
{
    char def[] = "¿Esto funciona?";
    abc = def;
}
```

3. ¿Qué hace el Programa 6.28 cuando se ejecuta?

Programa 6.28

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

main()
{
    char abc[] = "¿Esto se imprime?";
    int i;

    abc[7] = abc[strlen(abc)];
    printf("%s",abc);
}
```

4. ¿Qué hace el Programa 6.29 cuando se ejecuta?

Programa 6.29

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

main()
{
    char sa[80] = "La cara oculta de ";
    char sb[20] = "la medianoche.";
    char sc[] = "la luna.";

    strncat(sa,sb,3);
    strcat(sa,sc);
    printf("%s",sa);
}
```

5. ¿Es el entero que sale del Programa 6.30 positivo o negativo?

Programa 6.30

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

main()
{
    char sa[]    "cpu";
    char sb[]    "CPU";

    printf("%d", strcmp(sa, sb));
}
```

6. ¿Qué hace el Programa 6.31 cuando se ejecuta?

Programa 6.31

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

main()
{
    char string[] = "Esto es solo una prueba...";

    printf("%s", strchr(string, 't'));
    printf("%s", strchr(string, 't'));
}
```

Autoevaluación

DIRECTRICES

Responda a las siguientes preguntas mirando los programas en la Sección 6.8 (Programas de aplicación adicionales).

Preguntas

1. ¿Cuándo se usa la X en un número de ISBN? ¿Cómo genera el Programa 6.21 la X?
2. ¿Por qué `vowels` and `vcount` se han declarado como `static int` (y no simplemente `int`) en el Programa 6.22?
3. ¿Cómo sabe el Programa 6.23 cuando ha comprobado completamente un palíndromo?
4. ¿Por qué es más difícil comprobar los componentes léxicos en el vector `dobles` que en el `sencillos` en el Programa 6.24?
5. ¿Cómo se extrae un componente léxico de una cadena de caracteres en el Programa 6.24?
6. ¿Cómo se rellena la matriz de codificación en el Programa 6.25?

Problemas de fin de capítulo

Conceptos generales

Sección 6.1

1. ¿Cuál es el índice del primer carácter en una cadena de caracteres en C?
2. Indique cuántos elementos son necesarios en una cadena de caracteres de C compuesta por ocho caracteres. Explíquelo.
3. Explique el formato de una cadena de caracteres en C.
4. ¿Cómo se declaran en C las variables de cadenas de caracteres?

Sección 6.2

5. ¿Cuáles son las tres formas de iniciar la cadena de caracteres "Datos"?
6. ¿Por qué dimensionar una cadena de caracteres con un tamaño mayor al necesario? Por ejemplo, ¿por qué usar `char cadena[80] = "Hola";` en lugar de `char cadena[] = "Hola";`?
7. ¿Cuáles son las diferencias entre `scanf()` y `gets()` en relación a la entrada de cadenas de caracteres?

Sección 6.3

8. ¿Qué se pasa realmente entre funciones cuando se trabaja con cadenas de caracteres?
9. ¿Dónde debe reservarse el espacio en memoria cuando se pasan cadenas de caracteres entre funciones?

Sección 6.4

10. ¿Por qué hacer disponible la función `isprintf()`? ¿Qué valor tiene?
11. ¿Hace `getchar()` eco de los caracteres a la pantalla?
12. ¿Por qué escribir funciones de conversión propias si existen `atoi()` y `atof()`?

Sección 6.5

13. Si se pierde el carácter nulo del final de una cadena de caracteres, ¿qué hace la función `strlen()`?
14. ¿Qué ocurre a las cadenas de caracteres usadas en una operación `strcat()`?
15. ¿Por qué tiene sentido que "Javi" tenga un valor menor que "Javier"? ¿Cómo podría averiguarlo?
16. ¿Cómo podría usarse la función `strchr()` para buscar los símbolos +, -, * y / en una cadena de caracteres?
17. ¿Qué otras operaciones con cadenas de caracteres serían útiles? Explique cómo podrían ser implementadas.

Sección 6.6

18. ¿Qué es una matriz rectangular de caracteres? ¿Cómo se almacena en memoria?
19. ¿Para qué se usan las llaves {} en una declaración de cadenas de caracteres?
20. ¿Cuáles son las distintas formas de definir una cadena de caracteres de dos dimensiones?
21. ¿Qué es una matriz irregular?

Sección 6.7

22. ¿Cómo se define una cadena de caracteres muy larga?
23. ¿Cuál es la técnica usada para expandir una línea de texto?

Diseño de programas

Para cada programa que se asigne, documente su proceso de diseño y refléjelo en un programa. Este proceso debería incluir el esquema de diseño, proceso de la entrada y salida necesaria, así como el algoritmo del programa. Asegúrese de incluir toda la documentación en el programa final. Esta debería estar compuesta, al menos, del bloque del programador, prototipos de funciones y descripción de cada función así como de los argumentos formales que se utilicen.

25. Escriba un programa C que pida al usuario un número en notación científica (vea la Sección 6.4) y compruebe su validez. Si es válido, conviértalo al valor entero apropiado.
26. Escriba un programa C que genere e imprima una matriz de 10 por 10 letras del alfabeto aleatorias.

	Columna			
	1	2	3	4
Fila	1	Rojo	Verde	Azul
	2	Violeta	Ámbar	Marrón
	3	Naranja	Rosa	Magenta
	4	Plata	Oro	Pizarra

Figura 6.8. Sistema de rejilla para el Problema 33.

27. Modifique el Problema 26 de forma que busque fila a fila y columna a columna cualquiera de las siguientes palabras:
- | | | | | | |
|-------|-----------|---------|--------|-------|--------|
| uno | dos | tres | cuatro | cinco | seis |
| siete | ocho | nueve | diez | ayuda | ver |
| vio | chico | chica | rapido | lento | arriba |
| abajo | izquierda | derecha | arriba | ir | parar |
28. Escriba su propia función `expandir_linea()` para el formateador de texto (Sección 6.7).
29. Mejore el analizador léxico desarrollado en el Programa 6.24 de forma que identifique más operadores estándares de C.
30. Cree un programa C que dé al usuario el color de un área de la rejilla que se muestra en la Figura 6.8. El usuario del programa debe proporcionar los números de fila y columna.

Vectores y matrices¹ numéricas

Objetivos

Este capítulo le da la oportunidad de aprender lo siguiente:

1. Qué es un vector numérico.
2. Cómo crear y asignar valores iniciales a un vector numérico.
3. Aplicaciones de los vectores numéricos.
4. Cómo pasar en C vectores numéricos entre funciones.
5. Los conceptos básicos de ordenación con vectores numéricos.
6. Cómo mezclar dos vectores ordenados.
7. Álgebra de matrices.

Palabras clave

Vector (*array*) numérico

Dimensión

Iniciación de un vector

Paso de vectores

Índice de un vector

Comparación

Ordenación por el método de la burbuja

Ordenación por el método de la cubeta

Ordenación por difusión

Ordenación rápida (*quick sort*)

Pivote

Vector (*array*) multidimensional

Matrices

Fila

Columna

Almacenamiento por filas

Especificador de ancho de campo

¹ El termino inglés **array** es uno de los más controvertidos en su traducción. En España se suele aceptar el término inglés, mientras que en Latinoamérica es común la palabra **arreglo**. En este libro se ha traducido bien como vector o matriz para referirse a una o más dimensiones, por respetar el espíritu del autor (*N. del RT.*).

Contenido

7.1. Vectores numéricos	7.3. Ordenación de vectores numéricos
7.2. Introducción a las aplicaciones con vectores numéricos	7.4. Matrices numéricas
	7.5. Programas de aplicación

Introducción

Usted encontrará algunas similitudes entre la operación con variables de cadenas de caracteres (tratadas en el Capítulo 6) y la operación con variables de vectores numéricos. Ambos utilizan múltiples elementos a los que se accede por medio de un índice, y ambos deben ser dimensionados o iniciados de antemano para trabajar apropiadamente. Este capítulo se centra en el uso de vectores numéricos en aplicaciones nuevas y examina los métodos que se utilizan para acceder, dividir, ordenar, mezclar y buscar en ellos.

7.1. VECTORES NUMÉRICOS

Presentación

Esta sección versa sobre **vectores numéricos**. Descubrirá cómo indicar la dimensión de un vector numérico y aprenderá más acerca de la relación entre los elementos de un vector y los punteros.

Idea básica

Para indicar la **dimensión** de un vector numérico, simplemente debe situarse un número entre sus corchetes. Como ejemplo, `int vector[3];` representa la declaración de un vector compuesto por tres elementos. Estos elementos son `vector[0]`, `vector[1]` y `vector[2]`.

El Programa 7.1 muestra la relación entre `vector` y `&vector[0]`.

Programa 7.1

```
#include <stdio.h>

main()
{
    int vector[3];

    printf("vector = %X\n",vector);
    printf("&vector[0] = %X\n",&vector[0]);
}
```

La ejecución del Programa 7.1 produce el mismo valor para cada una de las dos variables:

```
vector = 7325
&vector[0] = 7325
```

Esto significa que el nombre de la variable que representa el vector y la dirección del primer elemento del vector tienen el mismo valor, esto es, la primera dirección de memoria del vector. En la Figura 7.1 se presenta este concepto.

El vector de tipo int

Observe que el Programa 7.1 utiliza un int con un vector. Recuerde que en un computador personal un entero de tipo int utiliza dos posiciones de memoria de 8 bits (un char cada uno). Para ilustrar esto, en el Programa 7.2 se considera el mismo vector. La diferencia en este caso es que se imprime la dirección de cada uno de los elementos del vector.

Programa 7.2

```
#include <stdio.h>

main()
{
    int vector[3];

    printf("direccion de vector[0] => %d\n",&vector[0]);
    printf("direccion de vector[1] => %d\n",&vector[1]);
    printf("direccion de vector[2] => %d\n",&vector[2]);
}
```

La ejecución del Programa 7.2 produce la siguiente salida:

```
direccion de vector[0] = 7325
direccion de vector[1] = 7327
direccion de vector[2] = 7329
```

Nótese que cada dirección es dos unidades mayor que la anterior. Esto se debe a que los elementos del vector son de tipo int, y ocupan cada uno de ellos dos bytes. En la Figura 7.2 se presenta este hecho.

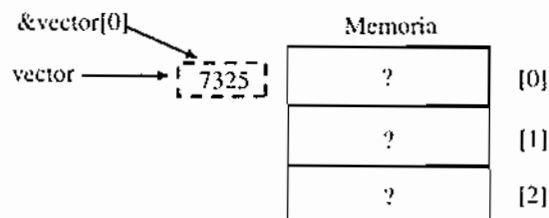


Figura 7.1. Primera dirección de un vector.

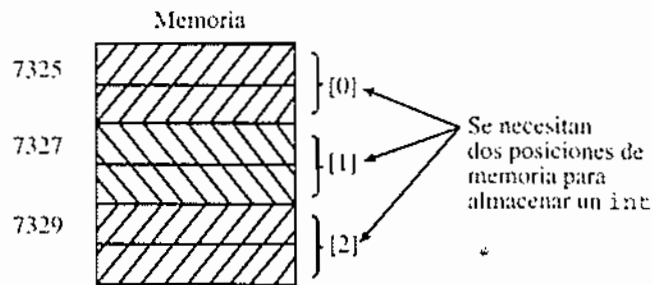


Figura 7.2. Dirección con vectores de tipo int.

Esto significa que C automáticamente asigna el espacio de memoria necesario dependiendo del tipo de datos que se esté utilizando en el vector, con todos los elementos del vector dispuestos de forma contigua en memoria.

Dentro de un vector

En el Programa 7.2 se vio la dirección de cada elemento del vector ¿Qué hay dentro de cada uno de estos elementos (lo que se almacena en esas direcciones)? El Programa 7.3 da una idea de esto.

Programa 7.3 `#include <stdio.h>`

```

main()
{
    int vector[3];

    printf("Contenido de vector[0] => %d\n", vector[0]);
    printf("Contenido de vector[1] => %d\n", vector[1]);
    printf("Contenido de vector[2] => %d\n", vector[2]);
}

```

La ejecución del Programa 7.3 produce la siguiente salida:

```

Contenido de vector[0] => 0
Contenido de vector[1] => -5672
Contenido de vector[2] => 58

```

¿De dónde vienen estos valores? Ellos son justo los que se encuentran en esas posiciones de memoria en el momento de la ejecución del programa. Lo que ocurre es que el programa ha reservado suficiente espacio en memoria para los tres elementos del vector, pero no ha almacenado nada en estas posiciones de memoria. Ahora considere el Programa 7.4, una ligera variante del Programa 7.3. En esta ocasión, sin embargo, el vector se declara como una variable global.

Programa 7.4

```
#include <stdio.h>

int vector[3];

main()
{
    printf("Contenido de vector[0] => %d\n", vector[0]);
    printf("Contenido de vector[1] => %d\n", vector[1]);
    printf("Contenido de vector[2] => %d\n", vector[2]);
}
```

Observe lo que ocurre cuando se ejecuta el Programa 7.4:

```
Contenido de vector[0] => 0
Contenido de vector[1] => 0
Contenido de vector[2] => 0
```

En esta ocasión, todos los elementos del vector tienen asignado el valor cero. La asignación de valores conocidos a los elementos de un vector se denomina **iniciación de un vector**.

La única diferencia entre los dos últimos programas es que en el Programa 7.3 el vector se declara como una variable automática (local), mientras que en el Programa 7.4 se declara como una variable global (externa). Esto muestra una importante característica de C: los vectores declarados de forma global son iniciados a cero por defecto, mientras que los vectores locales no.

Cuando se necesita iniciar un vector numérico local, una técnica alternativa que se puede utilizar es la siguiente:

```
main()
{
    static int vector[3];
    .
    .
    .
}
```

Nótese el uso de la palabra `static` en la declaración del vector. Un vector de enteros estático se inicia automáticamente a cero al comienzo de la ejecución del programa. Sin embargo, debido a que `static` presenta otras características, tales como el control sobre el tiempo de vida de las variables, es mejor para el programador utilizar la técnica de iniciación que se describe en la siguiente sección.

Asignación de valores por el usuario

El método para asignar valores a los elementos de un vector se muestra en el Programa 7.5.

Programa 7.5 `#include <stdio.h>`

```
main()
{
    int vector[3];
    int *ptr;

    vector[0] = 10;
    vector[1] = 20;
    vector[2] = 30;

    printf("Contenido de vector[0] => %d\n", vector[0]);
    printf("Contenido de vector[1] => %d\n", vector[1]);
    printf("Contenido de vector[2] => %d\n", vector[2]);

    ptr = vector;
    printf("Contenido de vector[0] => %d\n", *ptr);
    printf("Contenido de vector[1] => %d\n", *(ptr + 1));
    printf("Contenido de vector[2] => %d\n", *(ptr + 2));
}
```

La ejecución del Programa 7.5 produce la siguiente salida:

```
Contenido de vector[0] => 10
Contenido de vector[1] => 20
Contenido de vector[2] => 30
Contenido de vector[0] => 10
Contenido de vector[1] => 20
Contenido de vector[2] => 30
```

Como puede verse en el Programa 7.5, cada elemento del vector ahora contiene un valor asignado por usted. También se puede ver el uso de punteros para obtener el contenido de cada uno de los elementos del vector. Esto demuestra las siguientes igualdades:

```
vector[0] = *ptr
vector[1] = *(ptr + 1)
vector[2] = *(ptr + 2)
```

Observe que `*ptr` se declara de tipo `int`. Esto indica a C que cada incremento de `ptr` es de dos bytes y no uno (debido a que el tipo `int` utiliza dos bytes en memoria en un computador personal).

Paso de vectores numéricos

Se pueden pasar vectores numéricos de una función a otra. En el Programa 7.6 se muestra el **paso de vectores**.

Programa 7.6

```
#include <stdio.h>

void funcion(int v[]);

main()
{
    int vector[3];

    vector[0] = 10;
    vector[1] = 20;
    vector[2] = 30;

    funcion(vector);
}

void funcion(int v[])
{
    printf("Contenido de vector[0] => %d\n",v[0]);
    printf("Contenido de vector[1] => %d\n",v[1]);
    printf("Contenido de vector[2] => %d\n",v[2]);
}
```

La ejecución del Programa 7.6 produce de nuevo:

```
Contenido de vector[0] => 10
Contenido de vector[1] => 20
Contenido de vector[2] => 30
```

Como se puede observar no es necesario especificar el tamaño del vector dentro de la declaración formal de argumentos.

Análisis del programa

El Programa 7.7 declara un prototipo de función de tipo `void` que contiene un vector como argumento:

```
void funcion(int v[]);
```

Observe que el vector se denomina `v[]`. El operador de indirección `*` no se ha utilizado (aunque podría haberse utilizado) debido a que `v[]` es un puntero que apunta al primer elemento del vector.

La función `main()` también define un vector compuesto por tres elementos a los que asigna unos determinados valores.

```
int vector[3];

vector[0] = 10;
vector[1] = 10;
vector[2] = 10;
```

Después de estas asignaciones, se llama a la función `funcion()`:

```
funcion(vector);
```

Nótese que el argumento real no utiliza el operador `&`. Podría haberse utilizado `&vector[0]`, sin embargo, `vector` es la dirección de comienzo del vector y esto es lo que se pasa a la función.

A continuación, `funcion()`, recibe el valor de la dirección del primer elemento del vector y procede a imprimir los valores de sus tres primeros elementos.

```
printf("Contenido de vector[0] => %d\n",v[0]);
printf("Contenido de vector[1] => %d\n",v[1]);
printf("Contenido de vector[2] => %d\n",v[2]);
```

Todo esto se representa en la Figura 7.3.

El siguiente programa (Programa 7.7) ilustra el paso de un vector a una función.

Programa 7.7

```
#include <stdio.h>

void funcion(int v[]);

main()
{
    int vector[3];

    funcion(vector);

    printf("Contenido de vector[0] => %d\n",vector[0]);
    printf("Contenido de vector[1] => %d\n",vector[1]);
    printf("Contenido de vector[2] => %d\n",vector[2]);
}

void funcion(int v[])
{
    v[0] = 20;
    v[1] = 40;
    v[2] = 60;
}
```

Paso de vectores numéricos de vuelta

El paso de un vector numérico de vuelta de una función implica que se pase el vector entero. Lo que realmente ocurre, es que cuando se pasa un vector a una función, no se pasan los valores de los elementos del vector, sino simplemente la dirección de comienzo del mismo. Por tanto, la función llamada puede utilizar esta dirección para escribir nuevos valores en las posiciones de memoria reservadas para el vector numérico. El Programa 7.7 demuestra este principio. En este programa, `main()` pasa la dirección de comienzo del vector a `funcion()`. Durante la ejecución de `funcion()`, ésta asigna valores a los elementos del vector debido a que `funcion()` conoce la

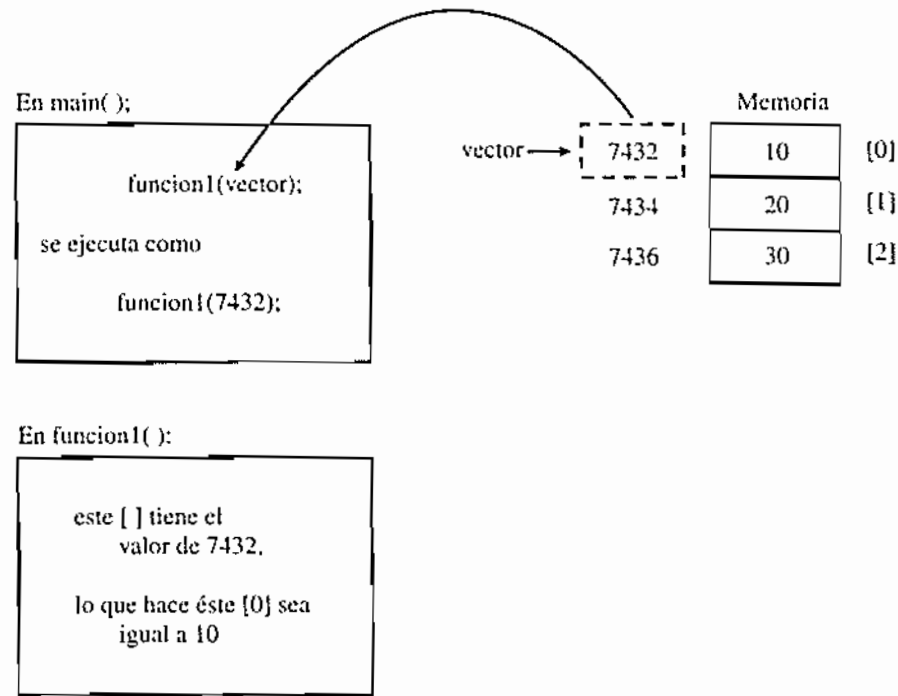


Figura 7.3. Pasando la dirección de un vector como parámetro de una función.

dirección de los elementos del vector y puede acceder a la dirección real de memoria de cada uno de los elementos.

La salida del Programa 7.7 es la siguiente:

```
Contenido de vector[0] => 20
Contenido de vector[0] => 40
Contenido de vector[0] => 60
```

De nuevo, se pasa la dirección de comienzo del vector a la función mediante `funcion(vector);`. Debido a que `funcion()` conoce la dirección de comienzo del vector, puede asignar un valor a cada uno de los elementos del vector. La Figura 7.4 ilustra este proceso.

Como se puede ver de los programas anteriores, las variables de tipo vector se pueden pasar fácilmente entre las funciones de un programa C.

Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 7.1

1. ¿Cómo se puede indicar a C el número de elementos de un vector?
2. ¿Cuál es siempre el índice del primer elemento en un vector en C?
3. Si la variable `int valor[5];` se declara en una función, ¿cuál es la relación que existe entre `valor` y `&valor[0]`?
4. ¿Qué diferencia existe entre la declaración de un vector local y un vector global?
5. ¿Por qué `char abc[3];` e `int def[3];` reservan cantidades diferentes de memoria?

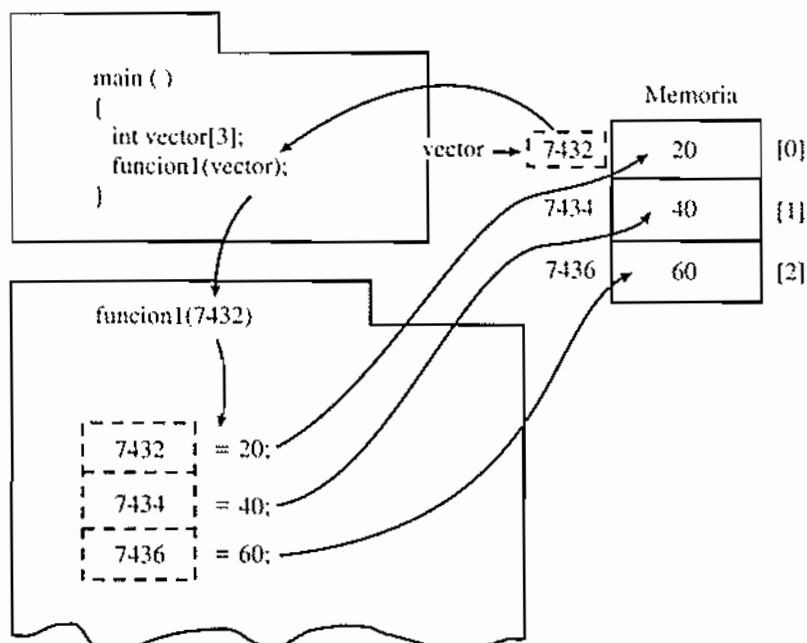


Figura 7.4. Proceso de iniciación de una función llamada.

7.2. INTRODUCCIÓN A LAS APLICACIONES CON VECTORES NUMÉRICOS

Presentación

Esta sección presenta los conceptos fundamentales de las aplicaciones de los vectores en programas técnicos. Se verá como utilizar vectores para cambiar el orden de una lista de números o encontrar el valor más pequeño de una secuencia de valores introducidos por el usuario.

Trabajar con el índice de un vector

La idea básica que hay detrás de las aplicaciones con vectores es la capacidad para trabajar con el índice del vector. Es importante no pensar en el valor del **índice de un vector** como un número fijo; estos valores pueden manipularse de la manera que se elija. Un buen manejo de este concepto puede dar lugar al desarrollo de programas técnicos muy potentes. El Programa 7.8 ilustra este concepto.

Programa 7.8

```
#include <stdio.h>

main()
{
    int vector_de_numeros[7];
    int indice;
```



```

/* Almacena los valores en el vector. */
vector_de_numeros[0] = 0;
vector_de_numeros[1] = 2;
vector_de_numeros[2] = 4;
vector_de_numeros[3] = 8;
vector_de_numeros[4] = 16;
vector_de_numeros[5] = 32;
vector_de_numeros[6] = 64;

for(indice = 1; indice <= 5; indice++)
{
    printf("vector_de_numeros[%d] = %d", indice,
        vector_de_numeros[indice]);
    printf(" vector_de_numeros[%d + 1] = %d", indice,
        vector_de_numeros[indice+1]);
    printf(" vector_de_numeros[%d - 1] = %d\n", indice,
        vector_de_numeros[indice-1]);
}
}

```

La ejecución del Programa 7.8 produce la siguiente salida:

```

vector_de_numeros[1] = 2 vector_de_numeros[1 + 1] = 4
vector_de_numeros[1 - 1] = 0
vector_de_numeros[2] = 4 vector_de_numeros[2 + 1] = 8
vector_de_numeros[2 - 1] = 2
vector_de_numeros[3] = 8 vector_de_numeros[3 + 1] = 16
vector_de_numeros[3 - 1] = 4
vector_de_numeros[4] = 16 vector_de_numeros[4 + 1] = 32
vector_de_numeros[4 - 1] = 8
vector_de_numeros[5] = 32 vector_de_numeros[5 + 1] = 64
vector_de_numeros[5 - 1] = 16

```

Como puede verse en esta salida, el cambio en el valor del índice de un vector provoca un cambio correspondiente en el valor de salida del programa. En el Programa 7.8, se asignan valores a cada uno de los elementos del vector. A continuación, se imprimen los elementos del vector. Sin embargo, para demostrar el efecto que produce en la salida el cambio de los valores de los índices, a cada índice se le incrementa en 1 y se le decrementa en 1. Así, `vector_de_numeros[3]` contiene el mismo valor que `vector_de_numeros[2 + 1]` y `vector_de_numeros[4 - 1]`.

Cambio en la secuencia

El Programa 7.9 ilustra la manipulación del índice de un vector. Este programa permite al usuario introducir nueve números. El programa los muestra a continuación, pero en orden inverso al que fueron introducidos. El programa además busca e imprime el valor mínimo dentro del vector.

Programa 7.9 #include <stdio.h>

```
#define numero_maximo 9    /* Número máximo de elementos en el vector. */

void invertir(int vector[]);
int minimo(int vector[]);

main()
{
    int numeros[numero_maximo];
    int indice;
    int min;

    printf("Deme nueve numeros y los imprimire en orden inverso.\n");
    printf("Ademas imprimire el valor minimo.\n");
    for(indice = 0; indice < numero_maximo; indice++)
    {
        printf("Numero[%d] = ", indice);
        scanf("%d", &numeros[indice]);
    }

    printf("Gracias...\n");

    invertir(numeros);
    min = minimo(numeros);
    printf("\nEl valor minimo es %d\n", min);
}

void invertir(int vector[])
{
    int indice;

    printf("\n\nEstos son los numeros que introdujo en orden\n");
    printf("inverso de entrada:\n");

    for(indice = numero_maximo - 1; indice >= 0; indice--)
        printf("numero[%d] = %d\n", indice, vector[indice]);
}

int minimo(int vector[])
{
    register int indice;
    int min;

    min = vector[0];
    for(indice = 0; indice < numero_maximo; indice++)
        if(vector[indice] < min)
            min = vector[indice];
    return(min);
}
```

Considerando que el usuario introduce los números del 1 al 9, la ejecución del Programa 7.10 produciría la siguiente salida:

```
Deme nueve numeros y los imprimire en orden inverso.
Ademas imprimire el valor minimo.
numero[0] = 1
numero[1] = 2
numero[2] = 3
numero[3] = 4
numero[4] = 5
numero[5] = 6
numero[6] = 7
numero[7] = 8
numero[8] = 9
Gracias...
```

```
Estos son los numeros que introdujo en orden
inverso de entrada
numero[8] = 9
numero[7] = 8
numero[6] = 7
numero[5] = 6
numero[4] = 5
numero[3] = 4
numero[2] = 3
numero[1] = 2
numero[0] = 1
```

El valor minimo es 1

Análisis del programa

El programa define en primer lugar el tamaño máximo del vector con una directiva `#define`:

```
#define numero_maximo 9 /* Número máximo de elementos en el vector. */
```

A continuación se declaran los prototipos de las funciones que emplean un vector en su argumento formal:

```
void invertir(int vector[]);
void minimo(int vector[]);
```

Un bucle `for` de C se encarga de leer los valores introducidos por el usuario. Observe que en este bucle, el índice del vector se incrementa desde 0 (primer elemento del vector) a 1 menor que `numero_maximo` (último elemento del vector). Éste es un método eficaz para almacenar en un vector los valores introducidos por el usuario:

```
printf("Deme nueve numeros y los imprimire en orden inverso.\n");
for(indice = 0; indice < numero_maximo; indice++)
{
```

```

        printf("Numero[%d] = ", indice);
        scanf("%d", &numeros[indice]);
    }

```

El programa llama a continuación a una función que imprime los elementos del vector en orden inverso. Observe que el argumento real que se emplea es la dirección de comienzo del vector:

```
invertir(vector);
```

Esta función utiliza otro bucle `for` de C, pero en esta ocasión el bucle comienza con el índice del vector en `numero_maximo - 1` (el máximo índice del vector) y luego finaliza en 0, mediante el operador de C `--` aplicado a la variable índice.

```

for(indice = numero_maximo - 1; indice >= 0; indice--)
    printf("numero[%d] = %d\n", indice, vector[indice]);

```

En cada caso, se imprime el valor de la variable. A continuación se utiliza la función `minimo` para calcular el valor mínimo del vector. Esta función define una variable denominada `min`, que almacenará el valor mínimo del vector. Inicialmente se asigna a `min` el primer elemento del vector:

```
min = vector[0];
```

Esto se hace para que la variable `min` tome un valor del vector con el cual compare el resto de valores. Esta comparación se realiza mediante el siguiente bucle `for` de C:

```

for(indice = 0; indice < numero_maximo; indice++)
    if(vector[indice] < min)
        min = vector[indice];

```

Se va comparando la variable `min` con cada uno de los elementos del vector. Cuando este elemento es menor que `min`, este valor se asigna a la variable `min`. De esta forma se obtiene el valor más pequeño del vector. Una vez obtenido este valor, se devuelve a la función `main()` para que lo imprima.

```
return(min);
```

Conclusión

En esta sección se ha presentado el concepto del acceso a los datos de un vector. Se ha visto la forma de modificar el índice de un vector para llevar a cabo esta tarea. En esta sección también se ha presentado un método que permite encontrar el valor mínimo de una lista de valores introducidos por el usuario. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 7.2

1. Indique la idea básica que hay detrás de las aplicaciones con vectores.

2. Explique el método de programación empleado para obtener y almacenar en un vector un conjunto de valores introducidos por el usuario.
3. Indique el método empleado que permite que una serie de números introducidos por el usuario sean impresos en orden inverso al que fueron introducidos.
4. ¿Qué método se emplea para buscar el valor mínimo de una lista de valores?
5. ¿Cómo se puede obtener el valor máximo de una lista de números?

7.3. ORDENACIÓN DE VECTORES NUMÉRICOS

En esta sección se examinarán diferentes técnicas de ordenación de vectores. Estas técnicas son la siguientes: ordenación por el *método de la burbuja*, ordenación por el *método de la cubeta*, ordenación por *fusión* y ordenación *rápida (quicksort)*.

Una medida que permite medir la eficacia de los algoritmos de ordenación es el número de **comparaciones** que se realizan durante el proceso. Debido a que los métodos empleados en cada técnica son fundamentalmente diferentes, algunas técnicas de ordenación son más eficaces que otras (siendo la ordenación por el método de la burbuja la menos eficiente). Se examinarán cada una de estas técnicas y se mostrará la manera de ordenar un vector de enteros. Para evaluar la eficiencia de las distintas técnicas se calculará el número de comparaciones que se requieren para ordenar un vector.

Ordenación por el método de la burbuja

La ordenación por el **método de la burbuja** toma su nombre del hecho de que los números en el vector a ordenar recorren la lista a borbotones hacia el comienzo de la misma mientras progresa la ordenación.

Como ejemplo, considere la siguiente lista de números:

6
3
6
8

Supóngase que se tiene que ordenar la lista en orden ascendente (primero el número más pequeño y el último el más grande). Los pasos a seguir para ordenar esta lista utilizando el método de la burbuja son los siguientes:

Reglas para la ordenación por el método de la burbuja (para ordenar en orden ascendente).

1. Sólo se comprueban dos números cada vez, comenzando con los dos primeros números.
2. Si el primer número es más pequeño, dejarlo con está. En caso contrario, se intercambian los dos números.
3. Avanzar un número y comparar dicho número con el que le sigue a continuación. Estos dos números constituirán el nuevo par a comparar.

4. Este proceso se continúa hasta que no se requiera ningún intercambio de números en la pasada completa sobre la lista.

Para realizar la ordenación en orden descendente, simplemente es necesario cambiar el paso 2 por el siguiente:

2. Si el primer número es mayor, dejarlo como está. En caso contrario, intercambiar los dos números.

Para ordenar la lista anterior utilizando este método, se comienza por la primera regla. Sólo se comprueban dos números cada vez, comenzando con los dos primeros números de la lista:

6
3

El primer número es el mayor, así que utilizando la regla 2 se intercambian:

3
6

La lista a continuación es la siguiente:

3
6
6
8

Se avanza al siguiente número de la lista y se compara con el que le sigue. El nuevo par de números a comparar es:

6
6

Estos números son iguales. Puesto que son iguales, no se realiza ninguna acción. El siguiente par de números es:

6
8

Debido a que el primer número es menor que el segundo, se deja el par tal y como está. Ya se ha completado un recorrido a la lista. Debido a que se realizó un intercambio durante esta pasada, es necesario volver a comenzar desde el principio:

Se comparan los dos primeros números de la lista:

3
6

no se requiere ningún intercambio. Ahora se comparan los dos siguientes:

6
6.

De nuevo no se requiere ningún intercambio. Los dos siguientes números son:

6
8

Tampoco se ha requerido un intercambio. Por tanto, como durante este recorrido no se ha realizado ningún intercambio de números, la ordenación ha concluido dando el siguiente resultado:

3
6
6
8

Programa de ejemplo

El Programa 7.10 implementa la ordenación de un vector por el método de la burbuja.

Programa 7.10 #include <stdio.h>

```
#define numero_maximo 9 /* Número máximo de elementos en el vector. */
int ordenar_burbuja(int vector[]);
void imprimir_vector(int vector[]);
main()
{
    int numeros[numero_maximo];
    int indice, comparaciones;

    printf("Deme nueve numeros y los ordenare: \n");
    for (indice = 0; indice < numero_maximo; indice++)
    {
        printf("Numero[%d] = ", indice);
        scanf("%d", &numeros[indice]);
    }

    comparaciones = ordenar_burbuja(numeros);
    imprimir_vector(numeros);
    printf("\nEl numero de comparaciones es %d", comparaciones);
}

int ordenar_burbuja(int vector[])
{
    int indice;
    int seguir;
    int temp;
    int comp = 0;

    do
    {
```

```

        seguir = 0;
        for (indice = 0; indice < numero_maximo; indice++)
        {
            comp++;
            if((vector[indice] > vector[indice + 1])
                &&(indice != numero_maximo - 1))
            {
                temp = vector[indice];
                vector[indice] = vector[indice + 1];
                vector[indice + 1] = temp;
                seguir = 1;
            }
        }
    } while (seguir);
    return(comp);
}

void imprimir_vector(int vector[])
{
    int indice;
    printf("\n\nLos valores ordenados son: \n");
    for (indice = 0; indice < numero_maximo; indice++)
        printf("Numero[%d] = %d\n", indice, vector[indice]);
}

```

Asumiendo que el usuario introduce los números que se muestran a continuación, la ejecución del Programa 7.10 produce la siguiente salida:

Deme nueve numeros y los ordenare:

```

Numero[0] = 6
Numero[1] = 7
Numero[2] = 5
Numero[3] = 8
Numero[4] = 4
Numero[5] = 9
Numero[6] = 3
Numero[7] = 0
Numero[8] = 2

```

Los valores ordenados son:

```

Numero[0] = 0
Numero[1] = 2
Numero[2] = 3
Numero[3] = 4
Numero[4] = 5
Numero[5] = 6
Numero[6] = 7
Numero[7] = 8
Numero[8] = 9

```

El numero de comparaciones es 72

Análisis del programa

Después de la sentencia `#define numero_maximo 9`, que se emplea para indicar el tamaño máximo del vector, el programa define dos prototipos de funciones:

```
int ordenar_burbuja(int vector[]);
void imprimir_vector(int vector[]);
```

Los parámetros formales de ambas funciones se declaran como vectores de tipo `int`. La declaración `vector[]` es un puntero de igual forma que `*vector`. Se utilizará para almacenar la dirección del primer elemento (`[0]`) del vector. La primera función `ordenar_burbuja` será utilizada por `main()` para realizar la ordenación del vector. La siguiente función, `imprimir_vector`, se utiliza para imprimir el vector ordenado.

La función `main()` lee nueve números del usuario del programa:

```
int numeros[numero_maximo];
int indice, comparaciones;

printf("Deme nueve numeros y los ordenare: n");
for (indice = 0; indice < numero_maximo; indice++)
{
    printf("Numero[%d] = ", indice);
    scanf("%d", &numeros[indice]);
}
```

El extracto del programa anterior define un vector denominado `numeros` que consta de `numero_maximo` elementos. Esto significa que el primer elemento del vector será `numeros[0]` y el último `numeros[8]`. Cuando el usuario introduce los nueve números, el programa llama a las dos funciones —una para realizar la ordenación, y otra para imprimir el vector ordenado. Observe que el parámetro real que se pasa a las funciones es la dirección del primer elemento. Recuerde que ésta contiene la dirección del primer elemento del vector. Esto es similar a utilizar `&numeros[0]` como parámetro.

```
comparaciones = ordenar_burbuja(numeros);
imprimir_vector(numeros);
```

La función `ordenar_burbuja` declara cuatro variables: `indice`, `seguir`, `temp`, y `comp`. La variable `indice` se utiliza como índice para cada elemento del vector. `seguir` se utiliza para saber si el proceso de ordenación ha concluido. `temp` se utiliza para almacenar de forma temporal el valor de un elemento del vector mientras se intercambia con otro elemento del vector. En `comp` se almacena el número de comparaciones que tienen lugar. Se puede pensar en la variable `seguir` como una variable que toma dos valores (SÍ o NO).

```
int indice;
int seguir;
int temp;
int comp = 0;
```

En el cuerpo de la función `ordenar_burbuja`, se utiliza un bucle `do` de C para recorrer de forma repetida el vector y comprobar si se necesita intercambiar dos elementos consecutivos del mismo. El bucle comienza asignando a la variable `seguir` el valor 0, es decir, el valor falso.

```
do
{
    seguir = 0;
```

A continuación se utiliza un bucle `for` para analizar cada elemento del vector.

```
for (indice = 0; indice < numero_maximo; indice ++)
```

Dentro de este bucle `for`, se realiza una comparación de cada elemento del vector y se verifica que no se comprueban más elementos que el número máximo de elementos del vector (`vector[8]` es el último elemento del vector).

```
if ((vector[indice] > vector[indice + 1])
    && (indice != numero_maximo - 1))
```

Si se necesita realizar un intercambio, se lleva a cabo la siguiente fase:

```
{
    temp = vector[indice];
    vector[indice] = vector[indice + 1];
    vector[indice + 1] = temp;
    seguir = 1;
}
```

Observe que en este caso se hace uso de la variable `temp`. Su funcionamiento es similar a disponer de una taza de leche y de un vaso de zumo de naranja y tener que poner la leche en el vaso y el zumo de naranja en la taza. Una forma de llevar a cabo este proceso es hacer uso de un tercer recipiente. Se vierte la leche en este recipiente, a continuación se vierte el zumo en la taza y por último se echa la leche en el vaso. La variable `temp`, se utiliza en este caso para llevar a cabo un proceso similar. Si se realiza un intercambio de elementos, la variable `seguir` se pone a 1 (verdadero), lo que significa que se tiene que repetir de nuevo el bucle `do`.

```
)while (seguir);
```

Recuerde que la forma de asegurarse de que el proceso de ordenación ha concluido viene dado en el momento en el que no es necesario realizar ningún intercambio dentro del bucle.

La función que imprime el vector, simplemente utiliza un contador como índice en un bucle `for` de C para imprimir los valores ordenados en la pantalla.

```
for (indice = 0; indice < numero_maximo; indice++)
    printf("Numero[%d] = %d\n", indice, vector[indice]);
```

Nótese que el número de comparaciones realizadas (72) es casi igual al número de elementos elevado al cuadrado ($9^2 = 81$). De esta manera, la ordenación por el método de la burbuja de 50 números requeriría del orden de 2500 comparaciones.

Ordenación por el método de la cubeta

A diferencia de la ordenación por el método de la burbuja, la cual puede ordenar números de cualquier tamaño, la ordenación por el **método de la cubeta** requiere que los números a ordenar se encuentren dentro de un intervalo predeterminado. Por ejemplo, considere los mismos números utilizados en la ejecución del Programa 7.10:

6 7 5 8 4 9 3 0 2

Ninguno de estos números es mayor que 9. Además, no se repite ninguno de los números. Estos son dos requisitos importantes para este método de ordenación. Conocer por adelantado que los números a ordenar no se encuentran duplicados y que todos son menores o iguales que 9, permite iniciar un *vector de cubeta* de diez elementos, como se muestra en la Figura 7.5. Como se puede observar, a cada elemento del vector se le asigna el valor inicial 0, el cual se utiliza para representar una *cubeta vacía*.

El proceso de ordenación utilizando el método de la cubeta se realiza ahora de forma muy sencilla. Para cada elemento del vector inicial, se asocia un elemento en el vector de cubeta igual a 1 (o un entero distinto de cero). Este valor representa una *cubeta llena*. Por ejemplo, la Figura 7.6 muestra el vector de cubeta después de procesar el primer elemento del vector (6). Observe el valor 1 en el elemento situado en la posición [6].

Después de procesar todos los elementos, el vector de cubeta contiene los valores que se muestran en la Figura 7.7. El único elemento con valor cero indica que en el vector de entrada no se encontraba el 1.

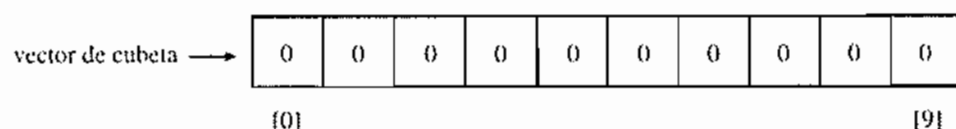


Figura 7.5. Inicializando el vector de cubeta.

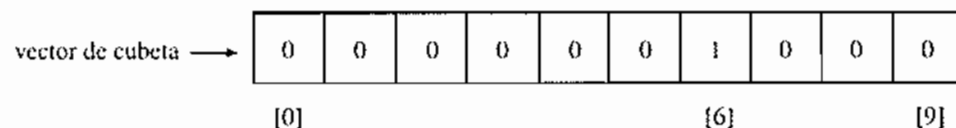


Figura 7.6. Vector de cubeta después de procesar el primer elemento.

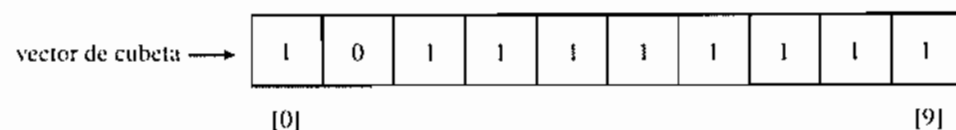


Figura 7.7. Vector de cubeta final.

Para imprimir el vector ordenado, sólo es necesario recorrer el vector de cubeta desde el comienzo, e imprimir el índice de cada elemento que contenga un 1.

El Programa 7.11 muestra la implementación de la ordenación por el método de la cubeta. Debido a que los números utilizados son los mismos que los del Programa 7.10, éstos no son introducidos por el usuario, sino que son asignados al vector de forma directa.

Programa 7.11 #include <stdio.h>

```
int ordenacion_cubeta(int ent[], int k, int sal[]);

main()
{
    int vector_ent[20] = {6, 7, 5, 8, 4, 9, 3, 0, 2};
    int n = 9;
    int vector_sal[20];
    int i, comparaciones;

    printf("Vector no ordenado => ");
    for(i = 0; i < n; i++)
        printf("%4d", vector_ent[i]);
    printf("\n");
    comparaciones = ordenacion_cubeta(vector_ent, n, vector_sal);
    printf("Vector ordenado => ");
    for(i = 0; i < 20; i++)
        if(vector_sal[i] != 0)
            printf("%4d", i);
    printf("\nEl numero de comparaciones es %d", comparaciones);
}

int ordenacion_cubeta(int ent[], int k, int sal[])
{
    int j;

    for(j = 0; j < 20; j++)
        sal[j] = 0;
    for(j = 0; j < k; j++)
        sal[ent[j]] = 1;
    return(k);
}
```

La ejecución del Programa 7.11 produce la siguiente salida:

```
Vector no ordenado => 6 7 5 8 4 9 3 0 2
Vector ordenado     => 0 2 3 4 5 6 7 8 9
El número de comparaciones es 9
```

El número de comparaciones es engañoso, debido a que en realidad no se realiza ninguna comparación. El número de «comparaciones» en este método de ordenación debería considerarse como número de «pasos». Recuerde que en la ordenación por el

método de la burbuja (Programa 7.10) se necesitaban 72 comparaciones. Si estas comparaciones se consideraran como 72 pasos, entonces la ordenación por el método de la cubeta es claramente más eficiente. No olvide, sin embargo, que este tipo de ordenación sólo se puede utilizar en situaciones limitadas.

Ordenación por fusión

La técnica de **ordenación por fusión** (*merge sort*) presenta una eficiencia que se encuentra entre las obtenidas en las dos técnicas descritas anteriormente. A diferencia de la ordenación por el método de la cubeta, la ordenación por fusión puede aplicarse a cualquier conjunto de números, sin importar el tamaño ni la existencia de duplicados. El ahorro en el número de comparaciones viene del hecho de que la lista entera de números de entrada se divide en listas más pequeñas, que requieren menor número de comparaciones para ordenarlas y en la mezcla de ellas al final. La Figura 7.8 muestra la primera etapa de este proceso. El vector de entrada es idéntico al utilizado en el Programa 7.11.

En la etapa 1 se determina la mitad del vector de entrada. En este caso el elemento situado en la cuarta posición determina esta mitad. A continuación el vector se rompe en dos subvectores que contienen los elementos del 0 al 4 y los elementos del 5 al 8.

En la etapa 2, cada subvector se divide a su vez en dos nuevas partes. Observe que en este momento tres de los subvectores contienen solamente dos elementos. En esta situación se realiza una única comparación sobre cada uno de los tres subvectores de dos elementos para determinar el orden de cada elemento.

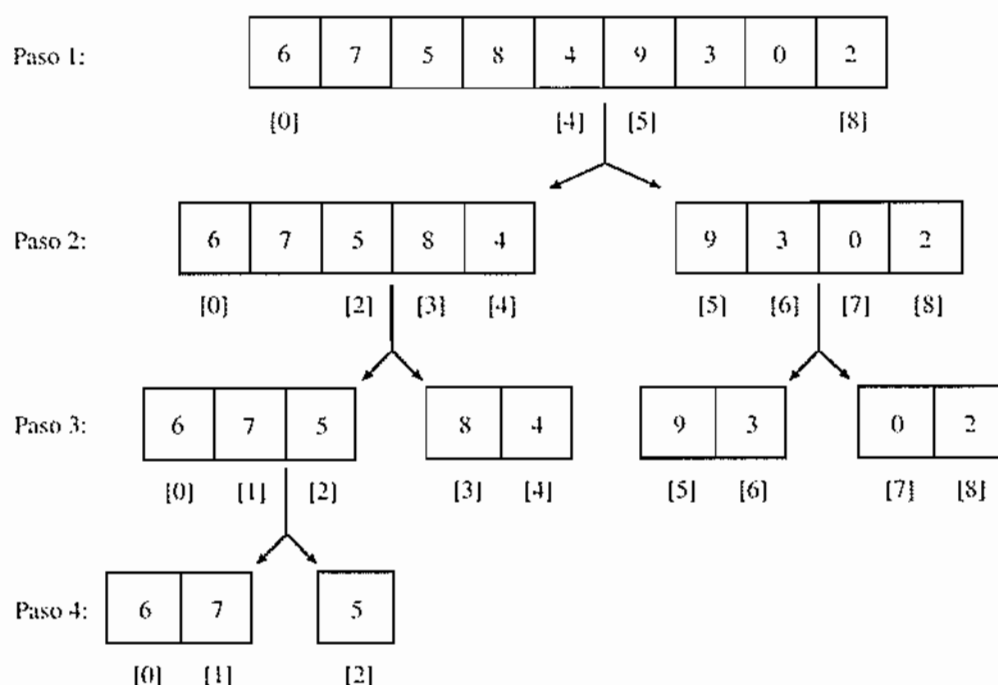


Figura 7.8. Descomposición del vector de entrada en la ordenación por fusión.

En la etapa 3, el subvector que contiene los elementos 0, 1 y 2 se divide de nuevo, dando lugar a los vectores que se muestran en la etapa 4. Los subvectores resultantes tienen uno y dos elementos cada uno de ellos. Sobre el vector de dos elementos se realiza una comparación para obtener un subvector ordenado. Sobre el subvector de 1 elemento no se necesita ninguna comparación. De esta manera, aunque se comenzó con un vector compuesto por nueve números, sólo se han realizado cuatro comparaciones para determinar la ordenación relativa.

La Figura 7.9 muestra los cinco subvectores obtenidos antes al comienzo del proceso de mezcla. Los números almacenados en el tercer y cuarto vectores se han intercambiado como resultado del proceso de comparación. De esta forma, los cinco vectores están ordenados.

La mezcla de los vectores ordenados es un proceso relativamente simple. Se van comparando los elementos de cada vector uno cada vez, manteniéndose un puntero por cada vector que indica la posición actual de comparación. Cuando se realiza la comparación, se escribe en el vector resultante el valor más pequeño y se avanza el puntero del vector al que pertenecía dicho elemento. Si se alcanza el final de un vector, los restantes números del otro vector se copian directamente al vector resultante. Por ejemplo, dados los siguientes vectores ordenados, junto con los respectivos punteros:

17	22	24	y	15	19	21
^				^		

Puesto que 17 es mayor que 15, se escribirá el valor 15 en el resultado y se avanza el puntero del segundo vector, lo que resulta en lo siguiente:

17	22	25	y	15	19	21
^				^		

En este caso, 19 es mayor que 17, por lo tanto, se selecciona el valor 17 y se avanza el puntero del primer vector, dando lugar a la siguiente situación:

17	22	25	y	15	19	21
^				^		

La comparación de los valores causa que se almacene 19 en el resultado y se avance el puntero del segundo vector, dejando los punteros de la siguiente forma:

17	22	25	y	15	19	21
^				^		

En esta situación, se escribe 21 y se detecta que se ha llegado al final del segundo vector por lo que se copian directamente los restantes elementos (22 y 25) del primero.

La Figura 7.10 muestra el resultado de aplicar este proceso de mezcla a los vectores del ejemplo. El vector resultante contiene los nueve elementos ordenados correcta-



Figura 7.9. Subvectores previos a la fusión.

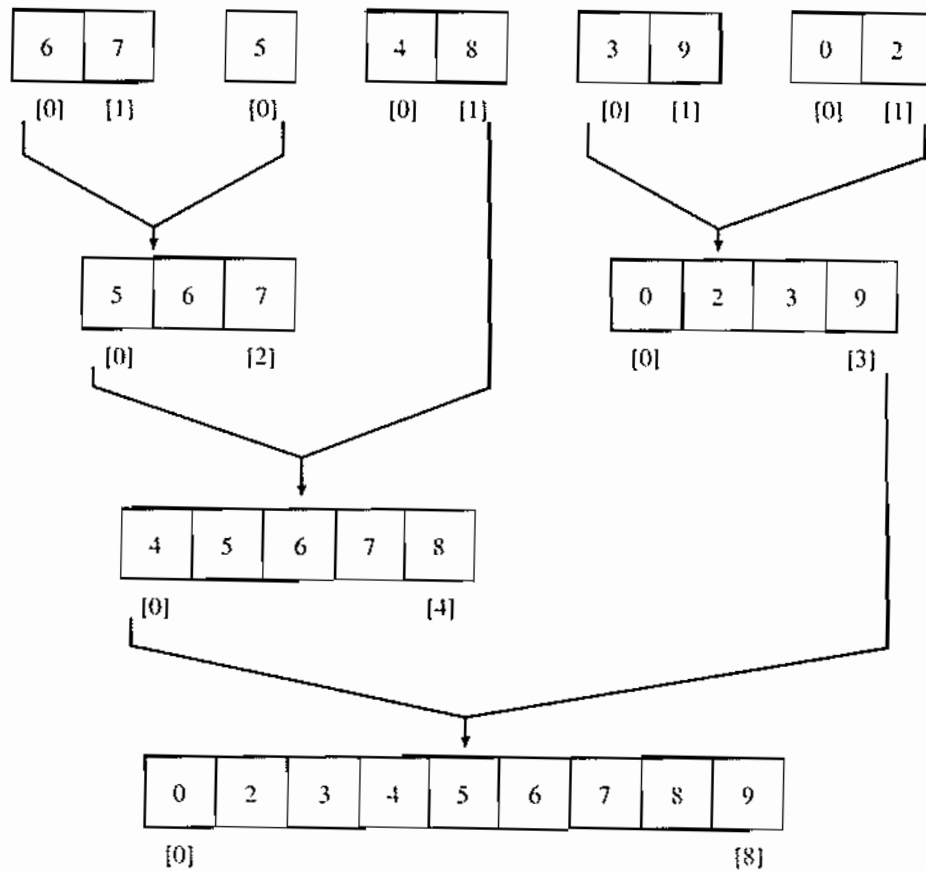


Figura 7.10. Fundiendo subvectores.

mente. En el Programa 7.12 se muestra cómo se puede implementar este método de ordenación.

Programa 7.12 `#include <stdio.h>`

```
void ordenar_por_fusion(int ent[], int a, int b, int sal[]);
void mostrar_vector(int ent[], int k);
void mezclar_vectores(int ent1[], int ent2[], int n1, int n2,
                      int sal[]);
int comparaciones = 0;

main()
{
    int vector_ent[20] = {6, 7, 5, 8, 4, 9, 3, 0, 2};
    int n = 9;
    int vector_sal[20];

    printf("Vector no ordenado => ");
    mostrar_vector(vector_ent, n);
    ordenar_por_fusion(vector_ent, 0, n-1, vector_sal);
    printf("Vector ordenado => ");
```

```

        mostrar_vector(vector_sal,n);
        printf("\nEl numero de comparaciones es %d",comparaciones);
    }
void mostrar_vector(int ent[], int k)
{
    int i;
    for(i = 0; i < k; i++)
        printf("%4d",ent[i]);
    printf("\n");
}

void ordenar_por_fusion(int ent[], int a, int b, int sal[])
{
    int m;
    int sal1[20], sal2[20];

    /* Comprobar si el vector contiene solo un elemento */
    if(a == b)
        sal[0] = ent[a];          /* Devuelve el único elemento */
    else
        /* Comprobar si el vector contiene dos elementos. */
        if(1 == (b - a))
        {
            if(ent[a] <= ent[b])    /* No intercambiar los elementos. */
            {
                sal[0] = ent[a];
                sal[1] = ent[b];
            }
            else                    /* Intercambiar los elementos. */
            {
                sal[0] = ent[b];
                sal[1] = ent[a];
            }
            comparaciones++;
        }
    else
    {
        /* Dividir el vector de tres o mas elementos. */
        m = a + (b - a)/2;          /* Cálculo de la mitad */
        ordenar_por_fusion(ent,a,m,sal1); /* Ordenar la primera
                                           mitad */
        ordenar_por_fusion(ent,m+1,b,sal2); /* Ordenar la segunda
                                           mitad */

        /* Mezclar las dos mitades. */
        mezclar_vectores(sal1,sal2,1+m-a,b-m,sal);
    }
}

void mezclar_vectores(int ent1[], int ent2[], int n1, int n2, int sal[])
{

```



```

int i = 0,j = 0,k = 0;
while((i < n1) && (j < n2))
{
    /* Comprobar si el primer elemento del vector es
       el más pequeño */
    if(ent1[i] <= ent2[j])
    {
        sal[k] = ent1[i];
        i++;
        /* Actualizar el índice */
    }
    else /* El segundo elemento es más pequeño. */
    {
        sal[k] = ent2[j];
        j++;
        /* Actualizar el índice. */
    }
    k++;
    /* Actualizar el índice de salida. */
    comparaciones++;
}

/* Comprobar si hay elementos a la izquierda
   en el primer vector. */
if(i != n1)
{
    do
        /* Escribir los elementos restantes de ent1
           al vector de salida. */
        {
            sal[k] = ent1[i];
            i++;
            k++;
        } while(i < n1);
}
else
    /* Escribir los elementos restantes de ent2
       al vector de salida. */
    {
        do
        {
            sal[k] = ent2[j];
            j++;
            k++;
        } while(j < n2);
    }
}

```

La ejecución del programa produce lo siguiente:

Vector no ordenado => 6 7 5 8 4 9 3 0 2
 Vector ordenado => 0 2 3 4 5 6 7 8 9

El numero de comparaciones es 21

El número total de comparaciones requeridas puede descomponerse en dos partes: las comparaciones necesarias para realizar la partición de los vectores y las utilizadas para el proceso de mezcla. El resultado es significativamente mejor que el obtenido por el método de la burbuja (19 comparaciones frente a 72).

Al analizar el programa surge la siguiente pregunta: ¿Dónde se almacenan los vectores intermedios? Para poder contestarla, hay que darse cuenta de que el programa usa un esquema recursivo. Así, la primera llamada a la función `ordenar_por_fusion()` es la siguiente:

```
ordenar_por_fusion(vector_ent, 0, 8, vector_sal)
```

Esta función reserva espacio para los vectores `sal1` y `sal2`. Es importante resaltar que al tratarse de variables automáticas, cada invocación de la función tendrá una copia propia de dichos vectores. La función realiza la partición del vector y se invoca de forma recursiva una vez por cada parte:

```
ordenar_por_fusion(ent, 0, 4, sal)
ordenar_por_fusion(ent, 5, 8, sal)
```

La ejecución de la función `ordenar_por_fusion()` con la primera parte del vector (0-4), realiza a su vez las siguientes llamadas:

```
ordenar_por_fusion(ent, 0, 2, sal)
ordenar_por_fusion(ent, 3, 4, sal)
```

La ejecución de `ordenar_por_fusion()` con la segunda parte del vector (5-8), lleva a cabo las siguientes llamadas recursivas:

```
ordenar_por_fusion(ent, 5, 6, sal)
ordenar_por_fusion(ent, 7, 8, sal)
```

Por último, la ejecución de `ordenar_por_fusion()` con los elementos del vector del 0 al 2, produce las siguientes llamadas:

```
ordenar_por_fusion(ent, 0, 1, sal)
ordenar_por_fusion(ent, 2, 2, sal)
```

Hay un total de nueve llamadas a la función `ordenar_por_fusion()` y llega a haber hasta tres llamadas activas simultáneamente (en el caso de la división de la primera parte del vector [0-4]), que se corresponde con la profundidad máxima del árbol de recursividad. Este esquema recursivo requiere, por lo tanto, una cantidad de memoria sustancial, lo que puede impedir su aplicación a vectores de gran tamaño.

Ordenación rápida

El método de **ordenación rápida** (*quick sort*) es el último que se presentará en este capítulo. Se le considera el método más eficiente cuando se usa de forma óptima. Como se verá más adelante, la versión que se utilizará en esta sección no es la óptima, proporcionando un rendimiento similar a la ordenación por fusión.

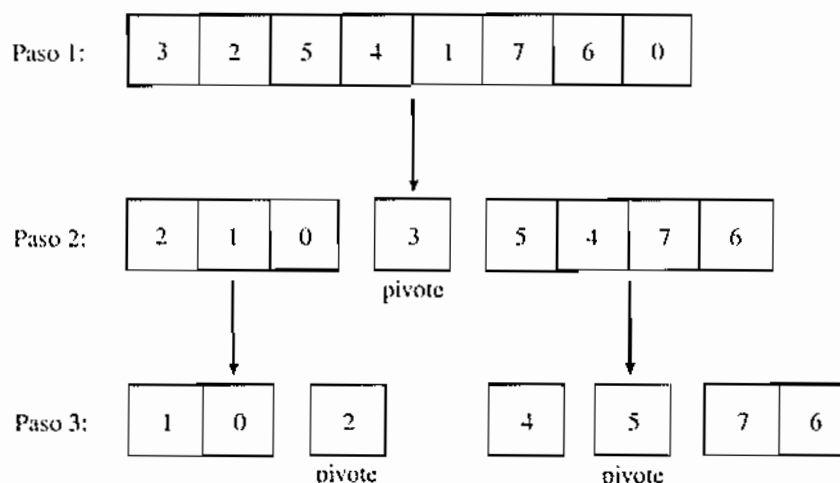


Figura 7.11. División de un vector durante la ordenación rápida.

La Figura 7.11 muestra parte del proceso de ordenación de un vector usando este método. La idea básica de este algoritmo es usar un elemento como **pivote**, de manera que el vector pueda dividirse en dos partes: una parte que contenga los elementos menores o iguales que el pivote y la otra que contenga los estrictamente mayores. Como se observa en la figura, en el primer paso se selecciona como pivote el primer elemento del vector (3). Una vez seleccionado, se compara con el resto de los elementos dando lugar a los nuevos vectores mostrados en el segundo paso. Observe que el pivote ya está colocado en la posición correcta entre los dos vectores generados.

En el segundo paso, se seleccionan pivotes para los dos vectores generados y se realiza la partición de dichos vectores de acuerdo a los pivotes elegidos, generándose los vectores mostrados en el tercer paso. Este tratamiento se aplica a los vectores que se van generando hasta que sólo contengan como máximo dos elementos. Si el vector tiene un solo elemento, el proceso sobre dicho vector ha terminado. Si contiene dos elementos, como sucede con los vectores [1,0] y [7,6] del tercer paso, sólo es necesario realizar una comparación para ordenarlos.

Al final del tercer paso, todos los números están en su posición correcta siendo sólo necesario combinarlos de nuevo en un único vector. A diferencia del método de ordenación por difusión, no se necesita ninguna comparación para construir el vector conjunto puesto que los elementos ya están totalmente ordenados.

El Programa 7.13 implementa este método de ordenación. El programa usa un esquema recursivo. La función `ordenacion_rapida()` divide el vector que recibe como parámetro de entrada en dos vectores por lo que necesita reservar espacio para almacenar los vectores resultantes de la partición (`ent1` y `ent2`).

Programa 7.13 `#include <stdio.h>`

```
void ordenacion_rapida(int ent[], int a, int b, int sal[]);
void mostrar_vector(int ent[],int k);
int comparaciones = 0;
```

```

main()
{
    int vector_ent[20] = {6, 7, 5, 8, 4, 9, 3, 0, 2};
    int n = 9;
    int vector_sal[20];

    printf("Vector no ordenado => ");
    mostrar_vector(vector_ent,n);
    ordenacion_rapida(vector_ent,0,n-1,vector_sal);
    printf("Vector ordenado => ");
    mostrar_vector(vector_sal,n);
    printf("\nEl numero de comparaciones es %d",comparaciones);
}

void mostrar_vector(int ent[], int k)
{
    int i;

    for(i = 0; i < k; i++)
        printf("%4d",ent[i]);
    printf("\n");
}

void ordenacion_rapida(int ent[], int a, int b, int sal[])
{
    int pivote, i = 0, j = 0, k = 1, z = 0;
    int ent1[20], ent2[20];
    int sal1[20], sal2[20];

    if(b != -1)          /* Sólo un elemento. */
        if(a == b)
            sal[0] = ent[a];
        else
            /* Sólo dos elementos. */
            if(1 == (b - a))
            {
                if(ent[a] <= ent[b])          /* No intercambiar */
                {
                    sal[0] = ent[a];
                    sal[1] = ent[b];
                }
                else                          /* Intercambiar */
                {
                    sal[0] = ent[b];
                    sal[1] = ent[a];
                }
                comparaciones++;
            }
        else          /* Tres o más elementos. */
        {
            pivote = ent[0];          /* Tomar pivote */

```

```

while(k <= b)
{
    if(pivote > ent[k]) /* Comparar pivote */
    {
        /* Escribir el vector de salida más pequeño. */
        ent1[i] = ent[k];
        i++;
    }
    else /* Pivote es más pequeño. */
    {
        /* Escribir el vector de salida más grande. */
        ent2[j] = ent[k];
        j++;
    }
    k++; /* Actualizar el índice */
    comparaciones++;
}

/* Ordenar la partición más pequeña */
ordenacion_rapida(ent1,0,i-1,sal1);

/* Ordenar la partición más grande */
ordenacion_rapida(ent2,0,j-1,sal2);

/* Escribir a la salida el vector más pequeño. */
for(k = 0; k < i; k++)
{
    sal[z] = sal1[k];
    z++;
}
sal[z] = pivote; /* Escribir el pivote en la salida */
z++;

/* Escribir el vector más grande en la salida */
for(k = 0; k < j; k++)
{
    sal[z] = sal2[k];
    z++;
}
}
}

```

La ejecución del programa produce lo siguiente:

Vector no ordenado => 6 7 5 8 4 9 3 0 2
 Vector ordenado => 0 2 3 4 5 6 7 8 9

El numero de comparaciones es 21

El número de comparaciones requeridas es ligeramente mayor que en el caso de la ordenación por difusión que requería 19. Un punto interesante es que si se modifica el orden del vector original, se obtiene un resultado diferente en cuanto a la eficiencia. Así, si se intercambian los valores 5 y 3 en el vector original, se produce el siguiente resultado:

```
Vector no ordenado => 6 7 3 8 4 9 5 0 2
Vector ordenado    => 0 2 3 4 5 6 7 8 9
```

El número de comparaciones es 17

Se ha disminuido en 4 el número de comparaciones requeridas. Sin embargo, el algoritmo de difusión sigue necesitando 19 comparaciones para ordenar el vector. Por lo tanto, en el método de ordenación rápida, el orden de los elementos del vector inicial afecta a la eficiencia del algoritmo. La ordenación rápida óptima no selecciona como pivote el primer elemento del vector, sino el elemento que corresponde con la *mediana* de la serie de números que se pretende ordenar. Por ejemplo, dados los siguientes números:

6 7 5 8 4 9 3 0 2

el primer pivote sería el 5, puesto que hay tantos valores por encima de 5 como por debajo. El uso de este valor mantiene iguales los tamaños de los vectores resultantes de la partición, lo que resulta en un número menor de niveles de recursividad y, por lo tanto, en un número menor de comparaciones.

qsort()

C proporciona una función interna `qsort()` definida en `<stdlib.h>`, que implementa el algoritmo de ordenación rápida para cualquier tipo de datos numéricos. La función `qsort()` tiene cuatro parámetros: la dirección del vector que se desea ordenar, el número de elementos del vector, el tamaño de cada elemento y la función de comparación.

Se necesita especificar el tamaño de cada elemento, ya que `qsort()` puede ordenar vectores de diferentes tipos. La función de comparación es invocada por `qsort()` para comparar los distintos elementos del vector y debe devolver alguno de los siguientes valores al comparar *a* y *b*:

- un entero negativo si *a* es menor que *b*
- 0 si *a* es igual a *b*
- un entero positivo si *a* es mayor que *b*

El Programa 7.14 muestra cómo usar esta función.

Programa 7.14

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

void imprimir_vector(int in[],int k);
```

```

int cmp(const void *a, const void *b);
int comparaciones = 0;

main()
{
    int vector_ent[20] = {6, 7, 5, 8, 4, 9, 3, 0, 2};
    int n = 9;

    printf("Vector no ordenado => ");
    imprimir_vector(vector_ent, n);
    qsort(vector_ent, n, sizeof(vector_ent[0]), cmp);
    printf("Vector ordenado => ");
    imprimir_vector(vector_ent, n);
    printf("\nEl numero de comparaciones es %d", comparaciones);
}

int cmp(const void *a, const void *b)
{
    comparaciones++;
    return(*((int *) a) - *((int *) b));
}

void imprimir_vector(int in[], int k)
{
    int i;

    for(i = 0; i < k; i++)
        printf("%4d", in[i]);
    printf("\n");
}

```

El Programa 7.14 muestra cómo usar la función `qsort()` para ordenar un vector. La función denominada `cmp()` tiene el siguiente prototipo:

```
int cmp(const void *a, const void *b);
```

que indica que la función recibirá como argumentos dos punteros a un tipo sin especificar (`void`). Dentro de la función se usa la técnica de conversión de tipos para comparar los valores teniendo en cuenta el tipo de datos con el que se trabaja (en este caso, entero).

```
return(*((int *) a) - *((int *) b));
```

Si se necesitase comparar dos números de tipo real, la sentencia debería ser:

```
return(*((float *) a) - *((float *) b));
```

Observe también el uso de la función `sizeof()` en la llamada a `qsort()`:

```
qsort(vector_ent, n, sizeof(vector_ent[0]), cmp);
```

De esta forma, se especifica el tamaño de cada elemento del vector, en este caso el tamaño de un entero.

La ejecución del programa produce lo siguiente:

```
Vector no ordenado => 6 7 5 8 4 9 3 0 2
Vector ordenado    => 0 2 3 4 5 6 7 8 9
```

El numero de comparaciones es 23

Como se puede observar, la eficiencia de la rutina `qsort()` es similar a la obtenida con el método por difusión y con la ordenación rápida.

Conclusión

En esta sección se han mostrado diferentes técnicas para ordenar números, evaluando la eficiencia de las mismas basándose en el número de comparaciones que realizan para llevar a cabo la ordenación. Compruebe su comprensión de los conceptos presentados en esta sección mediante el siguiente repaso.

Repaso de la Sección 7.3

1. ¿Por qué la ordenación por fusión requiere menos comparaciones que la ordenación por el método de la burbuja?
2. ¿Cuál es el número mínimo de veces que un algoritmo de ordenación debe recorrer la lista de números para ordenarlos completamente? ¿En qué circunstancias sucederá esto?
3. ¿Qué determina si se está ordenando una lista de menor a mayor o de mayor a menor?
4. Dado que se precisa ordenar de menor a mayor la siguiente secuencia de números:

8 7 3 1

¿cuántas veces se debe recorrer la lista para cada uno de los siguientes métodos de ordenación?

- a) método de la burbuja
 - b) método de la cubeta
 - c) ordenación por difusión
5. ¿Cuál es el motivo de que el método de la cubeta sea tan eficiente? ¿Qué limitaciones tiene este método?
 6. ¿Cuántos niveles de recursividad son necesarios para que la ordenación por fusión descomponga la siguiente lista de números?

0 3 8 1 9 2 6 4 7 5

7. ¿Qué diferencias hay entre la ordenación por fusión y la ordenación rápida?
8. ¿Hay diferencia entre ordenar números y ordenar cadenas de caracteres?

7.4. MATRICES NUMÉRICAS

Presentación

En las últimas secciones, se ha trabajado con vectores de una sola dimensión. Sin embargo, en muchas aplicaciones se usan vectores con más de una dimensión, denominados **vectores multidimensionales** o **matrices**. Esta sección se centra en el caso de dos dimensiones, presentando una aplicación que usa este tipo de estructura de datos.

Idea básica

Sólo se puede dar valor inicial a vectores estáticos o externos. El Programa 7.15 muestra cómo se realiza esta operación para una cadena de caracteres.

Programa 7.15 `#include <stdio.h>`

```
main()
{
    static char vector[6] = {'H', 'o', 'l', 'a'};
    printf("%s", vector);
}
```

La ejecución del programa produce la siguiente salida:

Hola

Nótese que los valores iniciales están separados por comas y encerrados entre llaves {}. Este método puede utilizarse para cualquier vector aunque, como ya conoce el lector, no es el más recomendable para dar valor inicial a cadenas de caracteres.

Para dar valor inicial a matrices de dos dimensiones, se usa un método similar. Como muestra la Figura 7.12, las matrices de dos dimensiones pueden visualizarse como un tablero de ajedrez donde cada casilla contiene un dato y se identifica con un número de fila y columna único.

El Programa 7.16 muestra cómo dar valor inicial a una matriz de dos dimensiones.

Programa 7.16 `#include <stdio.h>`

```
main()
{
    static int matriz[2][3] = {
        {10, 20, 30},
        {11, 21, 31}
    };
}
```

		(Columna)						
		0	1	2	3	4	5	6...
(Fila)	0	[0][0]	[0][1]	[0][2]	[0][3]	...		
	1	[1][0]	[1][1]	[1][2]	...			
	2	[2][0]	[2][1]	...				
	3	[3][0]	...					
	4	⋮						

Figura 7.12. Concepto de matriz (array).

```

int fila;
int columna;

for(fila = 0; fila < 2; fila++)
{
    for(columna = 0; columna < 3; columna++)
        printf("%5d",matriz[fila][columna]);
    printf("\n\n");
}

```

La ejecución del programa produce:

```

10   20   30
11   21   31

```

Análisis del programa

En primer lugar se declara y da valor inicial a una matriz de dos dimensiones:

```

static int matriz[2][3] = {
    {10, 20, 30},
    {11, 21, 31}
};

```

Los valores [2][3] indican que se trata de una matriz de dos dimensiones con dos **filas** (numeradas 0 y 1) y tres **columnas** (numeradas 0, 1 y 2). Obsérvese que el número de parejas de símbolos [] indica cuántas dimensiones tiene la matriz. En C las matrices se almacenan en memoria por filas. A este tipo de disposición de los datos en memoria se le denomina **almacenamiento por filas** y se ilustra en la Figura 7.13.

A la hora de fijar los valores iniciales de la matriz, se ha especificado cada fila en una línea del programa. Se podría haber realizado también de la siguiente forma:

```

{{10, 20, 30}, {11, 21, 31}}

```

sin embargo, la forma usada en el programa es más descriptiva.

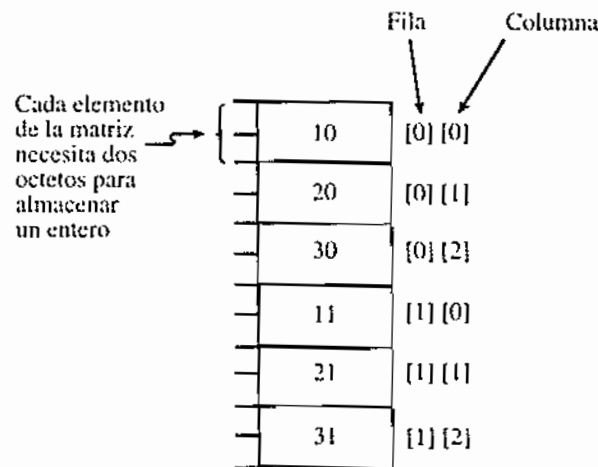


Figura 7.13. Asignación de memoria de fila y columnas para una matriz bidimensional.

A continuación, se declaran dos variables:

```
int fila;
int columna;
```

Estas variables se usarán para recorrer todos los elementos de la matriz e imprimir el valor almacenado en cada uno de ellos mediante los siguientes bucles `for` anidados:

```
for(fila = 0; fila < 2; fila++)
{
    for(columna = 0; columna < 3; columna++)
        printf("%5d",matriz[fila][columna]);
    printf("\n\n");
}
```

Puesto que el índice de `fila` comienza en 0 y termina en 1, y el índice de `columna` empieza en 0 y llega hasta 2, los dos bucles accederán a todos los elementos de la matriz usando la expresión:

```
matriz[fila][columna]
```

Como puede verse en la Figura 7.13, en la primera iteración del bucle, `fila` es igual a 0 y `columna` es igual a 0, por lo tanto, se imprimirá mediante la función `printf()` el valor almacenado en el elemento `matriz[0][0]`, o sea, 10. El próximo elemento que se imprime corresponde con `matriz[0][1]` que tiene un valor de 20. Este proceso continúa hasta que el contador asociado al bucle anidado alcanza un valor de 3. En ese momento, se termina dicho bucle, se ejecuta la sentencia `printf("\n\n");` que produce dos saltos de línea en la salida y se incrementa el contador asociado al bucle externo, repitiéndose nuevamente el proceso con la segunda fila.

Se usa un **especificador de ancho de campo** en la siguiente sentencia:

```
printf("%5d",matriz[fila][columna]);
```

El valor 5 que aparece en %5d especifica que se usen cinco caracteres para imprimir el valor.

Aplicaciones con matrices

Las matrices se pueden aplicar a la resolución de numerosos problemas de carácter técnico. En muchas ocasiones, se requiere el tratamiento de los elementos de la matriz y, por lo tanto, es importante alcanzar cierta soltura en la programación de este tipo de operaciones.

Uno de los procesos más sencillos y típicos es determinar la suma de una columna o fila de la matriz. El Programa 7.17 calcula la suma de cada columna y cada fila de una matriz. Para realizar el cálculo, este programa usa dos funciones: una que suma columnas y otra que suma filas.

Programa 7.17

```
void sumar_columnas(int matriz_ent[][3], int valor_columna[]);
void sumar_filas(int matriz_ent[][3], int valor_fila[]);

main()
{
    static int matriz[2][3] = {
                                {10,20,30},
                                {11,21,31}
                                };

    int fila;
    int columna;
    int valor_columna[3];
    int valor_fila[2];

    for(fila = 0; fila < 2; fila++)
    {
        for(columna = 0; columna < 3; columna++)
            printf("%5d", matriz[fila][columna]);
        printf("\n\n");
    }
    sumar_columnas(matriz, valor_columna);
    for(columna = 0; columna < 3; columna++)
        printf("La suma de la columna %d es %d\n", columna,
              valor_columna[columna]);
    sumar_filas(matriz, valor_fila);
    for(fila = 0; fila < 2; fila++)
        printf("La suma de la fila %d es %d\n", fila,
              valor_fila[fila]);
}
```

```

void sumar_columnas(int vector_ent[][3], int valor_columna[])
{
    int fila;
    int columna;

    for(columna = 0; columna < 3; columna++)
    {
        valor_columna[columna] = 0;
        for(fila = 0; fila < 2; fila++)
            valor_columna[columna] += vector_ent[fila][columna];
    }
}

void sumar_filas(int vector_ent[][3], int valor_fila[])
{
    int fila;
    int columna;

    for(fila = 0; fila < 2; fila++)
    {
        valor_fila[fila] = 0;
        for(columna = 0; columna < 3; columna++)
            valor_fila[fila] += vector_ent[fila][columna];
    }
}

```

La ejecución del Programa 7.17 produce la siguiente salida:

```

10   20   30
11   21   31
La suma de la columna 0 es 21
La suma de la columna 1 es 41
La suma de la columna 2 es 61
La suma de la fila 0 es 60
La suma de la fila 1 es 63

```

Análisis del programa

El programa comienza declarando el prototipo de dos funciones:

```

void sumar_columnas(int matriz_ent[][3], int valor_columna[]);
void sumar_filas(int matriz_ent[][3], int valor_fila[]);

```

Puesto que se usa una matriz de más de una dimensión, se necesita especificar en el parámetro formal el límite del resto de las dimensiones de la matriz. Esta exigencia de C se debe a la manera cómo se almacenan en memoria las matrices. En el caso de una matriz de dos dimensiones, el compilador necesita saber cuántas columnas hay en cada fila para acceder correctamente a la matriz.

Ambas funciones son de tipo void y tienen como segundo argumento un vector donde se almacenará el resultado de sumar cada columna (en el caso de `sumar_columnas()`) o cada fila (en el caso de `sumar_filas()`).

A continuación, se declara la matriz de la misma forma que en el Programa 7.16:

```
static int matriz[2][3] = {
                                {10,20,30},
                                {11,21,31}
                                };
```

Seguidamente se declaran las variables de la función `main()`:

```
int fila;
int columna;
int valor_columna[3];
int valor_fila[2];
```

Observe la declaración de los vectores `valor_columna` y `valor_fila`, que serán parámetros reales de las funciones `sumar_columnas()` y `sumar_filas()`, respectivamente. En estos vectores se almacenarán los resultados de la suma de cada columna y de cada fila.

A continuación, se imprime el contenido de la matriz con el mismo bucle que se usaba en el programa anterior.

```
for(fila = 0; fila < 2; fila++)
{
    for(columna = 0; columna < 3; columna++)
        printf("%5d", matriz[fila][columna]);
    printf("\n\n");
}
```

En este punto del programa, se llama a la función `sumar_columnas()`, que calculará la suma de las columnas de la matriz almacenando el resultado de cada columna en la posición correspondiente del vector `valor_columna`.

```
sumar_columnas(matriz, valor_columna);
```

Se analizan a continuación los detalles de esta función de tipo void.

```
void sumar_columnas(int vector_ent[][3], int valor_columna[])
{
    int fila;
    int columna;

    for(columna = 0; columna < 3; columna++)
    {
        valor_columna[columna] = 0;
        for(fila = 0; fila < 2; fila++)
            valor_columna[columna] += vector_ent[fila][columna];
    }
}
```

En cada iteración del bucle `for` más externo se va calculando la suma de una columna de la matriz. El bucle `for` interno realiza la suma de todos los elementos de la columna correspondiente usando el operador `+=`. Nótese que antes de entrar al bucle interno, se asigna un 0 a la variable `valor_columna[columna]` para asegurar que la suma acumulada empieza por 0. El resultado de la suma de cada columna se almacena en la posición correspondiente del vector `valor_columna`.

Es importante resaltar lo que sucede cuando se llama a esta función: Se le pasan a la función las direcciones de comienzo de la matriz `matriz`, donde están almacenados los valores de la matriz, y del vector `valor_columna`, donde la función almacenará los resultados de sumar cada fila de la matriz.

Cuando termina la función `sumar_columnas()`, los valores calculados se imprimen usando el siguiente bucle:

```
for(columna = 0; columna < 3; columna++)
    printf("La suma de la columna %d es %d\n", columna,
           valor_columna[columna]);
```

El tratamiento de las filas de la matriz es muy similar y, por lo tanto, se deja como ejercicio para el lector el análisis del fragmento del programa que realiza dicho tratamiento.

La multiplicación de matrices es otra típica operación sobre matrices. El principal requisito para poder multiplicar dos matrices es que el número de columnas de la primera matriz sea igual al número de filas de la segunda. Así, por ejemplo, una matriz cuadrada (una matriz con el mismo número de filas que de columnas) puede multiplicarse por sí misma por cumplir el anterior requisito. Dados los siguientes ejemplos de multiplicación de matrices:

- $A[2][3] * B[3][4]$
- $C[1][4] * D[4][2]$
- $E[2][3] * F[4][3]$

Solamente se pueden llevar a cabo las multiplicaciones de $A*B$ y $C*D$, ya que satisfacen el requisito antes expresado. El producto de $E*F$ no es factible puesto que E tiene 3 filas y F 4 columnas.

La Figura 7.14 intenta mostrar el motivo de este requisito. En ella se representa la multiplicación de dos matrices ($MATA$ y $MATB$) que da como resultado una tercera matriz ($MATC$). Para obtener un determinado elemento de $MATC$, hay que multiplicar cada elemento de la fila implicada de la matriz $MATA$ por el elemento correspondiente de la columna implicada de $MATB$ y sumar el resultado de estos productos. El requisito expuesto asegura que el número de elementos de las filas de una matriz coincide con el de las columnas de la otra.

El producto de matrices implica, por lo tanto, la multiplicación de cada fila de la primera matriz por cada columna de la segunda. La Figura 7.15 muestra un ejemplo de este proceso que es conveniente que el lector estudie para comprender correctamente cómo se realiza esta operación.

El Programa 7.18 realiza la multiplicación de una matriz de 2 filas y 3 columnas por una de 3 filas y 2 columnas, dando como resultado una matriz de 2 filas y 2

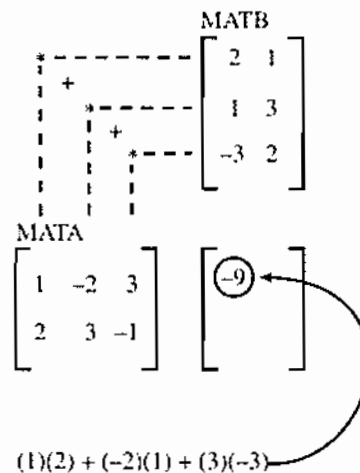


Figura 7.14. Encontrando un elemento de un producto de matrices.

columnas. Las dimensiones de la matriz resultante quedan determinadas por las de las matrices que se multiplican: su número de filas es el mismo que el de la primera matriz y su número de columnas es el mismo que el de la segunda.

Programa 7.18 #include <stdio.h>

```

main()
{
    int matriz_a[2][3] = {{1, -2, 3},
                          {2, 3, -1}};
    int matriz_b[3][2] = {{ 2, 1},
                          { 1, 3},
                          {-3, 2}};
    int matriz_c[2][2];
    int f,c,v,sum;

    for(f = 0; f < 2; f++)
        for(c = 0; c < 2; c++)
        {
            sum = 0;
            for(v = 0; v < 3; v++)
                sum += matriz_a[f][v] * matriz_b[v][c];
            matriz_c[f][c] = sum;
        }
    printf("La matriz producto es:\n");
    for(f = 0; f < 2; f++)
    {
        for(c = 0; c < 2; c++)
            printf("%5d",matriz_c[f][c]);
        printf("\n");
    }
}

```


$$\begin{array}{c}
 \text{MATB} \\
 \begin{bmatrix} 2 & 1 \\ 1 & 3 \\ -3 & 2 \end{bmatrix} \\
 \\
 \begin{array}{cc}
 \text{MATA} & \text{MATC} \\
 \begin{bmatrix} 1 & -2 & 3 \\ 2 & 3 & -1 \end{bmatrix} & \begin{bmatrix} -9 & 1 \\ 10 & 9 \end{bmatrix}
 \end{array}
 \end{array}$$

Figura 7.15. Producto de dos matrices.

Conclusión

En esta sección se ha presentado el concepto de matrices, centrándose más específicamente en el caso de matrices de dos dimensiones. Se ha mostrado cómo dar valor inicial a este tipo de estructuras de datos y cómo pasar valores de este tipo entre funciones. Por último, se han presentado aplicaciones que realizaban operaciones aritméticas sobre matrices. Utilice el siguiente repaso para comprobar su comprensión de los conceptos explicados en esta sección.

Repaso de la Sección 7.4

1. Explique cómo se puede dar valor inicial a una matriz.
2. ¿Cómo se declara una matriz en C?
3. ¿De qué forma se declara un vector de más de una dimensión como parámetro formal de una función?
4. Explique el método utilizado en esta sección para sumar valores almacenados en una matriz?
5. ¿Qué se debe hacer antes de usar el operador += con un vector al que no se le ha asignado un valor inicial?
6. ¿Qué dimensiones tendrán las matrices resultantes de las siguientes operaciones?
 - a) [1][2] * [2][3]
 - b) [3][3] * [3][1]
 - c) [2][1] * [1][4]
 - d) [3][1] * [2][4]

7.5. PROGRAMAS DE APLICACIÓN

Los programas de ejemplo que se presentan en esta sección muestran cómo realizar muchas cosas útiles e interesantes con vectores y matrices. Se recomienda al lector que estudie dichos ejemplos para lograr comprender perfectamente cómo funcionan.

Organización encadenada de archivos

La organización encadenada es una técnica usada en el Sistema Operativo DOS para el almacenamiento y recuperación eficiente de archivos. Se basa en el uso de una estructura de datos denominada tabla de asignación de archivos (*file allocation table*, FAT) que está almacenada en una zona especial del disco. La FAT es una lista de números que describe cómo están asignados los sectores (o unidades de asignación) del disco. Cada posición de la FAT se interpreta como una de las siguientes posibilidades:

- Sector libre
- Sector asignado
- Sector defectuoso
- Sector final del archivo

Uno de los principales beneficios del uso de la FAT es que permite que los archivos no tengan que estar necesariamente almacenados en sectores consecutivos del disco. Por ejemplo, supóngase que hay como mucho seis sectores libres consecutivos en el disco. Si el sistema operativo tuviera que almacenar de forma consecutiva los archivos, no podría almacenar en el disco un archivo que ocupase diez sectores. La organización encadenada sí lo permite, ya que los sectores del archivo pueden residir en cualquier parte del disco.

En la organización encadenada cada posición de la FAT contiene un número que indica una de las siguientes opciones:

- Éste es el último sector del archivo.
- El próximo sector del archivo es el X.

De esta forma, para enlazar una serie de sectores, se necesita solamente colocar el número del siguiente sector en la posición de la FAT correspondiente al sector previo. Para ilustrar esta técnica, se presenta como ejemplo la siguiente FAT para 16 sectores:

0	0	5	4	2	13	12	9
0	99	0	0	99	6	0	0

Cada posición de la FAT con un valor distinto de cero indica que el sector está asignado. El valor 99 especifica que se trata del último sector del archivo. Los elementos con un valor igual a cero se corresponden con sectores libres.

Suponiendo que en este ejemplo el directorio contiene la siguiente información:

<i>Archivo</i>	<i>Sector inicial</i>
ABC	3
DEF	7

¿qué sectores estarán asignados a cada archivo? El archivo ABC comienza en el sector 3. La posición de la FAT correspondiente a este sector contiene un 4, lo que indica que el siguiente sector del fichero es el número 4. Seguidamente la posición 4

referencia al sector 2 que será el tercer sector del archivo. Repitiendo este proceso se obtiene la cadena de sectores del archivo ABC:

3 4 2 5 13 6 12 99

Aplicando el mismo método se obtienen los sectores del archivo DEF:

7 9 99

El Programa 7.19 usa esta técnica para acceder a una FAT capaz de almacenar información sobre 64 sectores. Se usa una función denominada `enlaces()` para buscar a través de la FAT hasta que se encuentra el valor 99. Una segunda función, denominada `sectores_libres()`, se utiliza para determinar cuántos sectores están disponibles.

Programa 7.19 `#include <stdio.h>`

```
void enlaces(int cp);
void sectores_libres(void);

int fat[64] = {0, 0, 99, 54, 5, 6, 38, 24,
               0, 10, 49, 15, 59, 0, 12, 0,
               0, 99, 0, 0, 2, 0, 44, 0,
               29, 0, 11, 0, 0, 3, 0, 0,
               4, 17, 0, 99, 0, 0, 39, 9,
               0, 0, 6, 0, 45, 46, 26, 0,
               0, 33, 0, 0, 0, 0, 55, 56,
               57, 99, 0, 0, 61, 20, 0, 0};

main()
{
    int cadenas[] = {7, 32, 60, 14};
    int i;

    for(i = 0; i < 4; i++)
    {
        printf("Cadena enlazada del archivo #%d: ", i+1);
        enlaces(cadenas[i]);
    }
    printf("\n");
    sectores_libres();
}

void enlaces(int cp)
{
    do
    {
        printf("%d\t", cp);
        if((cp != 0) && (cp != 99))
            cp = fat[cp];
    } while ((cp != 0) && (cp != 99));
}
```

```

        if(cp == 0)
            printf("?\\nError! cadena perdida!");
        printf("\\n");
    }

void sectores_libres(void)
{
    int j,libres = 0;

    for(j = 1; j < 64; j++)
        if(fat[j] == 0)
            libres++;
    printf("Hay %d unidades libres de asignacion.", libres);
}

```

La ejecución del programa produce:

```

Cadena enlazada del archivo #1: 7      24      29      3      54      55
                               56      57
Cadena enlazada del archivo #2: 32      4      5      6      38      39
                               9      10      49      33      17
Cadena enlazada del archivo #3: 60      61      20      2
Cadena enlazada del archivo #4: 14      12      59      ?
Error! cadena perdida!

```

Hay 30 unidades libres de asignacion.

Nótese que hay un problema con el archivo 4 ya que no tiene asignado su cuarto sector. Éste es uno de los problemas inherentes a la organización encadenada: una situación anómala puede causar la rotura de la cadena de sectores del archivo. En el caso del archivo 4, la posición correspondiente al sector 59 contiene un 0 lo que implica que la cadena está rota. Hay otra situación similar en el sector 44.

La FAT definida en el programa contiene problemas adicionales. Por ejemplo, la posición correspondiente al sector 12 referencia al sector 59 pero, sin embargo, no existe ninguna referencia en la FAT al sector 12. A este tipo de problemas se le denomina la *unidad de asignación perdida* y causa una reducción del espacio utilizable del disco. Se deja como ejercicio la búsqueda de más situaciones de este tipo en la FAT definida en el programa, así como el diseño de una posible estrategia para identificar este problema y recuperar el espacio perdido.

Un generador de histogramas

Cuando se examinan grandes volúmenes de datos, una técnica para evaluar el conjunto de los datos es el histograma, que contiene la frecuencia de aparición de cada valor en el conjunto de datos. Por ejemplo, dada la siguiente lista de valores:

3 7 4 2 3 6 7 8 2 3 9

La lista contiene 2 doses, 3 treses, 1 cuatro, 1 seis, 2 setes, 1 ocho y 1 nueve. Por lo tanto, el histograma sería el siguiente:

```
0: 0
1: 0
2: 2
3: 3
4: 1
5: 0
6: 1
7: 2
8: 1
9: 1
```

El propósito del Programa 7.20 es generar un histograma de una lista de 100 valores, donde cada valor está comprendido en el intervalo del 1 al 10.

Programa 7.20

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

main()
{
    int valores[100];
    static int hist[10];
    int i;

    for(i = 0; i < 100; i++)
        valores[i] = 1 + rand() % 10;
    printf("El conjunto inicial de numeros es:\n");
    for(i = 0; i < 100; i++)
    {
        if(0 == i % 10)
            printf("\n");
        printf("%4d",valores[i]);
    }
    for(i = 0; i < 100; i++)
        hist[valores[i] - 1]++;
    printf("\n\n El histograma para los numeros dados es:\n\n");
    for(i = 0; i < 10; i++)
        printf("%4d:\t%d\n",i + 1,hist[i]);
}
```

Los valores se generan mediante la función `rand()` declarada en `<stdlib.h>`. Esta función devuelve un entero cuyo valor está comprendido entre 0 y `RAND_MAX` (un valor predefinido en `<stdlib.h>`). Para transformarlo en el intervalo requerido por el programa, se usa la siguiente sentencia:

```
valores[i] = 1 + rand() % 10;
```

El operador % calcula el resto entero del valor devuelto por `rand()` dividido por 10. Esto genera un intervalo del 0 al 9, con lo que al sumarle uno obtenemos el intervalo requerido.

La ejecución del programa produce:

El conjunto inicial de numeros es:

9	9	4	6	2	8	1	10	3	7
10	8	5	1	6	4	10	4	8	7
7	9	6	2	8	5	2	6	3	7
5	6	5	3	2	8	5	4	2	7
9	9	6	4	8	7	6	4	3	5
4	1	5	4	3	9	3	8	8	6
10	4	2	5	1	10	9	8	10	7
7	1	9	7	5	3	7	6	2	5
4	3	4	7	1	10	2	2	7	2
1	6	10	5	6	4	5	10	9	2

El histograma para el conjunto de numeros dados es:

```

1: 7
2: 11
3: 8
4: 12
5: 12
6: 11
7: 12
8: 9
9: 9
10: 9

```

Compruebe usted mismo que el histograma es correcto.

Tablas de dispersión

El término «tabla de dispersión» (*hash table*) es un nombre algo peculiar utilizado para denominar a una estructura de datos que almacena la información de forma que la posición en la tabla donde se guarda un determinado dato, se calcula mediante el uso de una *función de dispersión*. Una tabla de dispersión puede construirse sobre un vector de una dimensión o sobre una estructura de datos más compleja. En este ejemplo se utilizará una función de dispersión sencilla que genere un índice para una tabla de dispersión que contenga 32 posiciones.

¿Por qué se necesitan tablas de dispersión? Considere el proceso de compilar un programa C. Cada nueva variable o función que encuentra el compilador cuando procesa el programa la almacena en una estructura de datos denominada *tabla de símbolos*. Usando esta tabla, el compilador puede determinar si no se ha definido una variable o si está redefinida.

Cuando se declaran muchas variables y funciones en un programa, el proceso de búsqueda en la tabla puede consumir un tiempo considerable.

El propósito de la tabla de dispersión y de la función de dispersión asociada es intentar minimizar el tiempo de búsqueda en la tabla. El nombre de la variable o función se usa para generar, mediante la función de dispersión, un valor que indica la posición dentro de la tabla de dispersión donde se almacena la información de dicha variable o función. Por ejemplo, dadas dos variables denominadas ABC y DEF, se pueden sumar los valores ASCII de cada letra de su nombre y dividirlos por tres obteniendo el siguiente resultado:

ABC = 66

DEF = 69

Como se observa, se generan dos valores diferentes que pueden usarse como índices dentro de la tabla de dispersión.

El Programa 7.21 muestra la implementación y uso de una función de dispersión sencilla. La tabla de dispersión es un vector de 32 elementos de tipo entero donde cada elemento tiene un valor inicial igual a cero. Este valor indicará que la posición de la tabla esta vacía. El usuario introduce un nombre de variable que se le pasa como parámetro a la función `dispersion()` que aplica la función de dispersión a ese identificador. En primer lugar, realiza la suma de los códigos ASCII correspondientes al primer carácter del nombre, al carácter central y al último carácter. El resultado de la suma se desplaza a la derecha dos bits y, por último, se realiza un AND del valor obtenido con 31, lo que produce un valor resultante incluido en el intervalo del 0 al 31.

Programa 7.21

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

int dispersion(char nombre[]);

main()
{
    static int tabla_dispersión[32], fin;
    int direccion;
    char var[10];
    do
    {
        printf("Introduzca el nombre de una variable => ");
        gets(var);
        if(0 != strcmp(var, "fin"))
        {
            direccion = dispersion(var);
            if(0 == tabla_dispersión[direccion])
            {
                printf("Variable nueva, direccion en la tabla =");
                printf(" %d\n", direccion);
            }
            else
            {

```

```

        printf("Conflicto con %d otras variables!\n",
               tabla_dispersión[dirección]);
        printf("Dirección de la tabla = %d\n", dirección);
    }
    tabla_dispersión[dirección]++;
}
else
    fin++;
} while(!fin);
}

int dispersión(char nombre[])
{
    int val, mitad;

    mitad = strlen(nombre) / 2;
    val = nombre[0] + nombre[mitad] + nombre[strlen(nombre) - 1];
    val >>= 2;
    val &= 31;
    return(val);
}

```

A continuación se muestra la salida correspondiente a una ejecución de este programa:

```

Introduzca el nombre de una variable => indice
Variable nueva, dirección en la tabla = 13
Introduzca el nombre de una variable => var1
Variable nueva, dirección en la tabla = 6
Introduzca el nombre de una variable => var7
Variable nueva, dirección en la tabla = 7
Introduzca el nombre de una variable => fila
Variable nueva, dirección en la tabla = 12
Introduzca el nombre de una variable => num
Variable nueva, dirección en la tabla = 20
Introduzca el nombre de una variable => var2
Conflicto con 1 otras variables!
Dirección de la tabla = 6
Introduzca el nombre de una variable => cuenta
Conflicto con 1 otras variables!
Dirección de la tabla = 12
Introduzca el nombre de una variable => sum
Variable nueva, dirección en la tabla = 21
Introduzca el nombre de una variable => valor
Conflicto con 1 otras variables!
Dirección de la tabla = 21
Introduzca el nombre de una variable => col
Variable nueva, dirección en la tabla = 15
Introduzca el nombre de una variable => fin

```


Obsérvese que al aplicar la función de dispersión a algunas variables se obtiene el mismo valor. A esta situación se le denomina *colisión*. Una buena función de dispersión intenta minimizar el número de colisiones, ya que sólo se puede almacenar un valor en cada posición de la tabla. Puesto que no se pueden eliminar totalmente las colisiones (ni siquiera predecirse), generalmente no se usan vectores para construir tablas de dispersión, sino estructuras bidimensionales (matrices o algún otro tipo de estructuras) que permiten almacenar más de una variable en cada posición de la tabla. Así, cuando se produce una colisión, se compara con todos los valores almacenados en esa misma posición para determinar si el valor ya estaba almacenado o no, en cuyo caso se añadirá el nuevo valor a esa posición de la tabla.

Cuadrados mágicos

Un *cuadrado mágico* es una matriz cuadrada con un número impar de filas y columnas, cuyas filas y columnas (e incluso sus diagonales) suman el mismo valor. Por ejemplo, la matriz siguiente es un cuadrado mágico de 3 por 3:

6	1	8
7	5	3
2	9	4

Los números en cada fila, cada columna y cada diagonal suman 15.

La técnica que se utiliza para generar los cuadrados mágicos es muy simple. Se comienza fijando un valor de 1 en el elemento central de la primera fila. A continuación, se van escribiendo los sucesivos valores (2, 3, etc.) desplazándose desde la posición anterior una fila hacia arriba y una columna hacia la izquierda. Estos desplazamientos se realizan tratando la matriz como si estuviera envuelta sobre sí misma, de forma que moverse una posición hacia arriba desde la fila superior lleva a la inferior y moverse una posición a la izquierda desde la primera columna conduce a la última. Si la nueva posición está ya ocupada, en lugar de desplazarse una fila hacia arriba, se moverá una fila hacia abajo y continuará el proceso.

El Programa 7.22 muestra cómo se implementa esta técnica. Puesto que no se permiten matrices con un tamaño par, el programa comprueba el valor introducido por el usuario para asegurar que se trata de un número impar.

Programa 7.22 `#include <stdio.h>`

```
main()
{
    static int cuadrado_magico[9][9];
    int fila, columna, k, n, x, y;

    printf("Introduzca el tamaño del cuadrado magico => ");
    scanf("%d", &n);
    if(0 == n % 2)
    {
```

```

        printf("Lo siento, debe introducir un numero impar.");
        exit(0);
    }
    k = 2;
    fila = 0;
    columna = (n - 1)/2;
    cuadrado_magico[fila][columna] = 1;
    while(k <= n*n)
    {
        x = (fila - 1 < 0) ? n - 1 : fila - 1;
        y = (columna - 1 < 0) ? n - 1 : columna - 1;
        if(cuadrado_magico[x][y] != 0)
        {
            x = (fila + 1 < n) ? fila + 1 : 0;
            y = columna;
        }
        cuadrado_magico[x][y] = k;
        fila = x;
        columna = y;
        k++;
    }
    for(fila = 0; fila < n; fila++)
    {
        for(columna = 0; columna < n; columna++)
            printf("\t%d", cuadrado_magico[fila][columna]);
        printf("\n");
    }
}

```

Seguidamente se muestran algunos ejemplos de ejecución de este programa para diferentes valores de entrada.

Introduzca el tamaño del cuadrado magico => 3

6	1	8
7	5	3
2	9	4

Introduzca el tamaño del cuadrado magico => 4

Lo siento, debe introducir un numero impar.

Introduzca el tamaño del cuadrado magico => 5

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

¿Podría usted determinar una fórmula que prediga cuál será la suma de cada fila y columna dado un tamaño N ?

Ejercicios interactivos

DIRECTRICES

Ejecute cada programa en su computador y observe los resultados de su ejecución. En muchos casos se pide que prediga dichos resultados. Compare sus predicciones con el resultado real de la ejecución.

Ejercicios

1. Examine los índices del vector del Programa 7.23 ¿Cree usted que el programa compilará y ejecutará? ¿Qué sucede cuando lo intenta?

Programa 7.23

```
#include <stdio.h>
#define MAX 1000

main()
{
    int vector[MAX];
    int indice;

    for (indice = 0; indice > -3; indice--)
        printf("%d\n", vector[indice]);
}
```

2. ¿Qué número muestra el Programa 7.24?

Programa 7.24

```
#include <stdio.h>

int v1[4] = {3, 40, 100};

int mbt(int indice);

main()
{
    printf("%d", mbt(2));
}

int mbt(int indice)
{
    if(0 == indice)
        return(v1[indice]);
    else
        return(v1[indice] + mbt(indice - 1));
}
```

Autoevaluación

DIRECTRICES

Responda a las siguientes preguntas referidas a los programas de la Sección 7.3 (Ordenación de vectores numéricos).

Preguntas

1. ¿Por qué se realizan más comparaciones si se elimina del Programa 7.10 el uso de la variable `seguir`?
2. ¿Por qué se necesita en el Programa 7.10 la variable `temp` para realizar el intercambio de valores entre dos posiciones del vector?
3. ¿Cómo determina el método de la cubeta implementado en el Programa 7.13 qué números debe imprimir?
4. ¿Cómo podría usarse el método de la cubeta para ordenar números positivos y negativos?
5. ¿Por qué la ordenación por fusión requiere más espacio de almacenamiento total para ordenar un vector de 20 elementos que el método de la burbuja?
6. ¿Cómo se selecciona el punto central de un vector en la ordenación por fusión?
7. ¿Para qué sirve el pivote en el método de ordenación rápida?
8. ¿Cómo se puede calcular la mediana de una serie de números almacenados en un vector?
9. ¿Por qué se define la función de comparación usada por `qsort()` de la manera que se hace?
10. Dado el siguiente vector:

10 9 8 7 6 5 4 3 2 1

¿qué método de ordenación sería más eficiente: la ordenación por difusión o la ordenación rápida?

Problemas de fin de capítulo

Conceptos generales

Sección 7.1

1. ¿Qué significa el término «vector numérico»?
2. Explique cómo se declara un vector numérico de 10 elementos en C.
3. ¿Cuál es el índice del primer elemento de un vector de 10 elementos? ¿Cuál es el del último?
4. ¿Cuál es la diferencia entre un vector local y uno global?
5. ¿Qué ocurre cuando se declara un vector estático?

Sección 7.2

6. Explique la idea básica que subyace en las aplicaciones de vectores.
7. Explique qué debería hacer un programa para calcular el valor máximo de un vector.
8. ¿Cómo se puede de forma sencilla almacenar en un vector los datos introducidos por un usuario?

Sección 7.3

9. Explique las diferencias básicas entre los diferentes métodos de ordenación presentados en el capítulo.
10. ¿Cómo afecta el orden inicial de los elementos en la eficiencia de la ordenación?

11. Explique cómo se mezclan dos vectores ordenados.
12. ¿Cómo ayuda la recursividad al proceso de ordenación?
13. ¿Por qué la selección del pivote es tan importante en la ordenación rápida?

Sección 7.4

14. ¿Qué significa el término «almacenamiento por filas» aplicado a cómo se almacena una matriz en memoria?
15. ¿Cómo se pasa una matriz entre funciones?
16. ¿Cuántas operaciones matemáticas se requieren para multiplicar una matriz de 3 por 4 por una de 4 por 2?

Sección 7.5

17. ¿En qué consiste la técnica de organización encadenada de ficheros?
18. ¿Cómo se accede al vector `hist` del Programa 7.20?
19. ¿Para qué sirve una función de dispersión?

Diseño de programas

Los siguientes programas requerirán el uso de vectores y matrices. Para probar cada programa, se deberán construir los juegos de prueba correspondientes y fijar adecuadamente los valores iniciales de los vectores

20. El circuito de la Figura 7.16 es un circuito paralelo. Desarrolle un programa que calcule la resistencia total de la rama que seleccione el usuario. La resistencia total de una rama es:

$$R_T = R_1 + R_2 + R_3$$

Donde

R_T = Resistencia total de la rama (en ohmios)

R_1, R_2, R_3 = Valor de cada resistencia (en ohmios)

21. Modifique el Programa 20 para que calcule la resistencia total de las ramas que seleccione el usuario. La resistencia total se determina calculando la resistencia de las ramas implicadas y, a continuación, aplicando la fórmula de la resistencia de un circuito paralelo:

$$R_T = 1 / (1/R_{T1} + 1/R_{T2} + \dots + 1/R_{TN})$$

Donde

R_T = Resistencia total (en ohmios)

$R_{T1}, R_{T2}, \dots, R_{TN}$ = Valor de cada resistencia (en ohmios)

22. Expanda el Programa 21 para que el usuario pueda introducir el valor de las resistencias y para que las resistencias se muestren en orden numérico.
23. Escriba un programa que normalice los 20 números reales que están almacenados en un vector denominado ESTADISTICAS. Para llevar a cabo esta normalización, se debe en primer lugar calcular el número mayor y luego dividir cada número por dicho valor máximo, de forma que los valores resultantes estén comprendidos en el intervalo del 0 al 1.
24. Desarrolle un programa que genere y muestre una matriz de 10 por 10 de números entre 50 y 200.
25. La Figura 7.17 muestra los movimientos válidos del caballo del ajedrez. Escriba una función que, dadas la posición anterior (FilaAnt, ColAnt) y la nueva (FilaNue, ColNue), valide el movimiento devolviendo un 1 si es correcto y un 0 en caso contrario.

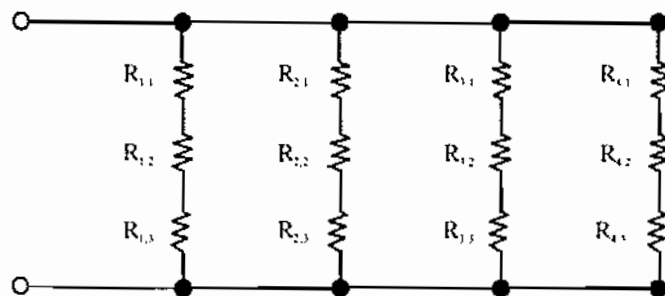


Figura 7.16. Circuito para los Problemas 20 y 21.

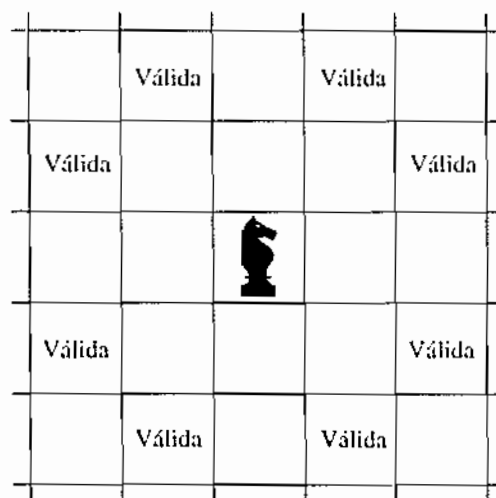


Figura 7.17. Diagrama para el Problema 25.

26. Escriba un programa que calcule todos los factores de un número entero introducido por el usuario. Almacene dichos factores en un vector.
27. Modifique el Programa 26 para determinar si el número introducido es *perfecto*. Para ello, la suma de los factores del número, excluyendo al propio número, debe ser igual a dicho número. Por ejemplo, el 6 es un número perfecto ($6=1+2+3$).
28. Otro método de ordenación es la ordenación por *inserción*. Esta técnica se basa en insertar cada número en su posición correcta desplazando el resto de números afectados. Por ejemplo, dado el siguiente vector:

5 6 9 12 24 39 52

el nuevo valor 18 se insertará entre el 12 y el 24, desplazándose una posición los elementos 24, 39 y 52. Escriba un programa que implemente este método usando como ejemplo el vector anterior. Se debe permitir al usuario introducir tres números mostrando el vector resultante después de cada inserción.

29. Escriba una función que calcule el valor mínimo y máximo de un vector de K elementos dando sólo una pasada al vector.
30. Escriba una función que realice la transposición de una matriz cuadrada. Por ejemplo, dada la matriz siguiente:

1	2	3
4	5	6
7	8	9

la función debería generar la siguiente matriz transpuesta:

1	4	7
2	5	8
3	6	9

Estructuras de datos

Objetivos

Este capítulo le da la oportunidad de aprender lo siguiente:

1. El significado de los tipos enumerados en C.
2. Cómo definir sus propios tipos de datos.
3. El significado de las estructuras de datos.
4. Por qué son útiles las estructuras de datos.
5. Construcción de estructuras de datos con vectores y matrices.
6. Cómo desarrollar estructuras de datos complejas.
7. Cómo utilizar la `union` de C.
8. Cómo utilizar estructuras de datos avanzadas.

Palabras clave

Enumerado	Lista enlazada
Definición de tipo	Pila
Miembro de estructura	LIFO
Estructura	Cola
Operador de miembro	FIFO
Plantilla	Árbol binario
Etiqueta	Recorrido
Lista de declaración de miembros	Emulador
<code>union</code>	Recorrido en profundidad
Vectores y matrices de estructuras	Recorrido en anchura
Nodo	

Contenido

8.1. Tipos enumerados	8.5. La union y vectores de estructuras
8.2. Dar nombre a sus propios tipos de datos	8.6. Formas de representar estructuras
8.3. Introducción a las estructuras de datos	8.7. Estructuras de datos avanzadas
8.4. Más detalles acerca de las estructuras de datos	8.8. Programa de aplicación: MiniMicro
	8.9. Programas de aplicación adicionales
-

Introducción

Hasta el momento, su preocupación se ha centrado principalmente en trabajar con un único tipo de datos cada vez. Por ejemplo, cuando usted trabaja con vectores y matrices, éstos forman un único tipo (tales como `int` o `char`). Un único tipo de datos tiene una estructura de datos simple. En aplicaciones complejas, puede ser necesario el empleo de más de un tipo de datos a la vez. Este capítulo le muestra cómo hacer esto.

Este es un capítulo muy útil. Esencialmente muestra como desarrollar programas que manejen problemas tecnológicos más complejos.

8.1. TIPOS ENUMERADOS

Presentación

Esta sección presenta otra forma de expresar datos en C. Hasta este momento usted ha tenido que emplear la directiva `#define`. Ahora tendrá la oportunidad de aprender otro método para expresar datos que permitirá que sus programas sean más legibles. El material que aprenderá aquí le ayudará a comprender el resto de las secciones de este capítulo.

Expresión de datos

Hay otra forma de expresar datos en C. Esto se realiza mediante `enum` (de **enumera-do**). Los miembros de una enumeración son constantes escritas como identificadores que tienen asignados valores enteros numéricos.

La forma de `enum` es la siguiente:

```
enum etiqueta { lista enumerada }
```

La declaración anterior presenta el nombre de una variable que identifica la enumeración y a continuación define los nombres dentro de la lista enumerada. La declaración comienza con la palabra reservada `enum`. La variable `enum` resultante puede

utilizarse en cualquier parte del programa como si se tratase de un tipo `int`. El Programa 8.1 muestra un ejemplo.

Programa 8.1

```
#include <stdio.h>

enum codigo_color {negro, marron, rojo, naranja, amarillo};

main()
{
    enum codigo_color color;
    char valor;

    printf("Introduzca un entero entre 0 y 4 => ");
    valor = getchar();

    switch(valor)
    {
        case '0' : color = negro;
                  break;
        case '1' : color = marron;
                  break;
        case '2' : color = rojo;
                  break;
        case '3' : color = naranja;
                  break;
        case '4' : color = amarillo;
                  break;
    }

    switch(color)
    {
        case negro      : printf("El color es negro.");
                          break;
        case marron     : printf("El color es marron.");
                          break;
        case rojo        : printf("El color es rojo.");
                          break;
        case naranja     : printf("El color es naranja.");
                          break;
        case amarillo    : printf("El color es amarillo.");
                          break;
    }
}
```

El Programa 8.1 pide al usuario que introduzca un entero entre 0 y 4. El programa da a continuación al usuario el código de color de la resistencia equivalente para ese número. Por ejemplo:

```
Introduzca un entero entre 0 y 4 => 3
El color es naranja.
```

La clave importante en este programa es que se utiliza un tipo de datos enumerado (enum) para hacer el código más legible.

Análisis del programa

En primer lugar, se declara un tipo de datos enumerado fuera de la función `main()`:

```
enum codigo_color {negro, marron, rojo, naranja, amarillo};
```

Esto declara un tipo de datos enumerado denominado `codigo_color`. La lista dentro de las llaves muestra los nombres que son valores válidos de una variable de tipo `enum`, `color_codigo`.

A continuación, dentro de la función `main()`, se declara una variable de tipo enumerado `color_codigo`:

```
main()
{
    enum codigo_color color;
```

Esto significa que la variable `color` puede tomar cualquiera de los siguientes valores: negro, marrón, rojo, naranja o amarillo.

Observe lo que hace el primer `switch` con el valor introducido por el usuario

```
switch(valor)
{
    case '0' : color = negro;
              break;
    case '1' : color = marron;
              break;
    case '2' : color = rojo;
              break;
    case '3' : color = naranja;
              break;
    case '4' : color = amarillo;
              break;
}
```

éste asigna el valor del color a la variable `color`. Por lo tanto, si el usuario elige 3 como entrada, a la variable `color` se le asignará el valor `naranja`. Este dato se utiliza luego en el siguiente `switch`:

```
switch(color)
{
    case negro      : printf("El color es negro.");
                      break;
    case marron     : printf("El color es marron.");
                      break;
    case rojo       : printf("El color es rojo.");
                      break;
    case naranja    : printf("El color es naranja.");
```

```

        break;
    case amarillo : printf("El color es amarillo.");
        break;
}

```

Éste da lugar a un case de C muy descriptivo. Se debería observar que `enum` no introduce un nuevo tipo de datos básico. Las variables de tipo `enum` son tratadas como si fuesen de tipo `int`. Lo que `enum` hace es mejorar la legibilidad de sus programas.

Otra forma

Una declaración `enum` se puede presentar también de forma vertical:

```

enum codigo_color
{
    negro,
    marron,
    rojo,
    naranja,
    amarillo
}

```

Ejemplo de enumeración

Para demostrar exactamente lo que ocurre con un tipo enumerado (`enum`), observe el Programa 8.2.

Programa 8.2

```

#include <stdio.h>

enum codigo_color {negro, marron, rojo, naranja, amarillo}

main()
{
    enum codigo_color color;

    for(color = negro; color <= amarillo; color++)
        printf("Valor numerico del tipo enumerado color=> %d\n",
            color);
}

```

La ejecución del Programa 8.2 produce el siguiente resultado:

```

Valor numerico del tipo enumerado color=> 0
Valor numerico del tipo enumerado color=> 1
Valor numerico del tipo enumerado color=> 2
Valor numerico del tipo enumerado color=> 3
Valor numerico del tipo enumerado color=> 4

```

Observe que el tipo `enum` se utiliza en el bucle `for` como si se tratase de un tipo `int`.

Asignación de valores a tipo enumerados

Se pueden asignar valores a un tipo `enum` de C – como si se tratase de valores enteros. Esto se demuestra en el Programa 8.3. Este programa calcula la intensidad total para una fuente de alimentación cuando se conocen el voltaje equivalente de Thevenin* y la resistencia. En este programa, el voltaje equivalente de Thevenin* es de 12 voltios y la resistencia equivalente de Thevenin es de 150 ohmios. Nótese que estos valores son asignados en el tipo `enum` de C.

Programa 8.3

```
#include <stdio.h>

enum thevenin {fuente = 12, resistencia = 150};

main()
{
    float carga;
    float intensidad;

    printf("Introduzca el valor de la carga en la resistencia => ");
    scanf("%f",&carga);

    intensidad = fuente/(resistencia + carga);

    printf("La intensidad del circuito es de %f amperios.",
           intensidad);
}
```

En el Programa 8.3, el tipo de datos enumerado `thevenin` asigna los valores de la fuente (igual a 12) y la resistencia (igual a 150). Estos nombres se utilizan dentro del programa para realizar los cálculos necesarios. De nuevo, el objetivo que se persigue es hacer el programa más legible y permitir que los cambios en las constantes utilizadas en el programa se realicen de forma sencilla.

Conclusión

Esta sección ha presentado los tipos de datos enumerados en C (`enum`). Se ha descubierto otro método de utilizar datos dentro de un programa en C con el objetivo de mejorar su legibilidad. También se ha visto que el uso de la enumeración (`enum`) en C

* La forma equivalente de Thevenin de un circuito resistivo está compuesta de una fuente de alimentación equivalente y una resistencia equivalente. Estos valores dependen de los valores del circuito original. Este método se utiliza para simplificar los circuitos resistivos transformándolos en un circuito con una única fuente y una única resistencia.

es compatible con el tipo `int`. Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.1

1. En C, ¿qué significa la palabra reservada `enum`?
2. Indique cómo se construye un tipo de datos enumerado (`enum`) en C.
3. Para el siguiente fragmento de código, indique el valor entero de `primero`.

```
enum numeros {primero, segundo, tercero}
```
4. ¿Cuál es el principal propósito del empleo de `enum` en C?
5. ¿Se puede asignar a una constante de tipo `enum` un valor entero? Dé un ejemplo.

8.2. DAR NOMBRE A SUS PROPIOS TIPOS DE DATOS

Presentación

En la Sección 8.1 se presentó un método que hacía más fácil la lectura de un programa. Esta sección aborda el mismo tema. Sin embargo, al contrario de la sección anterior, esta sección mostrará cómo dar nombre a sus propios tipos de datos.

Dentro de `typedef`

El uso de `typedef` (**definición de tipo**) en C permite definir un nombre para un tipo de datos de C. Una declaración `typedef` es similar a la declaración de una variable con la excepción de que se utiliza la palabra reservada `typedef`. La forma es

```
typedef tipo nuevo-tipo
```

Por ejemplo, se podría tener:

```
typedef char LETRA;
```

Esto declara a `LETRA` como un sinónimo de `char`, lo que significa que `LETRA` puede ser utilizado para declarar una variable, como por ejemplo

```
LETRA caracter;
```

en vez de

```
char caracter;
```

Como ejemplo, supóngase que se tiene un programa que forma parte de un sistema de comprobación automatizado. En este programa el usuario introduce el estado de una luz indicadora y el programa da una respuesta dependiendo del estado dado. Este programa podría escribirse como se muestra en el Programa 8.4.

Programa 8.4 #include <stdio.h>

```
void comprobacion(void);
int comprobar_luces(char condicion);

main()
{
    comprobacion();
}

void comprobacion()
{
    char entrada;

    printf("\n\n1] Rojo 2] Verde 3] Off \n\n");
    printf("Seleccione la condicion luminica por numero => ");
    entrada = getchar();

    comprobar_luces(entrada);
    return;
}

int comprobar_luces(char condicion)
{
    switch(condicion)
    {
        case '1': printf("Comprobar la presion del sistema.\n");
                  break;

        case '2': printf("Sistema correcto.\n");
                  break;

        case '3': printf("Comprobar el fusible del sistema.\n");
                  break;
    }
}
```

La ejecución del Programa 8.4 produce la siguiente salida:

```
1] Rojo 2] Verde 3] Off
Seleccione la condicion luminica por numero => 2
Sistema correcto.
```

Sin embargo, el programa podría hacerse más descriptivo definiendo un tipo de datos llamado `estado_luces`. El Programa 8.5 muestra el programa anterior con la definición de este nuevo tipos de datos.

Programa 8.5

```

#include <stdio.h>

typedef enum estado_luces {Rojo, Verde, Off};
void comprobacion(void);
int comprobar_luces(enum estado_luces condicion);

main()
{
    comprobacion();
}

void comprobacion()
{
    char entrada;
    enum estado_luces lectura;

    printf("\n\n1] Rojo 2] Verde 3] Off \n\n");
    printf("Seleccione la condicion luminica por numero => ");
    entrada = getchar();

    switch (entrada)
    {
        case '1' : lectura = Rojo;
                    break;
        case '2' : lectura = Verde;
                    break;
        case '3' : lectura = Off;
                    break;
    }

    comprobar_luces(lectura);
}

int comprobar_luces(enum estado_luces condicion)
{
    switch(condicion)
    {
        case Rojo : printf("Comprobar la presion del sistema.\n");
                    break;

        case Verde: printf("Sistema correcto.\n");
                    break;

        case Off  : printf("Comprobar el fusible del sistema.\n");
                    break;
    }
}

```

La salida del Programa 8.5 es la misma que la del Programa 8.4. La diferencia entre los dos programas es que se ha creado un nuevo nombre de tipo para un tipo de datos existente. Esto permite que nuevos identificadores en C se puedan utilizar como

tipos de datos. Estos nuevos tipos de datos se pueden emplear en los cuerpos de las funciones así como en los parámetros de las funciones. Es importante notar que la declaración `typedef` no crea un nuevo tipo, simplemente crea sinónimos para tipos ya existentes.

Análisis del programa

La palabra reservada en C `typedef`, se utiliza para indicar que se va a usar una nueva definición para un tipo de datos.

```
typedef enum estado_luces {Rojo, Verde, Off};
```

En este caso, se trata de un tipo de datos enumerado denominado `estado_luces`, que consiste en las constantes enumeradas `Rojo`, `Verde` y `Off`. Ahora se pueden definir otras variables utilizando este nuevo tipo de datos. Véase el parámetro formal del prototipo de la función:

```
int comprobar_luces(enum estado_luces condicion);
```

Éste declara una nueva variable denominada `condición` cuyo tipo es `estado_luces`. Este tipo se utiliza de nuevo en la definición de la función `comprobación()`.

```
void comprobacion()
{
    char entrada;
    enum estado_luces lectura;
```

Esto significa que la variable `lectura` es del tipo de datos enumerado `estado_luces` y puede tomar los valores `Rojo`, `Verde` y `Off`.

Después de asignar a la variable `lectura` uno de los tres posibles valores en la sentencia `switch` de C, ésta se pasa en la llamada a la función `comprobar_luces()`.

```
switch (entrada)
{
    case '1' : lectura = Rojo;
               break;
    case '2' : lectura = Verde;
               break;
    case '3' : lectura = Off;
               break;
}
```

```
comprobar_luces(lectura);
```


A continuación se utiliza, en la función llamada, dentro de una sentencia `switch` para hacer el código más legible:

```
int comprobar_luces(enum estado_luces condicion)
{
    switch(condicion)
    {
        case Rojo : printf("Comprobar la presion del sistema.\n");
                    break;

        case Verde: printf("Sistema correcto.\n");
                    break;

        case Off  : printf("Comprobar el fusible del sistema.\n");
                    break;
    }
}
```

Otras aplicaciones

Otra aplicación de `typedef` en C es la siguiente:

```
typedef char *CADENA;
```

Esto permite utilizar la nueva palabra `CADENA` como un puntero a `char`:

```
CADENA nombre_de_entrada;
```

Aquí, `nombre_de_entrada` se define de tipo `CADENA`, que realmente es un puntero a `char` (`char *`).

En las siguientes secciones se seguirán viendo aplicaciones más potentes del uso de `typedef` en C.

Conclusión

Esta sección ha presentado un método que permite dar nombres a tipos de datos utilizando C. Se ha visto como implementar esto en un programa donde se crean nuevos identificadores en función de estos nuevos tipos de datos. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.2

1. Indique que hace `typedef` en C.
2. Dado el siguiente fragmento de código, ¿cuál es el nombre del nuevo tipo de datos que resulta?

```
typedef char name[20];
```

3. ¿Cuál es el principal propósito de utilizar `typedef` en C?

8.3. INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS

Presentación

Hasta el momento, se han utilizado vectores y punteros para crear elementos del mismo tipo de datos. Nunca se han mezclado un tipo de datos `char` y un tipo `int` dentro de un mismo tipo de datos. Comprensiblemente, usted puede no haber considerado esta posibilidad. Sin embargo, la mayoría de los objetos de uso diario requieren el empleo de más de un tipo de datos. Considere su cuenta corriente. Cada cheque que usted escribe debe tener su nombre (un tipo `char`), la cantidad del cheque (un tipo `float`) y el número de cheque (un tipo `int`). Toda esta información (y mucho más) se encuentra en un elemento denominado cheque. Debido a que ésta es una forma muy natural de organizar la información, C proporciona una forma de llevar a cabo esto, que hace más legible los programas y más fácil el manejo de datos complejos compuestos de tipos de datos diferentes que se encuentran lógicamente relacionados.

La estructura en C

Una declaración de estructura define una variable estructura e indica una secuencia de nombres de variables —denominadas **miembros de la estructura**— que pueden tener tipos diferentes. La forma básica es

```
struct
{
    tipo identificador_1;
    tipo identificador_2;
    .
    .
    .
    tipo identificador_N;
} identificador_de_estructura;
```

Observe que la declaración de la estructura comienza con la palabra reservada `struct`.

Como ejemplo del uso de una estructura en C, considere la caja de piezas que se muestra en la Figura 8.1.

Considere que esta caja de piezas consta de un tipo de resistencias. Asuma por el momento que hay tres aspectos relacionados con las resistencias que se desean almacenar: el nombre del fabricante, la cantidad de resistencias en la caja y el precio de cada resistencia. Usted podría desarrollar un programa en C que hiciera fácilmente esto sin utilizar el concepto de estructura, pero para mantener las cosas sencillas por ahora, se utilizará la **estructura** de C para llevar este inventario. El Programa 8.6 ilustra la construcción de una estructura en C.

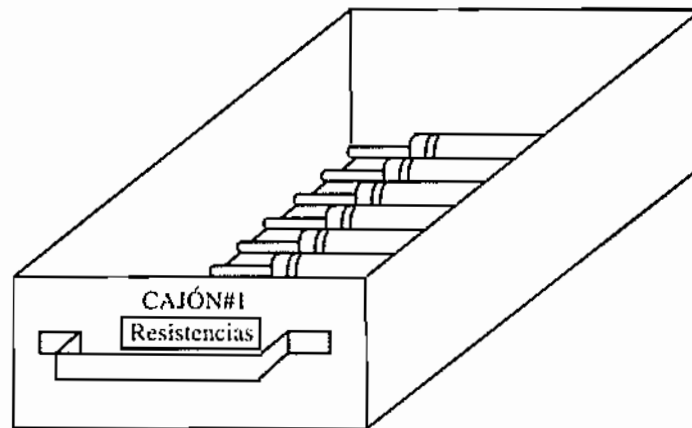


Figura 8.1. Caja de resistencias.

Programa 8.6 `#include <stdio.h>`

```
main()
{
    struct
    {
        char fabricante[20];    /* Fabricante de la resistencia. */
        int cantidad;           /* Número de resistencias. */
        float precio_unitario;  /* Precio de cada resistencia. */
    } resistencias;
}
```

La Figura 8.2 ilustra los puntos clave para la construcción de una estructura C.

Observe que la estructura del Programa 8.6 consta de un conjunto de elementos de datos, denominados miembros, que pueden ser del mismo tipo o de diferentes tipos y que se encuentran lógicamente relacionados. Los tipos de datos son `char` para la variable `fabricante`, `int` para la variable `cantidad` y `float` para la variable `precio_unitario`. La forma general es:

```
struct
{
    tipo identificador_1;
    tipo identificador_2;
    .
    .
    .
    tipo identificador_N;
} identificador_de_estructura;
```

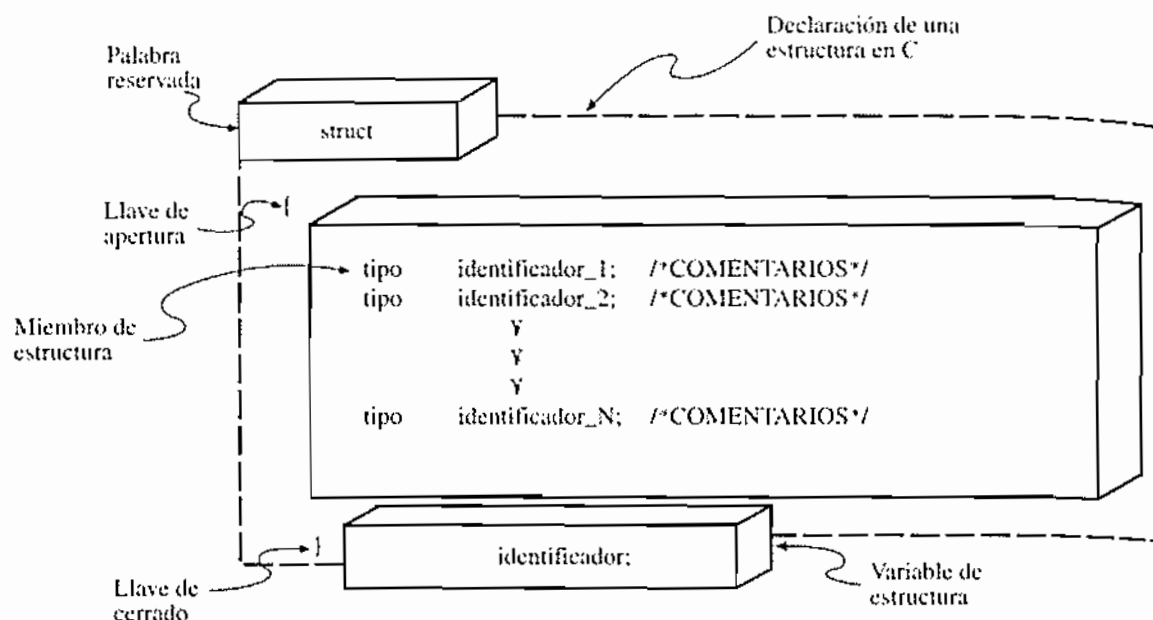


Figura 8.2. Partes clave de una estructura C.

Donde

`struct` = Palabra reservada que indica que a continuación sigue una estructura.
`{` = Apertura de llave necesaria para indicar que a continuación sigue una lista de elementos.
`tipo` = El tipo C de cada elemento de la estructura.
`identificador` = El identificador de la variable utilizada como miembro de la estructura.
`}` = Cierre obligatorio de llave.
`identificador_de_estructura` = Define la variable estructura.

Asignación de datos a una estructura

Ahora que se ha visto cómo es una estructura en C, es necesario saber cómo asignar valores a los miembros de una estructura. El Programa 8.7 muestra como hacer esto. Este programa toma información del usuario relacionada con el fabricante de las resistencias, la cantidad de las resistencias en la caja y el precio unitario de cada una de las resistencias. A continuación calcula el valor total de todas las resistencias e imprime este resultado junto con la información introducida por el usuario.

Para utilizar los datos de un miembro de una estructura se utiliza el **operador de miembro**. El operador de miembro especifica el nombre del miembro de la estructura y la estructura de la cual es miembro. Por ejemplo, en el Programa 8.7

```
resistencias.fabricante
```

representa la variable `fabricante`, que es un miembro de la estructura `resistencias`. Observe que el operador de miembro se representa por medio del punto (`.`).

Programa 8.7

```
#include <stdio.h>

main()
{
    struct
    {
        char fabricante[20]; /* Fabricante de la resistencia.*/
        int cantidad;        /* Número de resistencias. */
        float precio_unitario; /* Precio de cada resistencia. */
    } resistencias; /* Estructura */

    float valor_total; /* Valor total de las piezas. */

    /* Leer el nombre del fabricante: */
    printf("Nombre del fabricante => ");
    gets(resistencias.fabricante);

    /* Leer el número de piezas: */
    printf("Numero de piezas => ");
    scanf("%d",&resistencias.cantidad);

    /* Leer el precio de cada pieza: */
    printf("Precio de cada pieza => ");
    scanf("%f",&resistencias.precio_unitario);

    /* Calcular el valor total: */
    valor_total = resistencias.cantidad * resistencias.precio_unitario;

    printf("\n\n");
    printf("Articulo: Resistencias\n\n");
    printf("Fabricante: %s\n",resistencias.fabricante);
    printf("Precio unitario: %f\n",resistencias.precio_unitario);
    printf("Cantidad: %d\n",resistencias.cantidad);
    printf("Valor total: %f\n", valor_total);
}
```

Asumiendo que el usuario introduce lo siguiente, la ejecución del Programa 8.7 daría el siguiente resultado:

```
Nombre del fabricante => Ohmite
Número de piezas => 10
Precio de cada pieza => 0.05
```

```

Articulo:      Resistencias
Fabricante:    Ohmite
Precio unitario: $0.050000
Cantidad:      10.000000
Valor total:   $0.500000

```

Análisis del programa

El Programa 8.7 comienza con la misma estructura definida anteriormente y declara también otra variable, `valor_total`.

```

main()
{
    struct
    {
        char fabricante[20];    /* Fabricante de la resistencia.*/
        int cantidad;           /* Número de resistencias. */
        float precio_unitario;  /* Precio de cada resistencia. */
    } resistencias;            /* Estructura */

    float valor_total;         /* Valor total de las piezas. */

```

La variable `valor_total` se utiliza para almacenar el valor que representa el valor total de todas las resistencias de la caja de piezas.

A continuación el programa lee el nombre del fabricante de las resistencias.

```

/* Leer el nombre del fabricante: */
printf("Nombre del fabricante => ");
gets(resistencias.fabricante);

```

Observe como se hace esto. El nombre del fabricante debe almacenarse en el miembro `char fabricante[20]`. Debido a que la variable `fabricante[20]` se encuentra contenida en la estructura referida por la variable `resistencias`, esto debe reflejarse de algún modo en el programa. Para ello se utiliza el operador de miembro «`.`».

```
resistencias.fabricante
```

El operador de miembro especifica el nombre del miembro de la estructura y la estructura de la cual es miembro. El operador de miembro de C se utiliza de nuevo para leer los dos datos restantes:

```

/* Leer el número de piezas:          */
printf("Numero de piezas => ");
scanf("%d",&resistencias.cantidad);

/* Leer el precio de cada pieza:      */
printf("Precio de cada pieza => ");
scanf("%f",&resistencias.precio_unitario);

```

Nótese que esta vez se requiere el uso del operador de dirección &, debido al uso de la función `scanf()`. Pero el operador de miembro sigue siendo el mismo:

```
nombre_de_estructura.nombre_de_miembro
```

A continuación, se realiza el cálculo utilizando el operador de miembro:

```
/* Calcular el valor total: */
valor_total = resistencias.cantidad * resistencias.precio_unitario;
```

De nuevo, el nombre de la estructura y el nombre del miembro se utilizan para identificar las variables de la estructura. Los resultados se muestran a continuación. Observe que la variable `valor_total` no requiere el operador de miembro debido a que no es miembro de ninguna estructura.

```
printf("\n\n");
printf("Artículo:      Resistencias\n\n");
printf("Fabricante:     %s\n", resistencias.fabricante);
printf("Precio unitario:  $%f\n", resistencias.precio_unitario);
printf("Cantidad:       %d\n", resistencias.cantidad);
printf("Valor total:     $%f\n", valor_total);
```

Conclusión

Esta sección ha presentado el concepto de estructura en C. En la siguiente sección se verán modos más potentes de hacer uso de estructuras en C. Ahora, compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.3

1. Indique porqué la información de un cheque de una cuenta corriente puede ser tratada como una estructura.
2. ¿Qué es una estructura en C?
3. Describa la forma de representar una estructura en C que se ha seguido en esta sección.
4. Indique cómo se puede programar un miembro de una estructura para leer y mostrar datos.
5. Dé un ejemplo del Problema 4 anterior

8.4. MÁS DETALLES ACERCA DE LAS ESTRUCTURAS DE DATOS

Presentación

En la sección anterior, se presentó el concepto y la forma de una estructura en C. En esta sección, se verán diferentes formas de utilizar una estructura en un programa escrito en C. Se verá cómo declarar funciones que devuelven estructuras y cómo pasar estructuras entre funciones.

La etiqueta de estructura

En la sección anterior, se mostró el uso de una estructura para un inventario de resistencias del mismo tipo contenidas en una caja. Obsérvese cuidadosamente el lugar en el que se definía la estructura:

```
main()
{
    struct
    {
        char fabricante[20];    /* Fabricante de la resistencia.*/
        int cantidad;          /* Número de resistencias. */
        float precio_unitario; /* Precio de cada resistencia. */
    } resistencias; /* Estructura */
```

Esta estructura se definió dentro de la función `main()` y como consecuencia de esto sólo será conocida dentro de `main()`. Otra forma de crear una estructura es definirla antes de la función `main()`. De esta forma se crea una variable global.

C tiene una forma de anunciar la construcción de una estructura que servirá como una **plantilla** que puede ser utilizada por cualquier función dentro de un programa. Esto se ilustra en el Programa 8.8. La salida de este programa es exactamente la misma que la del Programa 8.7. La diferencia se encuentra en el lugar donde se sitúa la estructura en el programa.

Programa 8.8 #include <stdio.h>

```
struct registro_piezas
{
    char fabricante[20];    /* Fabricante de la resistencia. */
    int cantidad;          /* Número de resistencias. */
    float precio_unitario; /* Precio de cada resistencia. */
};

main()
{
    struct registro_piezas resistencias; /* Variable de tipo
                                           registro_piezas. */
    float valor_total;                /* Total value of parts. */

    /* Leer el nombre del fabricante: */
    /* Leer el número de piezas: */
    /* Leer el precio unitario: */
    /* Calcular el valor total: */
    /* Imprimir las variables: */
}
```

Análisis del programa

El Programa 8.8 declara una estructura denominada `registro_piezas`. No hay ningún identificador de variable. `registro_piezas` se utiliza como una **etiqueta** de estructura. Esta etiqueta es un identificador que da nombre a la estructura definida en la **lista de declaración de miembros**. Observe que esto se hizo después de la palabra reservada `struct`.

```
struct registro_piezas
{
    char fabricante[20]; /* Fabricante de la resistencia. */
    int cantidad;        /* Número de resistencias.      */
    float precio_unitario; /* Precio de cada resistencia.  */
};
```

Ahora cualquier función dentro del programa puede utilizar esta estructura haciendo referencia a la etiqueta de la estructura. Por ejemplo en `main()`.

```
main()
{
    struct registro_piezas resistencias; /* Variable de tipo
                                          registro_piezas.*/
}
```

Observe el formato de la declaración anterior. Utiliza la palabra reservada `struct` y el identificador de etiqueta `registro_piezas` para definir la variable `resistencia` como una estructura del tipo `registro_piezas`. La forma general de declaración cuando se utiliza una etiqueta de estructura es la siguiente:

```
struct etiqueta identificador_de_variable;
```

Para acceder a los miembros individuales de la estructura se utiliza el mismo método que se comentó anteriormente, el operador de miembro «`.`». Por ejemplo, `resistencias.cantidad` identifica el miembro de la estructura denominado `cantidad`. El uso de etiquetas y la declaración de la estructura fuera de la función `main`, crea una plantilla que puede utilizarse en cualquier función dentro del programa.

Nombrar una estructura

Otra forma de identificar una estructura es mediante el uso de `typedef`. Esto se muestra en el Programa 8.9.

Programa 8.9 `#include <stdio.h>`

```
typedef struct
{
    char fabricante[20]; /* Fabricante de la resistencia. */
    int cantidad;        /* Número de resistencias.      */
}
```

```

    float precio_unitario;    /* Precio de cada resistencia. */
} registro_piezas;           /* Nombre de esta estructura. */

main()
{
    registro_piezas resistencias; /* Variable de tipo
                                   registro_piezas. */
    float valor_total;           /* Valor total de las piezas. */

    /* Leer el nombre del fabricante: */

    /* Leer el número de piezas: */

    /* Leer el precio de cada pieza: */

    /* Calcular el valor total: */

    /* Imprimir las variables: */
}

```

El Programa 8.9 hace exactamente lo mismo que el Programa 8.8. La diferencia ahora se encuentra en el modo de declarar la estructura en C. En esta ocasión, se ha utilizado `typedef` para asignar un nombre al tipo de estructura `registro_piezas`.

```

typedef struct
{
    char fabricante[20]; /* Fabricante de la resistencia. */
    int cantidad;        /* Número de resistencias. */
    float precio_unitario; /* Precio de cada resistencia. */
} registro_piezas;      /* Nombre de esta estructura. */

```

Observe que en este caso no se está utilizando una etiqueta de estructura, sino que la variable `registro_piezas` está siendo definida como un tipo de datos por medio de la palabra reservada `typedef`. La ventaja de esta definición (al igual que antes) es que se puede emplear la estructura definida en cualquier función. Sin embargo, sólo es necesario declarar variables locales de tipo estructura en función de esta nueva definición de tipo. Por ejemplo, esto se hace dentro de la función `main()` para declarar una variable local denominada `resistencias`.

```
registro_piezas resistencias; /* Variable de tipo registro_piezas. */
```

Nótese que la variable `resistencias` es del tipo de datos `registro_piezas`. De nuevo, se puede acceder a cada miembro de la estructura utilizando el operador de miembro; por ejemplo, `resistencias.cantidad`, identifica al miembro `cantidad` dentro de la estructura `resistencias`.

En la Figura 8.3 se aprecian las diversas formas de declarar estructuras.

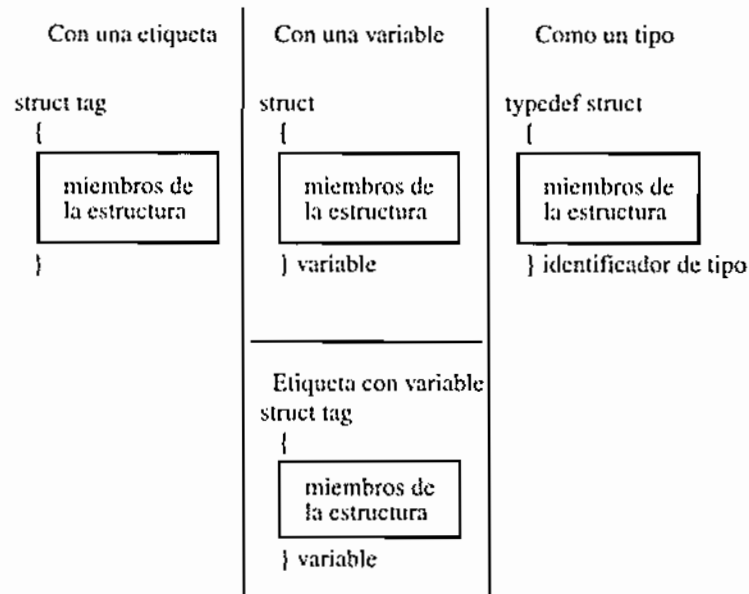


Figura 8.3. Formas de declarar estructuras.

Punteros a estructuras

Usted puede declarar una variable que sea un *puntero* a una estructura. En el Programa 8.10 se ilustra esta característica. Observe que se ha utilizado la palabra reservada de C `typedef` para definir un tipo de estructura. El programa hace exactamente lo mismo que antes. Sin embargo, lo hace utilizando un puntero a una estructura.

Programa 8.10

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

typedef struct
{
    char fabricante[20];      /* Fabricante de la resistencia. */
    int cantidad;            /* Número de resistencias. */
    float precio_unitario;    /* Precio de cada resistencia. */
} registro_piezas;

main()
{
    registro_piezas *ptr_reg; /* Puntero a estructura. */
    float valor_total;        /* Valor total de las piezas. */

    /* Asignar memoria al puntero a la estructura. */
    ptr_reg = (registro_piezas *) malloc(sizeof(registro_piezas));

    /* Leer el nombre del fabricante: */
    printf("Nombre del fabricante => ");
    gets(ptr_reg->fabricante);
```

```

/* Leer el número de piezas: */
printf("Numero de piezas => ");
scanf("%d",&ptr_reg->cantidad);

/* Leer el precio de cada pieza: */
printf("Precio de cada pieza => ");
scanf("%f",&ptr_reg->precio_unitario);

/* Calcular el valor total: */
valor_total = ptr_reg->cantidad * ptr_reg->precio_unitario;

/* Imprimir las variables: */
printf("\n\n");
printf("Articulo: Resistencias\n\n");
printf("Fabricante: %s\n",ptr_reg->fabricante);
printf("Precio unitario: %f\n",ptr_reg->precio_unitario);
printf("Cantidad: %d\n",ptr_reg->cantidad);
printf("Valor total: %f\n", valor_total);

/* liberar la memoria asignada */
free(ptr_reg);
}

```

Nótese que el puntero a la estructura se declara de la misma forma que se declara normalmente un puntero:

```

main()
{
    registro_piezas *ptr_reg; /* Puntero a estructura. */

```

Sin embargo, `ptr_reg` es ahora un puntero a una estructura. Hay que tener en cuenta que un puntero no reserva en memoria espacio para almacenar la estructura, simplemente contiene la dirección de una zona de memoria donde se encuentra la estructura. Debido a esto, es necesario asignar memoria a la variable de tipo puntero. Esto se consigue mediante:

```

/* Asignar memoria al puntero a la estructura. */
ptr_reg = (registro_piezas *) malloc(sizeof(registro_piezas));

```

El archivo de cabecera `<stdlib.h>` contiene la declaración de una función denominada `malloc()` que se emplea para asignar memoria de forma dinámica, es decir, en tiempo de ejecución. Esta función recibe el número de bytes que se necesita asignar, en este caso el tamaño de una estructura de tipo `registro_piezas` (`sizeof(registro_piezas)`). La función devuelve un puntero (una dirección) al bloque de memoria que se ha asignado. La asignación se realiza empleando una conversión de tipos (`registro_piezas *`).

Cuando se emplean punteros a estructuras, el acceso a los miembros de la estructura se realiza empleando el símbolo especial de C `->`:

```

/* Leer el nombre del fabricante: */

```

```
printf("Nombre del fabricante => ");
gets(ptr_reg->fabricante);
```

El símbolo `->` hace referencia al miembro de una estructura apuntada por un puntero. En efecto, éste reemplaza al operador de miembro `.` utilizado en los ejercicios previos.

Debido a que una estructura puede ser definida como un puntero, también se puede utilizar el operador de dirección `&` como con cualquier otra variable. Esto significa que la dirección de una estructura se puede asignar a un puntero de una estructura.

Por último, el programa finaliza liberando previamente la memoria asignada con `malloc()`. Esto se realiza mediante la función de biblioteca `free()`, definida igualmente en el archivo `<stdlib.h>`:

```
/* liberar la memoria asignada */
free(ptr_reg);
```

Funciones que devuelven estructuras

Se puede declarar una función en C de tal manera que ésta devuelva una estructura, es decir, que sea de tipo estructura. Esto se demuestra en el Programa 8.11. De nuevo, el programa hace exactamente lo mismo que los programas utilizados en las secciones anteriores. En este caso, sin embargo, `main()` se encarga de llamar a otras funciones, que son las que realizan el trabajo. Observe en el programa que la función `leer_datos` se declara de tipo `registro_piezas`, lo que significa que la función devuelve una estructura de tipo `registro_piezas`. Observe también, que una variable estructura se utiliza como argumento de una función. Esto permite pasar los datos de una estructura a la función `imprimir_pieza()`.

Programa 8.11 `#include <stdio.h>`

```
typedef struct
{
    char fabricante[20];    /* Fabricante de la resistencia. */
    int cantidad;           /* Número de resistencias.      */
    float precio_unitario;  /* Precio de cada resistencia.  */
} registro_piezas;

/* Leer datos del usuario. */
registro_piezas leer_datos(void);

/* Imprimir la salida. */
void imprimir_pieza(registro_piezas resistencia);

main()
{
    registro_piezas resistencia; /* Estructura de tipo
                                registro_piezas. */
```

```

        resistencia = leer_datos(); /* Leer los datos del usuario. */
        imprimir_pieza(resistencia); /* Imprimir los datos de la pieza. */
    }

registro_piezas leer_datos(void)
{
    registro_piezas reg;

    /* Leer el nombre del fabricante: */
    printf("Nombre del fabricante => ");
    gets(reg.fabricante);

    /* Leer el número de piezas: */
    printf("Numero de piezas => ");
    scanf("%d",&reg.cantidad);

    /* Leer el precio unitario: */
    printf("Precio unitario de cada pieza => ");
    scanf("%f",&reg.precio_unitario);

    return(reg);
}

void imprimir_pieza(registro_piezas reg)
{
    /* Imprimir las variables: */

    printf("\n\n");
    printf("Articulo:          Resistencias\n\n");
    printf("Fabricante:           %s\n",reg.fabricante);
    printf("Precio unitario:      $%f\n",reg.precio_unitario);
    printf("Cantidad:            %d\n",reg.cantidad);
}

```

Análisis del programa

Lo novedoso de este programa aparece en los prototipos de las funciones:

```

/* Leer datos del usuario. */
registro_piezas leer_datos(void);

```

El programa ha definido un tipo de datos denominado `registro_piezas` empleando la palabra reservada de C `typedef`. A continuación la función `leer_datos` se ha definido de tipo `registro_piezas`.

Más tarde se ha utilizado este tipo como parámetro formal de la función:

```

/* Imprimir la salida. */
void imprimir_pieza(registro_piezas resistencia);

```

El parámetro formal `resistencia` es de tipo `registro_piezas` (que es el nombre del tipo de estructura definido anteriormente). Esto significa que el argumento que se pase

a la función debe ser del mismo tipo. Esto es lo que ocurre en el cuerpo de la función `main()`:

```
main()
{
    registro_piezas resistencia; /* Estructura de tipo
                                registro_piezas. */

    resistencia = leer_datos(); /* Leer los datos del usuario. */
    imprimir_pieza(resistencia); /* Imprimir los datos de la pieza. */
}
```

La variable `resistencia` se define de tipo `registro_piezas`. Como `resistencia` tiene el mismo tipo que el devuelto por la función `leer_datos()`, se puede asignar el valor que devuelve esta función a la variable `resistencia`.

En la definición de la función `leer_datos()`, se devuelve la estructura completa `reg`, que es también de tipo `registro_piezas`.

La última función, `imprimir_pieza()`, simplemente toma como parámetro real `resistencia` y la utiliza para imprimir los datos de la estructura.

Conclusión

Esta sección ha presentado varias formas de emplear las estructuras en C. Se han visto diferentes formas de construir una estructura. También se ha visto cómo definir funciones que devuelven estructuras y cómo pasar estructuras entre funciones. Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.4

1. Describa el significado de una etiqueta de estructura.
2. Dé un ejemplo de etiqueta de estructura.
3. Explique si una estructura puede ser un tipo de variable.
4. ¿Cómo se referencia a un miembro de una estructura cuando se utiliza un puntero a una estructura?
5. ¿Cuáles son las tres operaciones que se pueden hacer con una estructura?

8.5. LA unión Y VECTORES DE ESTRUCTURAS

Presentación

Esta sección presenta la unión de C. Como se verá, ésta es muy similar a una estructura de C en muchos aspectos pero con una diferencia importante. También se presentará la forma de combinar juntos las estructuras de C con los vectores y matrices. Esta combinación le dará una tremenda potencia de programación que le permitirá manejar tipos de datos muy diferentes. Esta sección combina muchos conceptos presentados hasta el momento.

La union de C

La union de C es muy similar a la estructura de C. La diferencia es que la union se utiliza para almacenar tipos de datos diferentes en la misma zona de memoria. La union de C es la siguiente:

```
union etiqueta
{
    tipo identificador_de_miembro_1;
    tipo identificador_de_miembro_2;
    .
    .
    .
    tipo identificado_de_miembro_N;
}
```

Como se puede ver, la declaración de una union tiene la misma forma que la declaración de una estructura. La diferencia se encuentra en la palabra reservada union. Una declaración de union da nombre a una variable union e indica el conjunto de miembros de la union. Estos miembros pueden tener diferentes tipos. Lo que ocurre en una union (a diferencia de una estructura) es que una variable de tipo union, almacena sólo uno de los valores definido por ese tipo. El Programa 8.12 demuestra el uso de la union de C.

Programa 8.12 #include <stdio.h>

```
main()
{
    union /* Definición de una union. */
    {
        int valor_entero;
        float valor_real;
    } entero_o_real;

    printf("Tamaño de la union => %d bytes.\n",
        sizeof(entero_o_real));

    /* Introducir un entero e imprimirlo: */
    entero_o_real.valor_entero = 123;
    printf("El valor entero es %d\n", entero_o_real.valor_entero);
    printf("Direccion de comienzo => %d\n",
        &entero_o_real.valor_entero);

    /* Introducir un valor real e imprimirlo: */
    entero_o_real.valor_real = 123.45;
    printf("El valor real es %f\n", entero_o_real.valor_real);
    printf("Direccion de comienzo => %d\n", &entero_o_real.valor_real);
}
```

La ejecución del Programa 8.12 produce la siguiente salida:

```
Tamaño de la union => 4
El valor entero es 123
Direccion de comienzo => 7042
El valor real es => 123.45
Direccion de comienzo => 7042
```

Como puede apreciarse, la union ocupa un tamaño de 4 bytes. Esto se debe a que se necesitan 4 bytes para almacenar una variable de tipo `float`. En primer lugar, se almacena y se recupera un valor entero de esta posición de memoria (cuya dirección de comienzo en la ejecución es 7042). A continuación se almacena y recupera un real (`float`) en el mismo espacio de memoria.

Como se ha podido ver en el Programa 8.12, la declaración de una union tiene la misma forma que la declaración de una estructura. Todas las reglas vistas hasta este momento para las estructuras son igualmente aplicables a las uniones. La cantidad de memoria que requiere una variable de tipo union es la cantidad de memoria que necesita el miembro más grande de la union. Todos los miembros se almacenan en la misma zona de memoria (pero no todos al mismo tiempo) con la misma dirección de comienzo.

Cómo asignar valor inicial a una estructura en C

Sólo se puede asignar valor inicial a variables de tipo estructura que sean globales o estáticas. Esto se muestra en el Programa 8.13.

```
Programa 8.13 #include <stdio.h>
typedef struct
{
    char pieza[20];          /* Tipo de pieza.          */
    int cantidad;            /* Número de piezas.    */
    float precio_unitario;   /* Precio de cada pieza. */
} registro_piezas;

main()
{
    static registro_piezas contenido_caja_1 =
    {
        "Resistencias",
        25,
        0.05
    };
    static registro_piezas contenido_caja_2 =
    {
        "Condensadores",
        37,
        0.16
    };
}
```

```

printf("Contenido de la caja #1:\n");
printf("Elemento => %s\n", contenido_caja_1.pieza);
printf("Cantidad => %d\n", contenido_caja_1.cantidad);
printf("Precio unitario => $%f\n",
       contenido_caja_1.precio_unitario);

printf("Contenido de la caja #2:\n");
printf("Elemento => %s\n", contenido_caja_2.pieza);
printf("Cantidad => %d\n", contenido_caja_2.cantidad);
printf("Precio unitario => $%f\n",
       contenido_caja_2.precio_unitario);
}

```

La ejecución del Programa 8.13 produce la siguiente salida:

```

Contenido de la caja #1:
Elemento => Resistencias
Cantidad => 25
Precio unitario => 0.05

Contenido de la caja #2:
Elemento => Condensadores
Cantidad => 37
Precio unitario => 0.016

```

Análisis del programa

El Programa 8.13 comienza definiendo un tipo de estructura denominado `registro_piezas`.

```

typedef struct
{
    char pieza[20];           /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
} registro_piezas;

```

Ahora `registro_piezas` es un tipo que representa a una estructura con tres miembros. Cada miembro contiene información acerca de las piezas de una caja. En la función `main()`, se asignan los datos para cada uno de los tres miembros. Observe que se definen dos variables estáticas (`static`) denominadas `contenido_caja_1` y `contenido_caja_2`, ambas de tipo `registro_piezas`:

```

static registro_piezas contenido_caja_1 =
{
    "Resistencias",
    25,
    0.05
};

```

```
static registro_piezas contenido_caja_2 =
{
    "Condensadores",
    37,
    0.16
};
```

En este momento hay dos estructuras del mismo tipo, `contenido_caja_1` y `contenido_caja_2`. Los miembros de ambas estructuras son iguales, pero los datos asignados a cada uno de ellos son diferentes.

El resto del programa utiliza el operador de miembro para imprimir los datos contenidos en cada estructura:

```
printf("Contenido de la caja #1:\n");
printf("Elemento => %s\n", contenido_caja_1.pieza);
printf("Cantidad => %d\n", contenido_caja_1.cantidad);
printf("Precio unitario => $%f\n", contenido_caja_1.precio_unitario);

printf("Contenido de la caja #2:\n");
printf("Elemento => %s\n", contenido_caja_2.pieza);
printf("Cantidad => %d\n", contenido_caja_2.cantidad);
printf("Precio unitario => $%f\n", contenido_caja_2.precio_unitario);
```

Vectores y matrices de estructuras

La potencia real del empleo de estructuras en C aparece cuando se emplean **vectores o matrices de estructuras**. Considere un programa de inventario donde hay diferentes cajas de piezas. Para cada caja se desea conocer el nombre de la pieza, la cantidad y el precio de cada pieza. De esta forma usted podría querer conocer la misma estructura de información sobre todas las cajas de piezas. Este concepto se ilustra en la Figura 8.4.

Debido a que la estructura de información para cada una de ellas es la misma, se puede escribir un programa en C con un vector del mismo tipo de estructura. Este concepto se muestra en la Figura 8.5.

El Programa 8.14, es un programa en C que contiene un vector de tres estructuras. Este programa permite al usuario introducir —para cada una de las tres cajas— el nombre de la pieza, el número de piezas en la caja y el precio de cada pieza. El programa imprimirá a continuación lo que el usuario ha introducido. Esto se consigue definiendo un tipo de estructura y creando a continuación un vector de este tipo.

Programa 8.14

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

typedef struct
{
    char pieza[20];          /* Tipo de pieza. */
    int cantidad;           /* Número de piezas. */
    float precio;            /* Precio unitario. */
};
```

```

        float precio_unitario; /* Precio de cada pieza. */
        char existe;           /* Comprobar si el registro existe. */
    } registro_piezas;

main()
{
    static registro_piezas contenido_cajas[3];
    int registro;
    int i;

    do
    {
        /* Leer el número de caja. */
        printf("Introduzca un numero de caja entre 1 y 3 (0 para salir) => ");
        scanf("%d", &registro);

        --registro; /* Comienza en la caja 0. */
        if(registro < 0) continue;

        /* Leer el nombre de la pieza. */
        printf("Nombre de la pieza => ");
        scanf("%s", contenido_cajas[registro].pieza);

        /* Leer el número de piezas. */
        printf("Numero de piezas => ");
        scanf("%d", &contenido_cajas[registro].cantidad);

        /* Leer el precio de cada pieza. */
        printf("Precio de cada pieza => ");
        scanf("%f", &contenido_cajas[registro].precio_unitario);

        /* Indicar que el registro tiene datos, V */
        contenido_cajas[registro].existe = 'V';
    } while (registro >= 0);

    /* Imprimir la información. */
    for(registro = 0; registro <= 2; registro++)
    {
        if(contenido_cajas[registro].existe == 'V')
        {
            printf("La caja %d contiene:\n", registro + 1);
            printf("Pieza=> %s\n", contenido_cajas[registro].pieza);
            printf("Cantidad => %d\n",
                contenido_cajas[registro].cantidad);
            printf("Precio unitario => $%f\n",
                contenido_cajas[registro].precio_unitario);
        }
    } /* Fin for. */
}

```

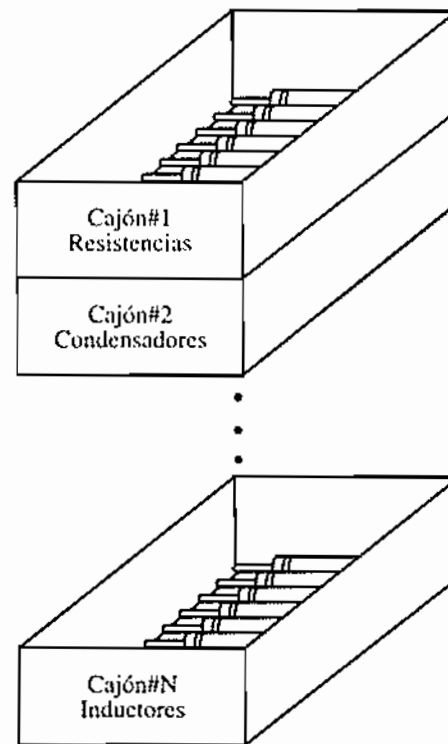


Figura 8.4. Cajones de almacenamiento con distintos componentes.

La ejecución del Programa 8.14 produce la siguiente salida:

```
Introduzca un numero de caja entre 1 y 3 (0 para salir) => 1
Nombre de la pieza => Inductancia
Numero de piezas => 12
Precio de cada pieza => 0.18
```

```
Introduzca un numero de caja entre 1 y 3 (0 para salir) => 3
Nombre de la pieza => Resistencia
Numero de piezas => 25
Precio de cada pieza => 0.04
```

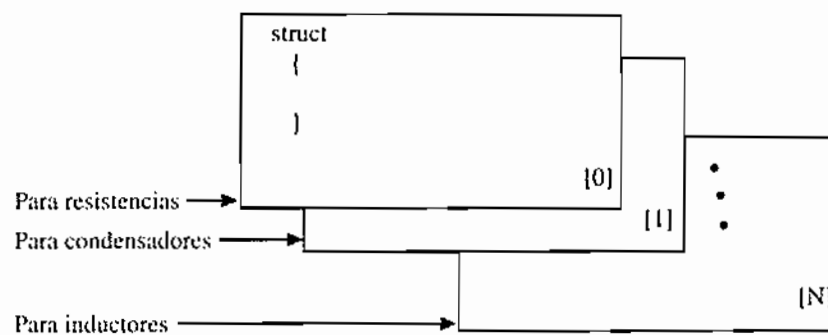


Figura 8.5. Creando un vector de estructuras.

Introduzca un numero de caja entre 1 y 3 (0 para salir) => 1

La caja 1 contiene:

Pieza => Inducatanca

Cantidad => 12

Precio unitario => \$0.18

La caja 3 contiene:

Pieza => Resistencia

Cantidad => 25

Precio unitario => \$0.14

Observe que no se ha introducido ningún dato para la caja número 2, y por lo tanto no se ha mostrado ningún dato para ella.

Análisis del programa

El Programa 8.14 define un tipo denominado `registro_piezas`, que es una estructura de cuatro miembros:

```
typedef struct
{
    char pieza[20];           /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
    char existe;              /* Comprobar si el registro existe. */
} registro_piezas;
```

El punto importante es el que sigue a continuación. Se ha definido un vector compuesto de tres elementos. Esta nueva variable se denomina `contenido_cajas[3]` y se define de tipo `registro_piezas`. Esto significa que se tiene un vector de tres elementos, cada uno de los cuales es una estructura de cuatro miembros definida según `typedef`.

```
main()
{
    static registro_piezas contenido_cajas[3];
```

Esto declara un vector de tres estructuras, cada una de las cuales tiene el mismo número y tipo de miembros. La siguiente parte del programa solicita al usuario que introduzca el número de una caja. Este valor se utilizará como índice del vector. Sin embargo, debido a que en C los vectores comienzan en 0, al número introducido por el usuario se le resta 1. Esto se hace utilizando la operación `--registro`. La entrada del usuario se encuentra dentro de un bucle `do` de C que se repite mientras la entrada de usuario no sea menor que 0 (si el usuario introduce un 0 se le resta 1 y se finaliza, puesto que el registro se hace menor que 0).

```
do
{
    /* Leer el número de caja. */
```

```

printf("Introduzca un numero de caja entre 1 y 3 (0 para salir) =>
");
scanf("%d", &registro);

--registro;          /* Comienza en la caja 0. */
if(registro < 0) continue;

```

Una vez leída la entrada, la forma de seleccionar una estructura específica del vector y un miembro determinado de esa estructura es mediante `variable_estructura[N].miembro_estructura`, siendo `N` el índice del vector.

```

/* Leer el número de piezas. */
printf("Numero de piezas => ");
scanf("%d", &contenido_cajas[registro].cantidad);

/* Leer el precio de cada pieza. */
printf("Precio de cada pieza => ");
scanf("%f", &contenido_cajas[registro].precio_unitario);

/* Indicar que el registro tiene datos, V */
contenido_cajas[registro].existe = 'V';
scanf("%f", &contenido_cajas[registro].precio_unitario);

```

La definición de la variable `registro_piezas` de forma estática permite asignar 0 a todos sus campos, con lo que se consigue que el miembro `existe` tome inicialmente el valor 0 (FALSO). Esto es necesario, para que los miembros de la estructura no tengan valores aleatorios al comienzo de la ejecución del programa. Así cuando se almacenan datos en un determinado registro se utiliza:

```

/* Indicar que el registro tiene datos, V */
contenido_cajas[registro].existe = 'V';

```

para indicar que el registro tiene datos almacenados en él. El programa emplea a continuación un bucle `for` para imprimir los datos. Es importante observar que sólo se muestran los registros en los que se ha almacenado algún valor (aquellos cuyo miembro `existe` toma el valor 'V').

```

for(registro = 0; registro <= 2; registro++)
{
    if(contenido_cajas[registro].existe == 'V')

        La salida se imprime simplemente utilizando la función printf.

    {
        printf("La caja %d contiene:\n", registro + 1);
        printf("Pieza => %s\n", contenido_cajas[registro].pieza);
        printf("Cantidad => %d\n", contenido_cajas[registro].cantidad);
        printf("Precio unitario => $%f\n",
            contenido_cajas[registro].precio_unitario);
    }
}

```

Conclusión

Esta sección ha presentado la unión de C. También se ha visto como combinar las estructuras y los vectores en C. Esta combinación es una herramienta muy potente para manejar estructuras de datos complejas. En la siguiente sección, se verá como se pueden mezclar estructuras y vectores para manejar estructuras de datos aún más complejas. Comprueba ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.5

1. Describa el propósito de la unión de C.
2. ¿Cómo se declara una unión en C?
3. ¿Cómo se puede asignar valor inicial a una estructura en un programa?
4. Explique la forma de declarar un vector de estructuras en un programa.
5. Indique cómo se puede acceder a un miembro determinado de un vector de estructuras.

8.6. FORMAS DE REPRESENTAR ESTRUCTURAS

Presentación

Las estructuras en C son ricas en la variedad de formas en las que pueden ser utilizadas. Esta sección presentará algunas de ellas. Encontrará muchas aplicaciones técnicas para esta potente característica de C.

Estructuras dentro de estructuras

En C se puede incluir una estructura dentro de otra. Considere el Programa 8.15.

Programa 8.15

```
#include <stdio.h>

typedef struct
{
    int miembro_1;
    char miembro_2;
} primera_estructura;

struct segunda_estructura
{
    primera_estructura segundo_miembro_1;
    int                segundo_miembro_2;
    char               segundo_miembro_3;
}

main()
{
}
```

Como se puede apreciar en el Programa 8.15, la estructura `segunda_estructura` contiene un miembro denominado `segundo_miembro_1` de tipo `primera_estructura`. Este es un caso de una estructura de C que contiene un miembro que es a su vez otra estructura. El programa de inventario de piezas, podría utilizar una estructura dentro de una estructura. Por ejemplo, el fabricante de la pieza podría ser una estructura que incluyese el nombre, la dirección y el número de teléfono como miembros.

Vectores y matrices dentro de estructuras

Usted también puede declarar vectores o matrices dentro de una estructura. Por ejemplo, considere el Programa 8.16. En él, uno de los miembros de la estructura es una matriz de dos dimensiones. Recuerde que usted ya empleo vectores de caracteres para representar cadenas de caracteres en estructuras anteriores. En esto no hay ninguna diferencia.

Programa 8.16 `#include <stdio.h>`

```

struct estructura
{
    float matriz[5][6];
    int    segundo_miembro_2;
    char   segundo_miembro_3;
}

main()
{
}

```

Como se puede ver en el Programa 8.16, el miembro `matriz[5][6]` es una matriz de dos dimensiones contenida dentro de una estructura de C

Vectores multidimensionales de estructuras

Usted también puede tener vectores multidimensionales cuyos elementos sean estructuras. Considere el Programa 8.17. Aquí se presenta una matriz de dos dimensiones cuyos elementos son estructuras que contienen tres miembros, uno de ellos es a su vez una matriz de dos dimensiones.

Programa 8.17 `#include <stdio.h>`

```

typedef struct
{
    float matriz[5][6];
    int    segundo_miembro_2;
    char   segundo_miembro_3;
}

```

```

    } registro;

    typedef registro matriz_compleja[3][2];

    main()
    {
    }

```

Como se puede observar en el Programa 8.17, `matriz_compleja[3][2]` es una matriz de dos dimensiones de tipo `registro`. De esta forma se tiene una estructura que consta de $3 \times 2 = 6$ estructuras, cada una de las cuales contiene a su vez tres miembros, uno de los cuales es también una matriz.

Conclusión

Con C se puede disponer de una rica variedad de estructuras y vectores. En esta sección se han visto algunos ejemplos de las diferentes posibilidades. Compruebe ahora lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.6

1. ¿Puede estar compuesto un vector de estructuras? Explique su respuesta.
2. ¿Puede contener una estructura a su vez otra estructura? Explique su respuesta.
3. ¿Es posible que un vector esté formado por estructuras que contengan a su vez un vector como uno de sus miembros? Explique su respuesta.

8.7. ESTRUCTURAS DE DATOS AVANZADAS

En esta sección vamos a examinar algunas estructuras de datos avanzadas típicas. Estas estructuras están basadas en la utilización de *nodos* de datos. Un **nodo**, en su forma más básica, consta de dos partes: un área de datos y un puntero. En el área de datos se puede almacenar cualquier cosa (enteros, reales, caracteres, cadenas de caracteres, vectores, etc.). El área de punteros contiene al menos un puntero (dirección) que apunta a otro nodo de información. La Figura 8.6 muestra la estructura de un nodo. La dos porciones del nodo tienen los nombres *datos* y *enlace* y se denominan normalmente *campo de datos* y *campo de enlace*. El campo de datos del nodo contienen el código para la letra 'x'. Normalmente, un grupo de nodos se asocia con una estructura de datos. La estructuras de datos que se van a examinar a continuación son las *listas enlazadas*, las *pilas*, las *colas* y los *arboles binarios*.

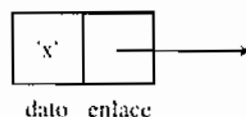


Figura 8.6. Un nodo que contiene la letra 'x'.

Listas enlazadas

Como se mencionó anteriormente, una estructura de datos está compuesta normalmente de un grupo de nodos. En un vector, cada elemento ocupa una posición de memoria consecutiva. Los nodos, por otra parte, pueden estar situados en posiciones diferentes dentro de la memoria que no tienen porque ser contiguas. Por ello es necesario el campo de enlace. El campo de enlace contiene un puntero (indicado por una flecha que sale del campo de enlace) que apunta al siguiente nodo (si lo hay) al que se encuentra conectado. La Figura 8.7 muestra como tres nodos se conectan juntos a través de sus campos de enlace.

Este conjunto de nodos se denomina **lista enlazada**. Observe que el campo de enlace del primer nodo apunta al segundo nodo, y que el campo de enlace del segundo nodo apunta al tercer nodo. El campo de enlace de este tercer nodo contiene NULL. Esto indica que el tercer nodo constituye el *fin* de la lista enlazada. La lista enlazada en su totalidad se encuentra apuntada por la variable `lista`.

Para definir el nodo de una lista enlazada en C se utiliza la siguiente estructura:

```
typedef struct nodo
{
    char data;           /* campo de datos. */
    struct nodo *enlace; /* campo de enlace. */
} LISTA;
```

El campo de datos en la estructura `LISTA` reserva espacio para un único carácter. Esta parte de la estructura puede cambiarse si la lista se utilizase para almacenar enteros, reales, vectores o algún otro tipo de datos. El campo de enlace se escribe de tal forma que contenga un puntero a una estructura similar `LISTA`.

¿Cómo se crean los nodos? Para responder a esta pregunta, revise cómo se creaba una cadena de caracteres. En C, la sentencia

```
char cadena[20] = "Pulse Return";
```

reserva 20 posiciones consecutivas de memoria (más una vigésimoprimera para el terminador `'\0'`) para los caracteres de la cadena. Este tipo de asignación de memoria se realiza cuando se compila el programa. Durante la ejecución, sólo las 20 posiciones de memoria reservadas pueden ser utilizadas para la cadena de caracteres. Si el programa intenta utilizar más de 20 posiciones (por ejemplo en una operación `strcat()`) es posible que se sobrescriban otras posiciones de memoria reservadas. Por esta razón, los nodos se crean de forma *dinámica* (cada vez que se necesiten) asignándoles en tiempo de ejecución posiciones de memoria no utilizadas. De esta forma, se pueden crear y utilizar nuevos nodos siempre que haya memoria libre disponible. ¿Ve la diferencia entre la asignación de memoria en tiempo de compilación y la asignación de memoria en tiempo de ejecución?

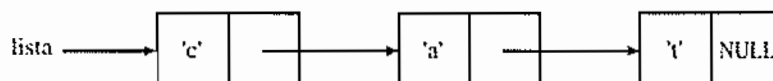


Figura 8.7. Una lista enlazada de tres nodos.

El archivo de cabecera `<stdlib.h>` contiene una función denominada `malloc()`, que se utiliza para asignar memoria dinámica en tiempo de ejecución. `malloc` recibe el número de bytes que necesitan ser asignados, y devuelve un puntero (una dirección) al bloque de bytes asignados si se encuentra memoria disponible. Si no hay memoria disponible para satisfacer la petición, `malloc` devuelve un puntero `NULL`. Así, para reservar memoria para un nuevo nodo de una lista, se utilizará la siguiente sentencia:

```
ptr = malloc(sizeof(LISTA))
```

La función `sizeof()` determina el tamaño, en bytes, que ocupa la estructura `LISTA`. La dirección de memoria asignada por `malloc` se asigna a la variable puntero `ptr`, que debe definirse de la siguiente forma:

```
LISTA *ptr;
```

La asignación de memoria de esta forma permite que el tamaño de las estructuras de datos se determine durante la *ejecución* del programa y no durante su compilación. La ventaja es clara: no se tiene que reservar memoria por adelantado para las estructuras de datos. Por ejemplo, un programa de ordenación puede que no conozca por adelantado el número de elementos a ordenar. De esta forma, ¿cuántos elementos se deberían reservar? ¿10? ¿20? ¿1000? Incluso así, ¿qué ocurriría si se introdujese un número más de los que el programador ha reservado? ¿Quién sabe lo que ocurriría?

El Programa 8.18 muestra la forma de crear una lista enlazada. El usuario simplemente introduce caracteres uno a uno y éstos se sitúan en nuevos nodos asignados a tal efecto. La lista se termina cuando el usuario pulsa la tecla de retorno de carro.

Programa 8.18

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

typedef struct nodo
{
    char dato;
    struct nodo *enlace;
} LISTA;

void mostrar_lista(LISTA *ptr);
void insertar(LISTA **ptr, char elemento);

main()
{
    LISTA *n1 = NULL;
    char elemento;

    do
    {
        printf("\nIntroduzca elemento: ");
        elemento = getchar();
        if(elemento != '\r')
            insertar(&n1, elemento);
    } while(elemento != '\r');
```

```

        printf("\nLa nueva lista enlazada es: ");
        mostrar_lista(n1);
    }

void mostrar_lista(LISTA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c", ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

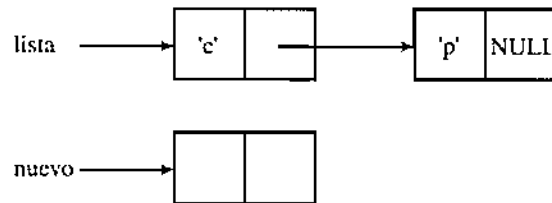
void insertar(LISTA **ptr, char elemento)
{
    LISTA *p1, *p2;

    p1 = *ptr;
    if(p1 == NULL)
    {
        p1 = malloc(sizeof(LISTA));
        if (p1 != NULL)
        {
            p1->dato = elemento;
            p1->enlace = NULL;
            *ptr = p1;
        }
    }
    else
    {
        while(p1->enlace != NULL)
            p1 = p1->enlace;
        p2 = malloc(sizeof(LISTA));
        if(p2 != NULL)
        {
            p2->dato = elemento;
            p2->enlace = NULL;
            p1->enlace = p2;
        }
    }
}

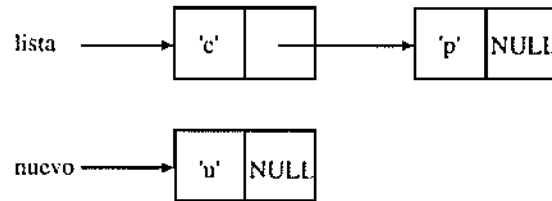
```

¿Qué ocurre cuando se añade un nuevo nodo a la lista enlazada? La Figura 8.8 muestra las tres etapas que requiere el proceso de añadir un nuevo nodo a una lista enlazada ya existente. Observe que el nodo nuevo se añade al final de la lista. Cuando la lista se encuentra inicialmente vacía, este proceso requiere una tercera etapa diferente, donde a la variable lista se le asigna la dirección del nuevo nodo.

1. Obtener el nuevo nodo:



2. Asignar el campo de datos y enlace:



3. Establecer un enlace a la lista que ya existe:

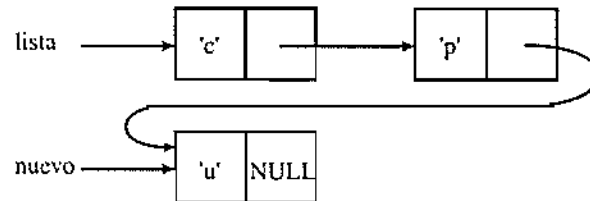


Figura 8.8. Añadir un nodo a una lista enlazada.

A continuación se muestra un ejemplo de ejecución del Programa 8.19.

```

Introduzca elemento: c
Introduzca elemento: p
Introduzca elemento: u
Introduzca elemento: La nueva lista enlazada es: cpu
  
```

Los próximos tres programas contienen listas predefinidas cuyo objetivo es ilustrar las operaciones básicas que se realizan sobre listas enlazadas. Estas son: insertar, borrar y buscar.

El Programa 8.19 muestra cómo insertar la letra 's' al comienzo de una lista enlazada compuesta por tres elementos, 'cat', para dar como resultado la lista enlazada 'scat', y como insertar al final de esta lista la letra 'm' para dar lugar a la lista 'scatm'.

Programa 8.19

```

#include <stdio.h>
#include <stdlib.h>

typedef struct nodo
{
  
```

```

    char dato;
    struct nodo *enlace;
} LISTA;

void mostrar_lista(LISTA *ptr);
void insertar_al_principio(LISTA **ptr, char item);
void insertar_al_final(LISTA *ptr, char item);

main()
{
    LISTA *n1, *n2, *n3;

    n1 = malloc(sizeof(LISTA));
    n2 = malloc(sizeof(LISTA));
    n3 = malloc(sizeof(LISTA));

    n1->dato = 'c';
    n1->enlace = n2;
    n2->dato = 'a';
    n2->enlace = n3;
    n3->dato = 't';
    n3->enlace = NULL;

    printf("La lista enlazada es como sigue: ");
    mostrar_lista(n1);
    insertar_al_principio(&n1, 's');
    printf("La nueva lista enlazada es: ");
    mostrar_lista(n1);
    insertar_al_final(&n1, 'm');
    printf("La nueva lista enlazada es: ");
    mostrar_lista(n1);
}

void mostrar_lista(LISTA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c", ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

void insertar_al_principio(LISTA **ptr, char item)
{
    LISTA *new;

    new = malloc(sizeof(LISTA));
    if(new != NULL)
    {
        new->dato = item;
        new->enlace = *ptr;
    }
}

```

```

        *ptr = new;
    }
}

void insertar_al_final(LISTA *ptr, char item)
{
    LISTA *new;

    while(ptr->enlace != NULL)
        ptr = ptr->enlace;
    new = malloc(sizeof(LISTA));
    if(new != NULL)
    {
        ptr->enlace = new;
        new->dato = item;
        new->enlace = NULL;
    }
}

```

La ejecución del Programa 8.19 produce la siguiente salida:

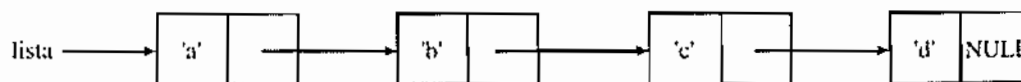
```

La lista enlazada es como sigue: cat
La nueva lista enlazada es: scat
La nueva lista enlazada es: scatm

```

La Figura 8.9 muestra como borrar un elemento del comienzo de una lista enlazada. La dirección a la que se dirige la variable `lista` pasa a apuntar a la dirección indicada por el campo de enlace del primer nodo de la lista. Esto hace que la variable `lista` apunte en este momento al segundo nodo. A continuación se libera la memoria reservada para el primer nodo, para así poder ser reutilizada en un futuro. Esta memoria se libera utilizando la función de biblioteca `free()` del archivo de cabecera `<stdlib.h>`. Si no se libera la memoria asignada para el primer nodo, ésta seguirá estando asignada, pero debido a que la variable `lista` ha sido cambiada, no habrá forma de acceder de nuevo al primer nodo.

1. Lista original:



2. Lista nueva:

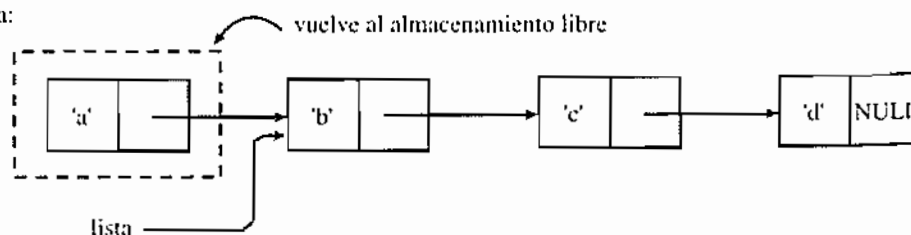


Figura 8.9. Quitar un nodo de la cabecera de una lista enlazada.

Esto hace que el primer nodo se convierta en *basura*. Cuando se descartan muchos nodos de esta forma, sin liberar la memoria utilizada por ellos, es posible que en un momento determinado no haya memoria disponible para nuevos nodos. Una ingeniosa rutina de recolección de *basura* (*garbage collection*) debe utilizarse para recuperar todos los nodos perdidos.

El Programa 8.20 muestra como borrar un elemento del comienzo de una lista enlazada y como borrar también un elemento del final de la lista.

Programa 8.20

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

typedef struct nodo
{
    char dato;
    struct nodo *enlace;
} LISTA;

void mostrar_lista(LISTA *ptr);
void eliminar_por_el_principio(LISTA **ptr);
void eliminar_por_el_final(LISTA **ptr);

main()
{
    LISTA *n1, *n2, *n3, *n4;

    n1 = malloc(sizeof(LISTA));
    n2 = malloc(sizeof(LISTA));
    n3 = malloc(sizeof(LISTA));
    n4 = malloc(sizeof(LISTA));

    n1->dato = 'a';
    n1->enlace = n2;
    n2->dato = 'b';
    n2->enlace = n3;
    n3->dato = 'c';
    n3->enlace = n4;
    n4->dato = 'd';
    n4->enlace = NULL;

    printf("La lista enlazada es como sigue: ");
    mostrar_lista(n1);
    eliminar_por_el_principio(&n1);
    printf("La nueva lista enlazada es: ");
    mostrar_lista(n1);
    eliminar_por_el_final(&n1);
    printf("La nueva lista enlazada es: ");
    mostrar_lista(n1);
}

void mostrar_lista(LISTA *ptr)
{
```

```

        while(ptr != NULL)
        {
            printf("%c", ptr->dato);
            ptr = ptr->enlace;
        }
        printf("\n");
    }

void eliminar_por_el_principio(LISTA **ptr)
{
    LISTA *p;
    p = *ptr;
    if(p != NULL)
    {
        p = p->enlace;
        free(*ptr);
    }
    *ptr = p;
}

void eliminar_por_el_final(LISTA **ptr)
{
    LISTA *p1, *p2;
    p1 = *ptr;
    if(p1 != NULL)
    {
        if(p1->enlace == NULL)
        {
            free(*ptr);
            *ptr = NULL;
        }
        else
        {
            while(p1->enlace != NULL)
            {
                p2 = p1;
                p1 = p1->enlace;
            }
            p2->enlace = NULL;
            free(p1);
        }
    }
}

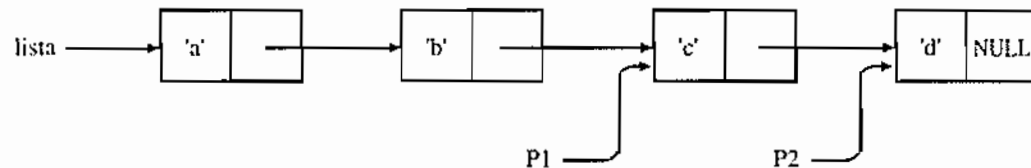
```

El borrado de un nodo del final de la lista enlazada requiere una búsqueda para encontrar el último nodo de la lista. La Figura 8.10 muestra todas las etapas necesarias. Es importante, en este caso, utilizar dos punteros durante del proceso de borrado. Es insuficiente un sólo puntero al último nodo, puesto que el campo de enlace del penúltimo nodo debe cambiarse para apuntar a NULL.

1. Lista original:



2. Encuentra el último nodo:



3. Reasignando el campo de enlace:

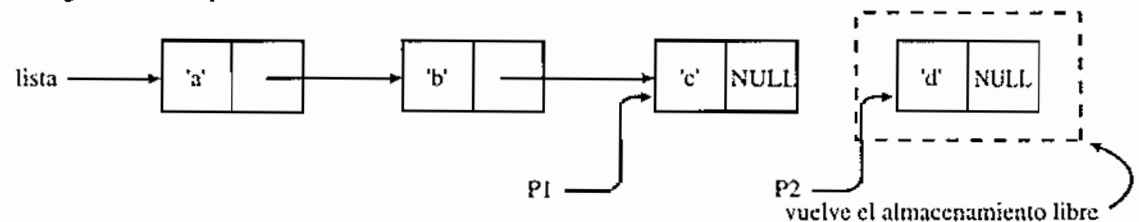


Figura 8.10. Quitar un nodo del final de una lista enlazada.

La salida que produce la ejecución del Programa 8.20 es la siguiente:

La lista enlazada es como sigue: abcd
 La nueva lista enlazada es: bcd
 La nueva lista enlazada es : bc

Observe que en los dos programas anteriores la complejidad de las funciones de inserción y borrado derivan del hecho de que la lista enlazada puede contener 0, 1 o más elementos, lo que requiere diferentes etapas en cada caso.

La última operación sobre listas enlazadas que se va a examinar es la de búsqueda de un elemento en una lista. El Programa 8.21 permite al usuario introducir un carácter que se buscará en una lista enlazada dada.

Programa 8.21

```

#include <stdio.h>
#include <stdlib.h>
typedef struct nodo
{
    char dato;
    struct nodo *enlace;
} LISTA;

void mostrar_lista(LISTA *ptr);
int buscar_en_lista(LISTA *ptr, char item);

main()
{
    LISTA *n1, *n2, *n3, *n4;
    char item;
  
```

```

    int encontrado;
    n1 = malloc(sizeof(LISTA));
    n2 = malloc(sizeof(LISTA));
    n3 = malloc(sizeof(LISTA));
    n4 = malloc(sizeof(LISTA));

    n1->dato = 'a';
    n1->enlace = n2;
    n2->dato = 'b';
    n2->enlace = n3;
    n3->dato = 'c';
    n3->enlace = n4;
    n4->dato = 'd';
    n4->enlace = NULL;

    printf("La lista enlazada es como sigue: ");
    mostrar_lista(n1);
    printf("Introduzca un caracter a buscar en la lista: ");
    item = getchar();
    encontrado = buscar_en_lista(n1,item);
    printf("\nEl caracter %c ",item);
    encontrado ? printf(" ha ") : printf(" no ha ");
    printf("sido encontrado en la lista: ");
    mostrar_lista(n1);
}

void mostrar_lista(LISTA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c",ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

int buscar_en_lista(LISTA *ptr, char item)
{
    if(ptr == NULL)
        return(0);
    else
    {
        do
        {
            if(ptr->dato == item)
                return(1);
            ptr = ptr->enlace;
        } while(ptr != NULL);
        return(0);
    }
}

```

A continuación se muestran dos ejecuciones del Programa 8.21.

```
La lista enlazada es como sigue: abcd
Introduzca un caracter a buscar en la lista: e
El caracter e no ha sido encontrado en la lista: abcd

La lista enlazada es como sigue: abcd
Introduzca un caracter a buscar en la lista: c
El caracter c ha sido encontrado en la lista: abcd
```

Observe que el tiempo empleado en la búsqueda de un carácter que no está en la lista se incrementa cuando el número de nodos de la lista crece. Esto puede ser una consideración a tener muy en cuenta en aplicaciones con restricciones de tiempo críticas. Por esta razón se utilizan tablas de dispersión (tratadas en el Capítulo 7) para llevar a cabo búsquedas eficientes. Recuerde que el uso de una tabla de dispersión puede producir *colisiones*, cuando dos o más variables tienen la misma dirección en la tabla. Este problema se puede solucionar utilizando listas enlazadas. Si dos o más variables de la lista tienen la misma dirección, éstas pueden ordenarse en una lista enlazada para esa posición.

Pilas y colas

Muchas computaciones se ven simplificadas enormemente por el uso de pilas y colas controladas por software. La evaluación de expresiones y algoritmos de turno rotatorio (*round-robin*) son sólo dos ejemplos de las aplicaciones de las pilas y colas. La forma de implementarlas no es crítica y en esta sección se implementarán como listas enlazadas.

Una característica de una pila es que el último elemento que se introduce es siempre el primer elemento que se saca. Por esta razón las pilas se conocen generalmente como estructuras **LIFO** (del inglés *Last In First Out*, último en entrar primero en salir). En los Programas 8.19 y 8.20 se realizaron operaciones sobre el primer nodo de una lista enlazada. Estas operaciones son similares a las operaciones de introducción de datos en la pila (*push*)* y de extracción de datos (*pop*). La Figura 8.11 muestra el contenido de una pila durante varias operaciones de introducción *push* y de extracción *pop*. La variable *cima* hace referencia a la cima de la pila y cambia con cada operación *push* y *pop*. Observe que una operación *pop* recupera el último elemento introducido en la pila (una 'c'). El Programa 8.22 muestra la forma de implementar una pila de caracteres. Debido a que la pila se implementa como una lista enlazada, las operaciones *push()* y *pop()* manipulan el primer nodo de la lista enlazada que actúa como pila.

Programa 8.22

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

typedef struct nodo
{
```

* Aun siendo de uso frecuente el nombre original inglés **push** y **pop** hemos decidido utilizar en esta obra los términos *introducir* y *extraer*.

```

        char dato;
        struct nodo *enlace;
    } PILA;

void mostrar_pila(PILA *ptr);
int vacia(PILA *ptr);
void introducir(PILA **ptr, char item);
void extraer(PILA **ptr, char *item);

main()
{
    PILA *pila = NULL;
    char item;

    introducir(&pila, 'a');
    introducir(&pila, 'b');
    introducir(&pila, 'c');
    printf("La pila es como sigue: ");
    mostrar_pila(pila);
    extraer(&pila, &item);
    printf("\nEl primer elemento obtenido es %c\n", item);
    extraer(&pila, &item);
    printf("\nEl segundo elemento obtenido es %c\n", item);
    extraer(&pila, &item);
    printf("\nEl tercer elemento obtenido es %c\n", item);
    extraer(&pila, &item);
}

void mostrar_pila(PILA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c", ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

int vacia(PILA *ptr)
{
    if(ptr == NULL)
        return(1);
    else
        return(0);
}

void introducir(PILA **ptr, char item)
{
    PILA *p;

    if(vacia(*ptr))
    {
        p = malloc(sizeof(PILA));
    }
}

```

```

        if(p != NULL)
        {
            p->dato = item;
            p->enlace = NULL;
            *ptr = p;
        }
    }
else
{
    p = malloc(sizeof(PILA));
    if(p != NULL)
    {
        p->dato = item;
        p->enlace = *ptr;
        *ptr = p;
    }
}

void extraer(PILA **ptr, char *item)
{
    PILA *p1;

    p1 = *ptr;
    if(vacia(p1))
    {
        printf("Error! La pila esta vacia.\n");
        *item = '\0';
    }
    else
    {
        *item = p1->dato;
        *ptr = p1->enlace;
        free(p1);
    }
}

```

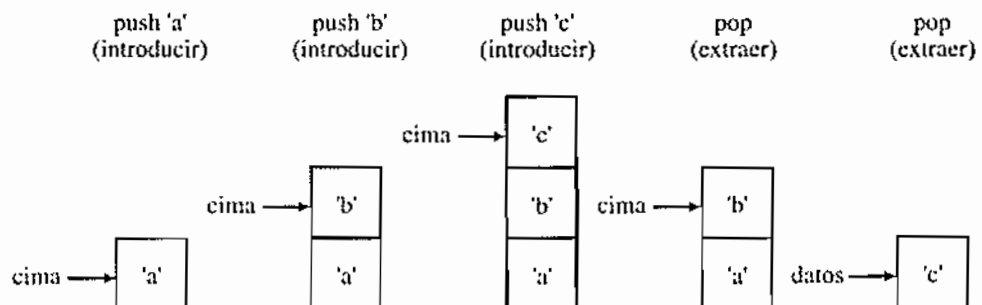


Figura 8.11. Operaciones en una pila durante una operación de introducción (*push*) y una operación de extracción (*pop*).

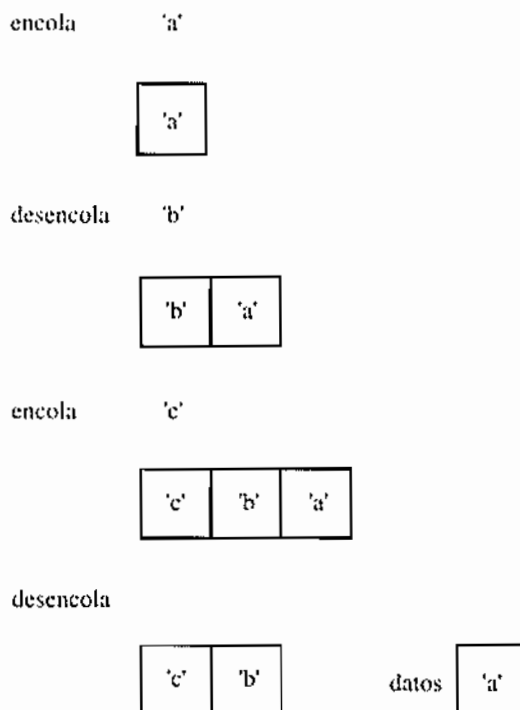


Figura 8.12. Operaciones de encolamiento y desencolamiento.

La ejecución del Programa 8.22 produce la siguiente salida:

La pila es como sigue: cba

El primer elemento obtenido es c
 El segundo elemento obtenido es b
 El tercer elemento obtenido es a
 Error! La pila esta vacia.

Es importante hacer uso de la función `vacía()` en la función `pop()`, debido a que no tiene sentido realizar una operación `pop` sobre una pila vacía.

Una estructura de datos muy similar a la pila es la **cola**. La diferencia más importante entre ambos tipos de estructura es que la cola es una estructura **FIFO** (del inglés *First In First Out*, primero en entrar primero en salir). La Figura 8.12 muestra un ejemplo de cola después de varias operaciones de insertado de elementos en la cola *encolar* (*en-queue*) y extracción de elementos de la cola *desencolar* (*de-queue*).

El Programa 8.23 muestra la forma de introducir datos en una cola y cómo eliminar datos de una cola. Al contrario de las operaciones sobre pilas del Programa 8.22, los datos en las colas se recuperan en el orden en el que se insertan.

Programa 8.23 `#include <stdio.h>`
`#include <stdlib.h>`

`typedef struct nodo`
`{`


```

        char dato;
        struct nodo *enlace;
    } COLA;

void mostrarCola(COLA *ptr);
int vacia(COLA *ptr);
void encolar(COLA **cabeza, char item);
void desencolar(COLA **cabeza, char *item);

main()
{
    COLA *cabeza = NULL;
    char item;

    encolar(&cabeza, 'a');
    encolar(&cabeza, 'b');
    encolar(&cabeza, 'c');
    printf("La cola es como sigue: ");
    mostrarCola(cabeza);
    desencolar(&cabeza, &item);
    printf("\nEl primer elemento obtenido de la cola es %c\n", item);
    desencolar(&cabeza, &item);
    printf("\nEl segundo elemento obtenido de la cola es %c\n", item);
    desencolar(&cabeza, &item);
    printf("\nEl tercer elemento obtenido de la cola es %c\n", item);
    desencolar(&cabeza, &item);
}

void mostrarCola(COLA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c", ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

int vacia(COLA *ptr)
{
    if(ptr == NULL)
        return(1);
    else
        return(0);
}

void encolar(COLA **cabeza, char item)
{
    COLA *p;

    p = malloc(sizeof(COLA));

```

```

        if(p != NULL)
        {
            p->dato = item;
            p->enlace = *cabeza;
            *cabeza = p;
        }
    }

void desencolar(COLA **cabeza, char *item)
{
    COLA *p1, *p2;

    p1 = *cabeza;
    if(vacia(p1))
    {
        printf("Error! La cola esta vacia.\n");
        *item = '\0';
    }
    else
    {
        p2 = *cabeza;
        while(p2->enlace != NULL)
        {
            p1 = p2;
            p2 = p2->enlace;
        }
        *item = p2->dato;
        p1->enlace = NULL;
        free(p2);
        if(p1 == p2)
            *cabeza = NULL;
    }
}

```

La ejecución del Programa 8.23 produce los siguientes resultados:

La cola es como sigue: cba

El primer elemento obtenido de la cola es a

El primer elemento obtenido de la cola es b

El primer elemento obtenido de la cola es c

Error! La cola esta vacia.

Una cola puede también implementarse como una lista *doblemente* enlazada, tal y como se muestra en la Figura 8.13. Se utilizan dos punteros para acceder a la cola. Estos dos punteros son *cabeza* (primer nodo) y *cola* (último nodo) de la cola. Observe que los nodos que constituyen la lista enlazada de la cola contienen dos campos de enlace. Un campo de enlace se utiliza para apuntar al siguiente nodo de la lista. El otro se utiliza para apuntar al nodo anterior de la lista. Esto permite fácilmente acceder a los datos situados al principio y final de la lista y elimina la necesidad de buscar el

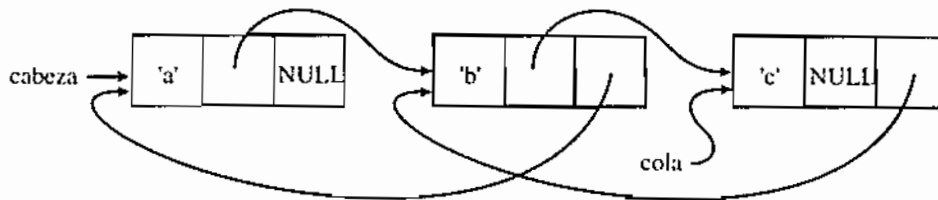


Figura 8.13. Una cola de tres elementos.

final de la lista cuando se realiza una operación de extracción de la cola (desencolar). La estructura C necesaria para este tipo de estructura podría ser la siguiente:

```
typedef struct nodo
{
    char dato;
    struct nodo *enlace_sig;
    struct nodo *enlace_ant;
} LISTA;
```

donde `enlace_sig` y `enlace_ant` representan el enlace a los elementos siguiente y anterior de la lista respectivamente.

Árboles binarios

Un **árbol binario** es una estructura de datos específica compuesta de nodos que contienen un campo de datos y dos campos de enlace, como se muestra en la Figura 8.14. Los campos de enlace se denominan *hijo izquierdo* e *hijo derecho*. El término binario viene del hecho de que cada nodo tienen la capacidad de apuntar exactamente a otros dos nodos. El primer nodo del árbol se denomina generalmente *nodo raíz*, y se encuentra situado en la parte superior del diagrama que representa el árbol. La Figura 8.15 muestra un ejemplo de árbol binario. El nodo raíz es el nodo del árbol que contiene el símbolo '*'. El árbol de esta figura se construyó de manera que cada campo de enlace apuntara a una de tres siguientes posibilidades: un nodo con una operación matemática, un nodo con el nombre de una variable o un NULL. Es importante poder recorrer el árbol binario y acceder a la información contenida en él. Durante el recorrido, los datos almacenados en cada nodo se imprimen o se acceden.

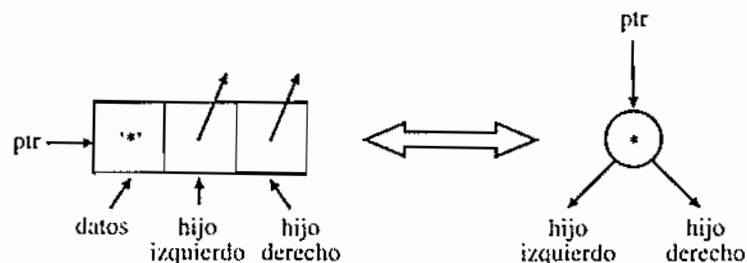


Figura 8.14. Un ejemplo de nodo de árbol binario.

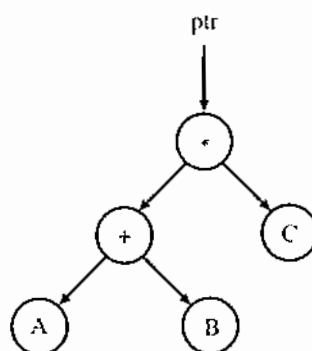


Figura 8.15. Un ejemplo de árbol binario.

Existen tres formas comunes de recorrer un árbol: en *preorden*, en *orden simétrico* (enorden) o en *postorden*. En todos estos métodos, se intenta recorrer el árbol hacia abajo y tan a la izquierda como sea posible antes de llegar a la derecha. El recorrido hacia abajo continúa hasta que se encuentra un campo de enlace con un NULL. Los resultados obtenidos difieren dependiendo del método de recorrido empleado, debido a que cada uno de éstos muestra o accede al campo de datos del nodo en un momento diferente.

Por ejemplo, en un recorrido en preorden, en cada nodo se realizan las siguiente etapas:

Acceso al campo de datos;
acceso al hijo izquierdo;
acceso al hijo derecho.

En un recorrido en orden simétrico, los campos se acceden del siguiente modo:

Acceso al hijo izquierdo;
acceso al campo de datos;
acceso al hijo derecho.

En un recorrido en postorden se realizan los siguientes pasos:

Acceso al hijo izquierdo;
acceso al hijo derecho;
acceso al campo de datos.

La Figura 8.16 muestra el orden en el que se pueden acceder a los nodos del árbol binario de ejemplo utilizando cada una de estas técnicas de recorrido. Los resultados de cada recorrido son los siguientes:

Preorden: *+ABC
Orden simétrico: A+B*C
Postorden: AB+C*

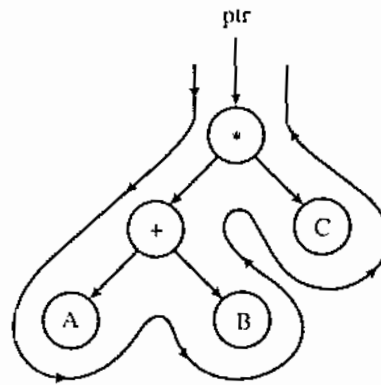


Figura 8.16. Recorrido en profundidad de un árbol binario.

Estas técnicas de recorrido son muy fáciles de implementar utilizando funciones recursivas. El Programa 8.24 ilustra estas funciones.

Programa 8.24

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

typedef struct nodo
{
    char dato;
    struct nodo *hijo_izq;
    struct nodo *hijo_der;
} ARBOL;

void PreOrden(ARBOL *ptr);
void OrdenSimetrico(ARBOL *ptr);
void PostOrden(ARBOL *ptr);

main()
{
    ARBOL *n1, *n2, *n3, *n4, *n5;

    n1 = malloc(sizeof(ARBOL));
    n2 = malloc(sizeof(ARBOL));
    n3 = malloc(sizeof(ARBOL));
    n4 = malloc(sizeof(ARBOL));
    n5 = malloc(sizeof(ARBOL));
    n1->dato = '*';
    n1->hijo_izq = n2;
    n1->hijo_der = n3;
    n2->dato = '+';
    n2->hijo_izq = n4;
    n2->hijo_der = n5;
    n3->dato = 'C';
    n3->hijo_izq = NULL;
    n3->hijo_der = NULL;
    n4->dato = 'A';
```

```

    n4->hijo_izq = NULL;
    n4->hijo_der = NULL;
    n5->dato = 'B';
    n5->hijo_izq = NULL;
    n5->hijo_der = NULL;
    printf("Recorrido en preorden => ");
    PreOrden(n1);
    printf("\nRecorrido en orden simetrico => ");
    OrdenSimetrico(n1);
    printf("\nRecorrido en postorden => ");
    PostOrden(n1);
}

void PreOrden(ARBOL *ptr)
{
    if(ptr != NULL)
    {
        printf("%c", ptr->dato);
        PreOrden(ptr->hijo_izq);
        PreOrden(ptr->hijo_der);
    }
}

void OrdenSimetrico(ARBOL *ptr)
{
    if(ptr != NULL)
    {
        OrdenSimetrico(ptr->hijo_izq);
        printf("%c", ptr->dato);
        OrdenSimetrico(ptr->hijo_der);
    }
}

void PostOrden(ARBOL *ptr)
{
    if(ptr != NULL)
    {
        PostOrden(ptr->hijo_izq);
        PostOrden(ptr->hijo_der);
        printf("%c", ptr->dato);
    }
}

```

Examine el método empleado para construir el árbol binario al comienzo del programa. El árbol representa una expresión matemática cuya forma original es:

$(A + B) * C$

Normalmente, el árbol binario que representa una expresión matemática se construye por medio de una función que sigue algunas de las guías de precedencia de

operadores (las cuales fueron vistas en la sección de programas de aplicación del Capítulo 7).

La ejecución del Programa 8.24 produce la siguiente salida:

```
Recorrido en preorden => *+ABC
Recorrido en orden simetrico => A+B*C
Recorrido en postorden => AB+C*
```

Se le anima para que invente sus propios árboles utilizando el Programa 8.24 y que realice sus propios recorridos.

Conclusión

Esta sección sobre estructuras de datos ha mostrado algunas de las herramientas fundamentales empleadas por los programadores. Estas potentes herramientas se utilizan con más eficacia cuando no se conoce el tamaño con el que se debería definir un vector o matriz o cuando no se conoce el número de estructuras que se debería asignar inicialmente. Empleando estas técnicas, se puede usar exactamente la cantidad de memoria necesaria. Compruebe lo que ha comprendido de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 8.7

1. Describa la estructura básica de un nodo.
2. Explique las diferencias que hay en uso de memoria entre una lista enlazada de caracteres y una cadena de caracteres ¿Tiene limitaciones alguna de estas estructuras?
3. ¿Qué tipos de datos se pueden almacenar en un nodo?
4. ¿Cuál es la importancia del puntero NULL?
5. ¿Cómo se asigna memoria en tiempo de ejecución?
6. ¿Cuáles son las operaciones básicas que se realizan sobre una lista enlazada, un pila, una cola y un árbol binario?
7. ¿Qué se entiende por FIFO y LIFO?

8.8. PROGRAMA DE APLICACIÓN: MINIMICRO

Presentación

En esta sección se examinará y desarrollará la operación de un simple microprocesador denominado MiniMicro. El conjunto de instrucciones de la máquina se muestra en la Tabla 8.1. Los tipos de instrucciones que incluye MiniMicro son los típicos que se encuentran en la mayoría de microprocesadores. La potencia de los tipos de datos enumerados (enum) permite definir LOAD, ADD y el resto de instrucciones que se muestra en la Tabla 8.1, sin tener que asignarlas ningún significado o valor.

Tabla 8.1. Juego de instrucciones MiniMicro

Sintaxis de la instrucción	Operación
LOAD valor	Carga el acumulador con valor
ADD valor	Suma valor al acumulador
SUB valor	Resta valor al acumulador
MUL valor	Multiplíca valor por el acumulador
DIV valor	Divide valor por el acumulador
PRINT	Imprime el contenido del acumulador
JNZ direccion	Salta a direccion si el acumulador es distinto de cero
JMP direccion	Salta a direccion
JSR direccion	Salta a una subrutina que empieza en direccion
RET	Vuelve de una subrutina
STOP	Detiene la ejecución

El problema

MiniMicro funciona como cualquier microprocesador, una y otra vez ejecuta la misma secuencia de operaciones:

- Lectura de la instrucción (*fetch*).
- Decodificación de la instrucción.
- Ejecución de la instrucción.

Debido a que la máquina debe acceder a memoria para leer una instrucción, se debe proporcionar algún tipo de memoria donde poder almacenar el programa y los datos y también un puntero que permita a MiniMicro acceder a la memoria. Algunas instrucciones utilizan el puntero para leer los datos durante la fase de ejecución (LOAD, ADD, SUB, MUL y DIV), y otras cargan el puntero con un nuevo valor (JNZ, JMP, JSR y RET).

Desarrollo del algoritmo

Las etapas que requiere la aplicación MiniMicro son las siguientes:

- Lectura de la instrucción (*fetch*).
- Lectura de los operandos de la instrucción de la siguiente posición.
- Decodificación de la instrucción.
- Ejecución de la instrucción.

Cada etapa requiere conocer el puntero de instrucción o contador de programa de la máquina. La etapa de ejecución emplea el acumulador de la máquina. La recursividad puede utilizarse para ejecutar las instrucciones JSR y RET, ya que ambas cambian y recuperan el contador de programa durante la ejecución. Para que el acumulador sea disponible en todos los niveles de la recursividad (en caso de que se utilicen subrutinas anidadas), éste se define como una variable *global*.

Desarrollo global

El Programa 8.25 contiene un bucle `do` que lleva a cabo todas las operaciones necesarias para ejecutar una instrucción simple. La sentencia `switch()` se usa para decodificar la instrucción leída. Cada línea de la sentencia `switch()` representa una de las instrucciones enumeradas de MiniMicro.

Programa 8.25 `#include <stdio.h>`

```

    /* Definir el conjunto de instrucciones */
    enum iset {LOAD, ADD, SUB, MUL, DIV, PRINT,
               JNZ, JMP, JSR, RET, STOP};

    /* Cargar el programa en memoria */
    int mem[256] = {LOAD, 100, JSR, 6, PRINT, STOP,
                   MUL, 9, DIV, 5, ADD, 32, RET};

    int acc;          /* Acumulador */

void ejecutar(int pc);

main()
{
    ejecutar(0);      /* Comenzar la ejecución en la dirección 0 */
}

void ejecutar(int pc)
{
    int ip;           /* Contador de programa */
    int inst;         /* Instrucción */
    int iop;          /* Operando de instrucción */

    ip = pc;          /* Cargar el contador de programa */
    do
    {
        inst = mem[ip++]; /* Lectura de la instrucción */
        iop = mem[ip++];  /* Leer el operando */
        switch(inst)
        {
            case LOAD : acc = iop; break;
            case ADD  : acc += iop; break;
            case SUB  : acc -= iop; break;
            case MUL  : acc *= iop; break;
            case DIV  : acc /= iop; break;
            case PRINT :
            {
                printf("ACUMULADOR = %5d\n", acc);
                ip--;
                break;
            }
            case JNZ : ip = (acc != 0) ? iop : ip; break;
        }
    } while (inst != STOP);
}

```

```

        case JMP : ip = iop; break;
        case JSR : ejecutar(iop); break;
        case RET : return; break;
        case STOP : break;
    }
    } while (inst != STOP);
}

```

Observe cómo se implementa la instrucción JSR:

```
case JSR : ejecutar(iop); break;
```

Se utiliza una llamada recursiva a `ejecutar()` para manejar las subrutinas. La dirección de comienzo de la subrutina se pasa a la función `ejecutar()` mediante `iop`. Esto permite que existan múltiples contadores de programa de forma simultánea.

Cuando se ejecuta la instrucción RET, su sentencia switch es la siguiente:

```
case RET : return; break;
```

Esta provoca que finalice el actual nivel de recursividad y que el control pase de nuevo al nivel anterior, donde se recupera el valor original del contador de programa.

El bucle `do` se repite hasta que se lee una instrucción STOP.

La Tabla 8.2 muestra un ejemplo de un programa MiniMicro que convierte 100 grados centígrados a grados Fahrenheit. Observe que la conversión se realiza en una subrutina.

La ejecución del Programa 8.25 produce la siguiente salida:

```
ACC = 212
```

la cual es la temperatura en grados Fahrenheit correcta.

Tabla 8.2. Programa de ejemplo con MiniMicro

Dirección	Instrucción	Comentario
0	LOAD 100	Carga el acumulador con el valor 100
2	JSR 6	Salta a la subrutina en la dirección 6
4	PRINT	Imprime el acumulador
5	STOP	Detiene el programa
6	MUL 9	Multiplica el acumulador por 9
8	DIV 5	Divide el acumulador por 5
10	ADD 32	Suma 32 al acumulador
12	RET	Vuelta de la subrutina

Conclusión

La máquina MiniMicro es un buen ejemplo de utilización de la enumeración en el mundo real. Incluso aunque la máquina tienen un conjunto limitado de instrucciones, la incorporación de más instrucciones es tan simple como añadir nuevos tipos enumerados a la lista, con las sentencias `case` adecuadas dentro de `switch`. Los programas tales como el 8.25 se denominan emuladores, debido a que emulan la operación de una máquina real, sin necesidad de que exista ésta.

8.9. PROGRAMAS DE APLICACIÓN ADICIONALES

Este capítulo concluye con cuatro programas más, diseñados para mostrar el uso de listas enlazadas, pilas colas y árboles binarios.

Contador de nodos

El Programa 8.26 contiene las sentencias necesarias para construir el árbol binario que representa la expresión de la Figura 8.15.

Programa 8.26

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

typedef struct nodo
{
    char dato;
    struct nodo *hijo_izq;
    struct nodo *hijo_der;
} ARBOL;

int numero_de_nodos = 0;
void contar_nodos(ARBOL *ptr);

main()
{
    ARBOL *n1, *n2, *n3, *n4, *n5;

    n1 = malloc(sizeof(ARBOL));
    n2 = malloc(sizeof(ARBOL));
    n3 = malloc(sizeof(ARBOL));
    n4 = malloc(sizeof(ARBOL));
    n5 = malloc(sizeof(ARBOL));
    n1->dato = '*';
    n1->hijo_izq = n2;
    n1->hijo_der = n3;
    n2->dato = '+';
    n2->hijo_izq = n4;
    n2->hijo_der = n5;
```

```

    n3->dato = 'C';
    n3->hijo_izq = NULL;
    n3->hijo_der = NULL;
    n4->dato = 'A';
    n4->hijo_izq = NULL;
    n4->hijo_der = NULL;
    n5->dato = 'B';
    n5->hijo_izq = NULL;
    n5->hijo_der = NULL;
    printf("Numero de nodos => ");
    contar_nodos(n1);
    printf("%d", numero_de_nodos++);
}

void contar_nodos(ARBOL *ptr)
{
    if(ptr != NULL)
    {
        numero_de_nodos++;
        contar_nodos(ptr->hijo_izq);
        contar_nodos(ptr->hijo_der);
    }
}

```

La función `contar_nodos()` recibe un puntero al nodo raíz del árbol. Esta función cuenta el número de nodos del árbol. Como se puede observar en la Figura 8.15, está claro que el árbol contiene cinco nodos. La ejecución del Programa 8.26 coincide con esto:

```
Numero de nodos => 5
```

La técnica utilizada para contar los nodos del árbol es similar a la utilizada para realizar un recorrido. Cada vez que se alcanza un nuevo nodo, se incrementa el contador de nodos. A continuación se examinan recursivamente los hijos izquierdo y derecho (si los hay).

Inversión de una cadena de caracteres mediante una pila

Las pilas tienen muchos usos cuando se utilizan en aplicaciones que manipulan cadenas de caracteres. Por ejemplo, se puede emplear una pila para verificar si una cadena de caracteres es un palíndromo, tal como `aabcebaa`. En este ejemplo, el Programa 8.27 emplea una pila para invertir una cadena de caracteres proporcionada por el usuario. La Figura 8.17 muestra el proceso básico. Los caracteres de la cadena se van insertando en la pila uno a uno. Cuando se finaliza el proceso, el último carácter de la cadena se encuentra en la cima de la pila. A medida que los caracteres son extraídos de la pila, se almacenan en una cadena de caracteres comenzando en la primera posición. De esta forma, la cadena de caracteres se invierte de forma automática.

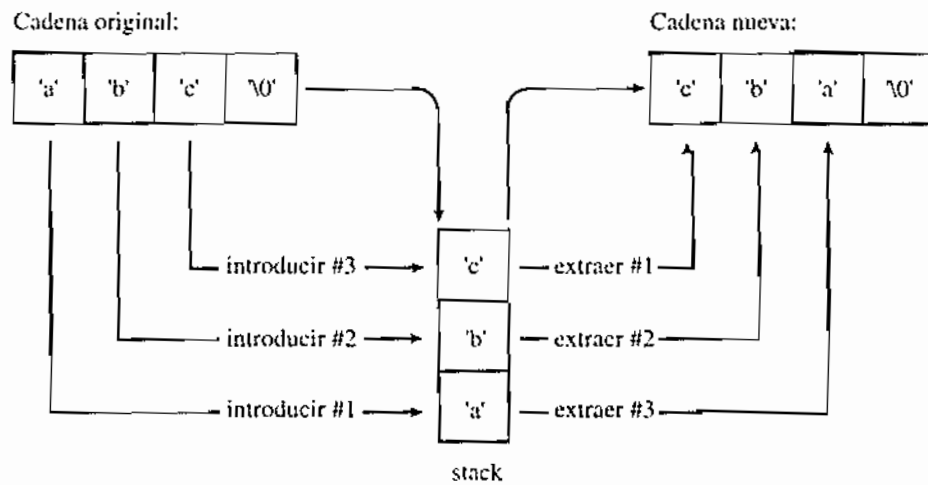


Figura 8.17. Inversión de una cadena de caracteres usando una pila.

El Programa 8.27 hace uso de la función de biblioteca `strlen()` para insertar los caracteres en la pila, y luego usa la función `vacía()` para extraer caracteres de la pila hasta que ésta se vacíe. Observe que el carácter `NULL` nunca se inserta en la pila, puesto que éste no se incluye en la longitud de la cadena de caracteres que determina la función `strlen()`.

Programa 8.27

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

#define TRUE 1
#define FALSE 0

typedef struct nodo
{
    char dato;
    struct nodo *enlace;
} PILA;

void mostrar_pila(PILA *ptr);
int vacia(PILA *ptr);
void introducir(PILA **ptr, char item);
void extraer(PILA **ptr, char *item);

main()
{
    PILA *pila = NULL;
    char cadena[40], item;
    int i;

    printf("Introduzca una cadena de caracteres: ");
    gets(cadena);
    for(i = 0; i < strlen(cadena); i++)
        introducir(&pila, cadena[i]);
```

```

        printf("\nLa pila es la siguiente: ");
        mostrar_pila(pila);
        i = 0;
        while(!vacía(pila))
        {
            extraer(&pila,&cadena[i]);
            i++;
        }
        printf("\nLa nueva cadena de caracteres es => %s",cadena);
    }

void mostrar_pila(PILA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c",ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

int vacía(PILA *ptr)
{
    if(ptr == NULL)
        return(TRUE);
    else
        return(FALSE);
}

void introducir(PILA **ptr, char item)
{
    PILA *p;

    if(vacía(*ptr))
    {
        p = malloc(sizeof(PILA));
        if(p != NULL)
        {
            p->dato = item;
            p->enlace = NULL;
            *ptr = p;
        }
    }
    else
    {
        p = malloc(sizeof(PILA));
        if(p != NULL)
        {
            p->dato = item;
            p->enlace = *ptr;
            *ptr = p;
        }
    }
}

```

```

    }
}

void extraer(PILA **ptr, char *item)
{
    PILA *p1;

    p1 = *ptr;
    if(vacia(p1))
    {
        printf("Error! La pila esta vacia.\n");
        *item = '\0';
    }
    else
    {
        *item = p1->dato;
        *ptr = p1->enlace;
        free(p1);
    }
}

```

A continuación se muestra un ejemplo de ejecución del Programa 8.27.

Introduzca una cadena de caracteres: Microprocesadores 123!

La pila es la siguiente: !321 serodasecorporciM

La nueva cadena de caracteres es =>: !321 serodasecorporciM

Imagine lo que se puede hacer si se utilizan dos o más pilas para procesar una cadena de caracteres.

Búsqueda binaria

La búsqueda binaria es una técnica de búsqueda muy eficaz empleada normalmente con árboles binarios. La Figura 8.18 muestra un ejemplo de árbol binario que contiene sus nodos enteros. El árbol se ha estructurado de tal forma que los enteros se han situado en sus posiciones correctas (con el propósito de búsqueda). Por ejemplo, observe que todos los nodos alcanzables desde el hijo izquierdo del nodo raíz tienen valores (2, 3, 5 y 6) más pequeños que el valor del nodo raíz (7). Todos los nodos alcanzables desde el hijo derecho del nodo raíz tienen valores (12, 13 y 17) más grandes que el valor del nodo raíz. La búsqueda de un valor determinado en el árbol —por ejemplo 6— involucra las siguientes comprobaciones:

1. ¿El valor a buscar coincide con el valor del nodo actual?

En este caso la respuesta es no, puesto el valor del nodo actual es 7.

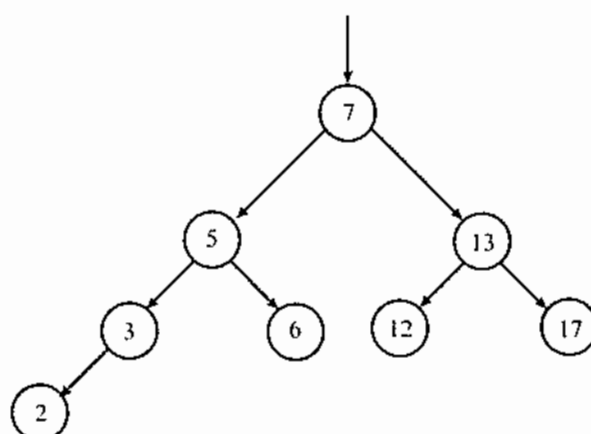


Figura 8.18. Árbol binario conteniendo enteros ordenados.

2. ¿El valor a buscar es más pequeño que el valor del nodo actual? Si la respuesta es correcta, el nodo al que apunta el hijo izquierdo se convierte en el nodo actual.

En este caso la respuesta es sí, puesto que 6 es más pequeño que 7. En este caso el valor del nodo actual se convierte ahora en 5. De nuevo se repite la etapa 1. Debido a que 5 no es igual a 6, se vuelve a la etapa 2. El valor a buscar en este caso es más grande que el valor del nodo actual, por lo tanto se ejecuta la etapa 3.

3. El nodo al que apunta el hijo derecho se convierte en el nodo actual.

Esto hace que el valor del nodo actual sea 6. Cuando de nuevo se repite la etapa 1, se encuentra el valor a buscar. De esta forma, el procedimiento de búsqueda binaria involucra elegir entre los nodos hijo izquierdo y derecho y descender en el árbol para buscar el valor.

En los casos en los que el valor a buscar no se encuentra en el árbol, las etapas 2 y 3 encontrarán en algún momento un NULL como hijo izquierdo o derecho. Si esto ocurre, la búsqueda termina concluyendo que el valor a buscar no se encuentra en el árbol.

El Programa 8.28 contiene la función `busqueda_binaria()`, que implementa la técnica que se ha descrito anteriormente. Una vez más, se hace uso de la recursividad, puesto que `busqueda_binaria()` se llama a sí misma para recorrer el árbol.

Programa 8.28

```

#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

typedef struct nodo
{
    int dato;
    struct nodo *hijo_izq;
    struct nodo *hijo_der;

```



```

} ARBOL;

int busqueda_binaria(ARBOL *ptr, int elemento);

main()
{
    ARBOL *n1, *n2, *n3, *n4, *n5, *n6, *n7, *n8;
    int valor, encontrado;

    n1 = malloc(sizeof(ARBOL));
    n2 = malloc(sizeof(ARBOL));
    n3 = malloc(sizeof(ARBOL));
    n4 = malloc(sizeof(ARBOL));
    n5 = malloc(sizeof(ARBOL));
    n6 = malloc(sizeof(ARBOL));
    n7 = malloc(sizeof(ARBOL));
    n8 = malloc(sizeof(ARBOL));
    n1->dato = 7;
    n1->hijo_izq = n2;
    n1->hijo_der = n3;
    n2->dato = 5;
    n2->hijo_izq = n4;
    n2->hijo_der = n5;
    n3->dato = 13;
    n3->hijo_izq = n6;
    n3->hijo_der = n7;
    n4->dato = 3;
    n4->hijo_izq = n8;
    n4->hijo_der = NULL;
    n5->dato = 6;
    n5->hijo_izq = NULL;
    n5->hijo_der = NULL;
    n6->dato = 12;
    n6->hijo_izq = NULL;
    n6->hijo_der = NULL;
    n7->dato = 17;
    n7->hijo_izq = NULL;
    n7->hijo_der = NULL;
    n8->dato = 2;
    n8->hijo_izq = NULL;
    n8->hijo_der = NULL;
    printf("Introduzca el valor a buscar => ");
    scanf("%d",&valor);
    encontrado = busqueda_binaria(n1,valor);
    printf("El valor %d ",valor);
    encontrado ? printf(" ha ") : printf(" no ha ");
    printf("encontrado en el arbol bonario.\n");
}

int busqueda_binaria(ARBOL *ptr, int elemento)
{

```

```

    if(ptr == NULL)
        return(FALSE);
    else
        if(ptr->dato == elemento)
            return(TRUE);
        else
            if(elemento < ptr->dato)
                busqueda_binaria(ptr->hijo_izq, elemento);
            else
                busqueda_binaria(ptr->hijo_der, elemento);
}

```

A continuación se muestran dos ejecuciones del Programa 8.28, una con éxito y la otra sin éxito.

```

Introduzca el valor a buscar => 6
El valor 6 ha sido encontrado en el arbol binario

```

```

Introduzca el valor a buscar => 22
El valor 22 no ha sido encontrado en el arbol binario

```

Escribir una función que cree un árbol de búsqueda binario es un reto de programación. Usted conocerá realmente cómo se utiliza un árbol binario después de hacerlo.

Recorrido en profundidad y en anchura

Las técnicas de recorrido en profundidad y en anchura son fundamentalmente diferentes. El **recorrido en profundidad** ya ha sido utilizado, puesto que es el empleado para recorrer el árbol binario del Programa 8.26 (y anteriormente ha sido utilizado en los recorridos en preorden, postorden y orden simétrico). Un recorrido en profundidad desciende hacia el lado izquierdo del árbol binario mientras se pueda. Cuando se encuentra un NULL, la búsqueda continúa en el siguiente hijo derecho situado más abajo.

Un **recorrido en anchura** accede a los nodos del árbol en un orden diferente. Como se muestra en la Figura 8.19, los nodos del árbol son accedidos por niveles, comenzando en el primer nivel (el nodo raíz). A los nodos hijos del nodo raíz (que ocupan el nivel dos) se accede en siguiente lugar. Luego se accede a sus nodos hijos (los cuatro nodos del nivel 3) y así sucesivamente. Esta técnica de recorrido se implementa fácilmente empleando una cola. Sólo se requieren dos etapas para controlar la cola:

1. Extraer un nodo de la cola. Éste se convierte en el nodo actual.
2. Situar todos los nodos hijos del nodo actual en la cola.

La cola se carga inicialmente con el nodo raíz del árbol. Las etapas 1 y 2 se van repitiendo hasta que la cola se vacía. Para el árbol de la Figura 8.19, las colas van tomando los siguientes valores durante el recorrido en anchura:

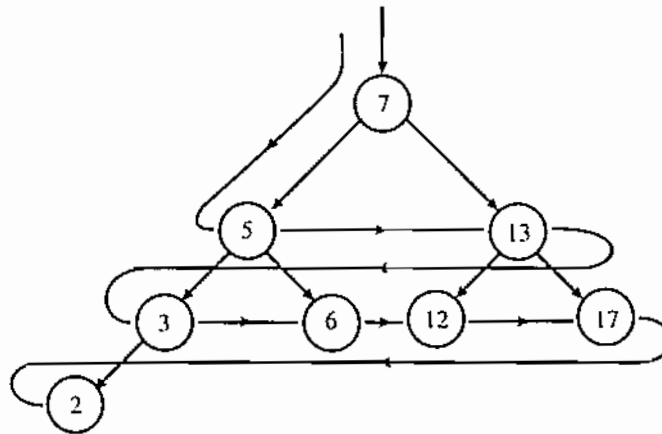


Figura 8.19. Recorrido en anchura de un árbol binario.

```

7          -- nodo inicial
13  5      -- se añaden los hijos del 7
6   3   13 -- se añaden los hijos del 5
17  12  6   3 -- se añaden los hijos del 13
2   17  12  6 -- se añaden los hijos del 3
2   17  12
2   17
2

```

El Programa 8.29 utiliza una cola de punteros a nodos de árbol para implementar el recorrido en anchura. Observe que este programa utiliza dos estructuras diferentes.

Programa 8.29

```

#include <stdio.h>
#include <stdlib.h>

typedef struct tnode
{
    int dato;
    struct nodot *hijo_izq;
    struct nodot *hijo_der;
} ARBOL;

typedef struct nodoq
{
    struct nodot *tptr;
    struct nodoq *enlace;
} COLA;

void bfs(ARBOL *ptr);
void dfs(ARBOL *ptr);
void mostrarCola(COLA *ptr);
int vacia(COLA *ptr);
void encolar(COLA **cabeza, ARBOL *ptr);
void desencolar(COLA **cabeza, ARBOL **ptr);

```

```

main()
{
    ARBOL *n1, *n2, *n3, *n4, *n5, *n6, *n7, *n8;

    n1 = malloc(sizeof(ARBOL));
    n2 = malloc(sizeof(ARBOL));
    n3 = malloc(sizeof(ARBOL));
    n4 = malloc(sizeof(ARBOL));
    n5 = malloc(sizeof(ARBOL));
    n6 = malloc(sizeof(ARBOL));
    n7 = malloc(sizeof(ARBOL));
    n8 = malloc(sizeof(ARBOL));
    n1->dato = 7;
    n1->hijo_izq = n2;
    n1->hijo_der = n3;
    n2->dato = 5;
    n2->hijo_izq = n4;
    n2->hijo_der = n5;
    n3->dato = 13;
    n3->hijo_izq = n6;
    n3->hijo_der = n7;
    n4->dato = 3;
    n4->hijo_izq = n8;
    n4->hijo_der = NULL;
    n5->dato = 6;
    n5->hijo_izq = NULL;
    n5->hijo_der = NULL;
    n6->dato = 12;
    n6->hijo_izq = NULL;
    n6->hijo_der = NULL;
    n7->dato = 17;
    n7->hijo_izq = NULL;
    n7->hijo_der = NULL;
    n8->dato = 2;
    n8->hijo_izq = NULL;
    n8->hijo_der = NULL;
    printf("Recorrido en profundidad => ");
    dfs(n1);
    printf("Recorrido en anchura => ");
    bfs(n1);
}

void dfs(ARBOL *ptr)
{
    if(ptr != NULL)
    {
        printf("%4d", ptr->dato);
        dfs(ptr->hijo_izq);
        dfs(ptr->hijo_der);
    }
}

```

```

    }
void bfs(ARBOL *ptr)
{
    COLA *cabezaq = NULL;
    ARBOL *p;

    encolar(&cabezaq, ptr);
    while(!vacía(cabezaq))
    {
        desencolar(&cabezaq, &p);
        printf("%4d", p->dato);
        if(p->hijo_izq != NULL)
            encolar(&cabezaq, p->hijo_izq);
        if(p->hijo_der != NULL)
            encolar(&cabezaq, p->hijo_der);
    }
}

int vacía(COLA *ptr)
{
    if(ptr == NULL)
        return(1);
    else
        return(0);
}

void encolar(COLA **cabeza, ARBOL *ptr)
{
    COLA *p;

    p = malloc(sizeof(COLA));
    if(p != NULL)
    {
        p->tptr = ptr;
        p->enlace = *cabeza;
        *cabeza = p;
    }
}

void desencolar(COLA **cabeza, ARBOL **ptr)
{
    COLA *p1, *p2;

    p1 = *cabeza;
    if(vacía(p1))
    {
        printf("Error! La cola esta vacía.\n");
        *ptr = NULL;
    }
    else
    {
        p2 = *cabeza;

```

```

        while(p2->enlace != NULL)
        {
            p1 = p2;
            p2 = p2->enlace;
        }
        *ptr = p2->tptr;
        p1->enlace = NULL;
        free(p2);
        if(p1 == p2)
            *cabeza = NULL;
    }
}

```

Las estructuras utilizadas son ARBOL (nodo árbol) y COLA (nodo cola). Ambas estructuras se utilizan en las funciones que manejan las colas. Las funciones `dfs()` y `bfs()` implementan los recorridos en profundidad y en anchura respectivamente. La ejecución del programa muestra los diferentes resultados de estos recorridos:

```

Recorrido en profundidad => 7 5 3 2 6 13 12 17
Recorrido en anchura      => 7 5 13 3 6 12 17 2

```

Cada tipo de recorrido tiene sus ventajas y sus inconvenientes. Para ilustrar esto, considere un programa para jugar al ajedrez. Después de unos cuantos movimientos, el programa hará uso de un gran árbol de decisión para evaluar el siguiente movimiento a realizar. Un recorrido en profundidad puede gastar demasiado tiempo examinando los nodos izquierdo del árbol de búsqueda antes de tomar una elección más deseable en la parte derecha. Un recorrido en anchura normalmente llegará al mismo nodo de forma más rápida.

Por otro lado, un programa diseñado para empaquetar objetos en una caja podría llegar a una solución más adecuada utilizando un recorrido en profundidad sobre el árbol de decisión, mientras que un recorrido en anchura consumiría más tiempo considerando muchas soluciones parciales. Es muy práctico conocer cuando emplear una técnica particular para recorrer un árbol.

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido para permitirle adquirir una valiosa experiencia y una realimentación inmediata de lo que hacen los conceptos y mandatos presentados en este capítulo. Además son divertidos.

Ejercicios

1. El Programa 8.30 ilustra el uso de tipos enumerados en C con constantes asignados a los identificadores ¿Cuál cree que será la salida del programa?

Programa 8.30 #include <stdio.h>

```
enum numeros {Uno = 1, Dos = 2, Tres = 3}

main()
{
    int resultado;

    resultado = Uno + Dos;
    printf("Resultado = %d", resultado);
}
```

-
2. Observe cómo enuncia el Programa 8.31 una lista de declaración de miembros ¿Funcionará esto? Inténtelo
-

Programa 8.31 #include <stdio.h>

```
main()
{
    struct {int numerol, numero2, numero3;} valor;

    valor.numerol = 1;
    valor.numero2 = 2;
    valor.numero3 = 3;

    printf("El valor de numerol es %d", valor.numerol);
}
```

-
3. El Programa 8.32 muestra como se puede emplear en C la sentencia define para sustituir a un identificador largo. Éste es un programa interesante de ejecutar.
-

Programa 8.32 #include <stdio.h>

```
#define S identificador_largo

main()
{
    struct {int numerol, numero2, numero3;} identificador_largo;

    S.numerol = 1;
    S.numero2 = 2;
    S.numero3 = 3;

    printf("El valor de numerol es %d", S.numerol);
}
```

-
4. El Programa 8.33 ilustra la utilización de una etiqueta de estructura. Vea cuál es su salida ¿Es lo que usted esperaba?

Programa 8.33 #include <stdio.h>

```
struct etiqueta
{
    int primero;
    int segundo;
};

main()
{
    struct etiqueta estructura;
    estructura.primero = 1;
    printf("El valor de estructura.primero es %d", estructura.primero);
}
```

5. El Programa 8.34 ilustra la utilización de una union en C. Recuerde que una union permite utilizar la misma posición de memoria para variables de tipos diferentes. Prediga el resultado de este programa y luego ejecútelo para comprobar su resultado.

Programa 8.34 #include <stdio.h>

```
typedef union
{
    int primero;
    float segundo;
} valor;

main()
{
    valor numerol;
    numerol.primero = 12;
    printf("El valor de numerol.primero es %d\n", numerol.primero );
    numerol.segundo = 34.5;
    printf("El valor de numerol.segundo es %f\n", numerol.segundo);
}
```

6. El Programa 8.35 emplea un vector de estructuras, cuyo miembro es otra estructura. Revise el programa para ver si su funcionamiento es el que usted espera.

Programa 8.35 #include <stdio.h>

```
typedef struct
{
    int primero;
}estructural;

typedef struct
{
```



```

        estructural segundo;
    } estructurar2;

typedef estructurar2 vector_de_estructuras[5];

main()
{
    vector_de_estructuras valores ;

    valores[1].segundo.primerio = 54;
    printf("El valor es %d", valores[1].segundo.primerio);
}

```

7. El Programa 8.36 crea una lista enlazada compuesta por tres nodos. Prediga el resultado del programa.
-

Programa 8.36

```

#include <stdio.h>
#include <stdlib.h>

typedef struct nodo
{
    char dato;
    struct nodo *enlace;
} LISTA;

void mostrar_lista(LISTA *ptr);

main()
{
    LISTA *n1, *n2, *n3;

    n1 = malloc(sizeof(LISTA));
    n2 = malloc(sizeof(LISTA));
    n3 = malloc(sizeof(LISTA));

    n1->dato = 'c';
    n1->enlace = n2;
    n2->dato = 'á';
    n2->enlace = n3;
    n3->dato = 't';
    n3->enlace = n2;
    printf("La lista enlazada es como sigue: ");
    mostrar_lista(n1);
}

void mostrar_lista(LISTA *ptr)
{
    while (ptr != NULL)
    {

```

```

        printf("%c", ptr->dato);
        ptr = ptr->enlace;
    }
}

```

Autoevaluación

DIRECTRICES

Responda a las siguientes preguntas referidas al Programa 8.25.

Preguntas

1. ¿Cómo se define el conjunto de instrucciones?
2. ¿Es significativo el orden en el que se definen las instrucciones en la sentencia `enum`?
3. ¿Por qué es necesario definir el acumulador como una variable global?
4. ¿Qué hace `exec(50)`?
5. ¿Qué le ocurre a la variable `iop` cuando se ejecuta `PRINT`, `RET` o `STOP`?
6. ¿Cómo se pueden añadir las siguientes instrucciones a la máquina MiniMicro: `AND`, `OR`, `XOR`? Cada instrucción opera sobre el acumulador y un operando que se proporciona.
7. ¿Cómo se puede añadir un segundo acumulador al emulador?
8. Explique el funcionamiento de la instrucción `JNZ`.
9. ¿Qué hace el siguiente programa?

```

0: LOAD 10
2: PRINT
5: SUB 1
3: JNZ 2
7: STOP

```

Problemas de fin de capítulo

Conceptos generales

Sección 8.1

1. ¿Qué tipo de datos se utilizan en C para describir un conjunto discreto de valores enteros?
2. ¿Cuál es el valor entero del primer tipo de datos enumerado que se declara en C?
3. Explique razonadamente si `enum` en C crea un nuevo tipo de datos.
4. Muestre cómo se puede asignar un valor entero a un tipo de datos `enum`.

Sección 8.2

5. Explique razonadamente si `typedef` crea un nuevo tipo de datos.

6. Indique el propósito de `typedef` en C.
7. ¿Cuál es el nombre del nuevo tipo que resulta del siguiente código:

```

typedef struct
{
    int valor;
} estructura;

```

Sección 8.3

8. Defina una estructura en C.
9. Dé un ejemplo de sistema de uso común que utilice el concepto de una estructura de C.
10. Ilustre la sintaxis de una estructura de C tal y como se presentó en esta sección.

11. Explique qué hace el operador de miembro de C. ¿Qué símbolo se utiliza para este operador en una estructura?

Sección 8.4

12. ¿Cuál es el identificador que da nombre al tipo de estructura definido en la lista de declaración de miembros de la estructura.
13. Dé un ejemplo del uso de una etiqueta de estructura.
14. ¿Cuál es el propósito de `->` cuando se aplica a las estructuras de C?
15. Indique las tres operaciones permitidas sobre estructuras.

Sección 8.5

16. Indique la forma de almacenar uno o más tipos de datos diferentes en la misma posición de memoria.
17. Indique la diferencia entre declarar una estructura en C y una union en C.
18. ¿Se puede asignar valor inicial a una estructura en C?
19. Explique qué representa la siguiente línea de código.
`variable[2].numero`

Sección 8.6

20. Explique razonadamente si una estructura en C puede contener un miembro que sea otra estructura.
21. Explique si un vector de estructuras puede contener un vector como uno de sus miembros.
22. Explique si un vector de estructuras puede ser miembro de otro vector de estructuras.

Sección 8.7

23. ¿Qué determina el tamaño de un nodo?
24. ¿Cómo son los enlaces en una lista enlazada?
25. ¿Cómo se insertan y borran nodos en los extremos de una lista enlazada?
26. Explique por qué las pilas y las colas son simplemente listas enlazadas accedidas de modo especial.
27. ¿Cómo se utiliza la recursividad para acceder a un nodo en un árbol binario?

Diseño de programas

Cuando escriba los siguientes programas en C, utilice las nuevas estructuras de datos que se han tratado en este capítulo. Se sugiere el uso de punteros y de recursividad para aquellas aplicaciones donde su uso sea el más adecuado.

28. Escriba un programa en C que muestre todas las combinaciones de tipos enumerados (enum) ROJO, VERDE y AZUL.
29. Escriba un programa en C que lea 10 números reales del usuario, los almacene en una lista enlazada y luego busque en la lista el número mayor para imprimirlo.
30. Escriba un programa que inserte un número entero en la posición correcta de una lista enlazada ordenada. Utilice la siguiente lista en su programa:

6 13 29 32 45

31. Escriba un programa en C que borre un nodo del medio de una lista enlazada.
32. Escriba un programa en C que utilice una pila para determinar si una cadena de caracteres es un palíndromo.
33. Escriba un programa en C que utilice una pila de enteros para evaluar la siguiente expresión en orden postfijo:

5 4 3 + 7 *

34. Escriba un programa que cree un árbol binario para representar una expresión infija introducida como entrada. Por ejemplo, $(A + B) * C$ debería producir el árbol que se muestra en la Figura 8.15.
35. Diseñe un programa que utilice una estructura para mantener la siguiente información referida al estado de diseño de un proyecto: número identificativo del proyecto, nombre del proyecto, nombre del cliente, fecha de finalización, y si el proyecto se ha finalizado o no.

Entrada/salida de disco

Objetivos

En este capítulo podrá aprender:

1. Cómo crear un archivo en el disco.
2. Cómo escribir datos en un archivo.
3. Cómo leer datos de un archivo.
4. Cómo realizar entrada/salida con registros.
5. Cómo suministrar información al programa desde la línea de órdenes (mandatos).
6. La diferencia entre los archivos de acceso secuencial y los de acceso aleatorio.

Palabras clave

Puntero a archivo	Archivo estándar
Flujo de caracteres	E/S estándar
Archivo de texto	E/S del sistema
Modo binario	Archivo de acceso secuencial
Modo texto	Archivo de acceso aleatorio
Archivo binario	Argumentos de la línea de órdenes
Memoria intermedia	Volcado de la memoria intermedia

Contenido

9.1. Entrada/salida de disco	9.4. Programa de aplicación: Una base de datos de piezas
9.2. Más sobre la E/S de disco	
9.3. Flujos de datos, punteros a archivos y argumentos de la línea de mandatos	9.5. Programas de aplicación adicionales

Introducción

En este capítulo se añadirá una nueva dimensión a la programación de aplicaciones utilizando el disco del computador. Se examinarán los mecanismos de creación y acceso a archivos, junto con los métodos para estructurar los datos en los mismos. Se verá también cómo los argumentos de la línea de mandatos pueden usarse como parámetros de entrada de los programas.

9.1. ENTRADA/SALIDA DE DISCO

Presentación

Los programas que se han desarrollado hasta este momento no permitían al usuario del programa salvar ninguna información antes de apagar el computador. Se trata de una importante restricción para los diferentes tipos de programas tecnológicos. En esta sección, se descubrirá cómo un programa puede salvar datos en el disco y cómo posteriormente recuperarlos del mismo.

Crear un archivo en el disco

Para almacenar los datos de un usuario sobre el disco del computador, se debe crear un archivo para guardarlos en él. A la hora de crear el archivo, se deben respetar las normas que con respecto a los nombres de los archivos imponga el sistema operativo que esté ejecutando en el computador. Los nombres de archivo que aparecen en los programas de este capítulo se han seleccionado de forma que sean válidos para la mayoría de los sistemas operativos. Concretamente, se usan nombres aceptados por DOS (*Disk Operating System*) y UNIX. De cualquier forma, dado que DOS presenta un esquema de nombres más restrictivo, se mostrará seguidamente un resumen de cómo deben de ser los nombres de archivos en dicho sistema.

Repaso del DOS

Los nombres de archivos en DOS tienen el formato siguiente:

[Dispositivo:]Archivo[.EXT]

Donde

Dispositivo = El nombre del dispositivo que se corresponde con el disco al que se quiere acceder.

Archivo = El nombre del archivo (no más de ocho caracteres).

.EXT = El nombre del archivo tiene una extensión opcional (no más de tres caracteres).

Tanto el dispositivo como la extensión son opcionales. Si no se especifica el dispositivo, se usará el dispositivo activo en ese momento. El nombre del dispositivo siempre irá seguido por dos puntos (:). Por ejemplo:

B:ARCHIVO.DAT

es el nombre de un archivo sobre el dispositivo B: con el nombre ARCHIVO y la extensión .DAT. No hay diferencia si se introduce el nombre con letras mayúsculas o minúsculas, el sistema operativo DOS siempre mostrará todos los archivos del disco con mayúsculas.

Programa que crea un archivo y almacena datos en él

El Programa 9.1 crea un archivo llamado ARCHIVO.DAT sobre el dispositivo activo y escribe en él el carácter C. Cuando se ejecuta el programa, el nuevo archivo ARCHIVO.DAT aparecerá en el dispositivo activo.

Programa 9.1

```
#include <stdio.h>
main()
{
    FILE *puntero_a_archivo; /* Esto es el puntero a archivo. */

    /* Crea un archivo llamado ARCHIVO.DAT y asigna su dirección a
       puntero_a_archivo */
    puntero_a_archivo = fopen("ARCHIVO.DAT", "w");

    /* Escribe una letra en el archivo abierto */
    putc('C', puntero_a_archivo);

    /* Cierra el archivo creado. */
    fclose(puntero_a_archivo);
}
```

Análisis del programa

El Programa 9.1 comienza declarando un puntero a archivo que es un tipo de datos llamado FILE:

```
FILE *puntero_a_archivo; /* Esto es el puntero a archivo. */
```

Éste es un tipo de datos definido en la mayoría de las versiones de C.

A continuación, se asigna al puntero a archivo el valor devuelto por la función `fopen()`. Para usar esta función, se deben especificar como argumentos el nombre del archivo y una cadena de caracteres que indica el modo de apertura del archivo (*w*

significa abierto para escribir y debe aparecer entre comillas dobles por tratarse de una cadena de caracteres).

```
puntero_a_archivo = fopen("ARCHIVO.DAT", "w");
```

Esto creará realmente un archivo llamado ARCHIVO.DAT sobre el dispositivo activo y lo abrirá para escribir en él.

Seguidamente se escribe la letra C dentro del archivo abierto empleando la función `putc()`. Esta función escribe un único carácter dentro de un archivo abierto. Requiere dos argumentos: el carácter que se quiere escribir y el puntero a archivo.

```
putc('C', puntero_a_archivo);
```

La siguiente sentencia es necesaria cuando se terminen de utilizar los archivos abiertos. Si no se utiliza, se pueden perder los datos escritos en los mismos.

```
fclose(puntero_a_archivo);
```

Se trata de la función `fclose()` cuyo argumento es un puntero a archivo.

Almacenamiento de una cadena de caracteres

El programa previo almacenaba sólo un carácter en un archivo. El Programa 9.2 enseña un método para almacenar (guardar) una cadena de caracteres en un archivo. En este caso, el usuario introducirá una cadena de caracteres desde el teclado y ésta se almacenará en un archivo llamado ARCHIVO.DAT. El final de la cadena de caracteres quedará determinado por la introducción de un carácter retorno de carro.

Programa 9.2

```
#include <stdio.h>
main()
{
    FILE *puntero_a_archivo; /* Esto es el puntero a archivo. */
    char caracter;           /* Carácter a escribir en el archivo. */

    /* Crea un archivo llamado ARCHIVO.DAT y asigna su dirección
       a puntero_a_archivo */
    puntero_a_archivo = fopen("ARCHIVO.DAT", "w");

    /* Escribe los datos leídos del teclado en el archivo */
    while((caracter = getchar()) != '\r')
        caracter = putc(caracter, puntero_a_archivo);

    /* Cierra el archivo creado. */
    fclose(puntero_a_archivo);
}
```

Como el programa anterior, este programa utiliza la función `fopen()` para abrir el archivo, la función `putc()` para escribir un único carácter cada vez y la función `fclose()` para cerrar el archivo.

Nótese que `putc()` está incluida en el siguiente bucle `while`:

```
while((character = getchar()) != '\r')
    character = putc(character, puntero_a_archivo);
```

Este bucle continuará mientras que el usuario no presione la tecla de retorno de carro.

Lectura de datos de un archivo

El Programa 9.3 muestra cómo leer todos los caracteres de un archivo existente. A esta secuencia de caracteres se le denomina flujo de datos de caracteres. Un **flujo de datos de caracteres** es una secuencia de octetos que se transfiere de un sitio a otro (por ejemplo, desde la memoria del computador al disco). El programa leerá del flujo de datos asociado al archivo hasta que encuentre una marca de fin de archivo (EOF) que indique que no hay más datos disponibles.

Programa 9.3

```
#include <stdio.h>

main()
{
    FILE *puntero_a_archivo; /* Esto es el puntero a archivo. */
    char character;          /* Carácter a leer del archivo. */

    /* Abre el archivo existente llamado ARCHIVO.DAT y asigna su
       dirección a puntero_a_archivo */
    puntero_a_archivo = fopen("ARCHIVO.DAT", "r");

    /* Lee caracteres del archivo abierto y los muestra por
       pantalla */
    while((character = getc(puntero_a_archivo)) != EOF)
        printf("%c", character);

    /* Cierra el archivo creado. */
    fclose(puntero_a_archivo);
}
```

Análisis del programa

Este programa que lee de un archivo existente es muy similar al programa anterior que creaba un archivo y escribía sobre él. Como en el caso anterior, se declara un puntero a archivo y una variable para almacenar el carácter que se va leyendo:

```
FILE *puntero_a_archivo; /* Esto es el puntero a archivo. */
char character;          /* Carácter a leer del archivo. */
```

Para abrir el archivo se utilizará como en el programa anterior la función `fopen()`, pero especificando la cadena de caracteres `"r"` para indicar que se abre para lectura.

```
puntero_a_archivo = fopen("ARCHIVO.DAT","r");
```

Se usa la función `getc()` para leer un carácter del archivo. El único argumento de esta función es un puntero a archivo devolviendo como resultado el carácter leído. El bucle que se observa debajo se repetirá hasta que se lea la marca de final de archivo. Cada carácter leído se va imprimiendo por la pantalla usando la función `printf()`.

```
while((caracter = getc(puntero_a_archivo)) != EOF)
    printf("%c", caracter);
```

El último paso importante es cerrar el archivo cuando se deje de utilizar:

```
fclose(puntero_a_archivo);
```

Conclusión

En esta sección se ha presentado el concepto de creación de un archivo desde un programa. Asimismo, se ha mostrado cómo salvar caracteres en un archivo y recuperarlos del mismo. Ahora compruebe si ha asimilado correctamente esta sección mediante el siguiente repaso.

Repaso de la Sección 9.1

1. Describa cómo es el formato del nombre de un archivo en DOS.
2. Explique qué es un puntero a archivo. Ponga un ejemplo.
3. Nombre cuál es la función para abrir un archivo en C.
4. Especifique qué sentencia se usa en C para abrir un archivo.
5. ¿Qué diferencia hay entre cómo se abre un archivo para leer y cómo se abre para escribir?
6. ¿Qué se debe de hacer cuando se ha terminado de utilizar un archivo abierto? Ponga un ejemplo.

9.2. MÁS SOBRE LA E/S DE DISCO

Presentación

En la sección previa se introdujeron las operaciones básicas de entrada/salida sobre archivos. En esta sección se van a presentar operaciones de carácter más avanzado que permitirán almacenar y recuperar tipos de datos complejos como vectores y estructuras.

Las alternativas

La Tabla 9.1 muestra las condiciones que se pueden encontrar cuando se trabaja con datos en un disco. Como se puede ver en la tabla, hay cuatro posibilidades que se explicarán a lo largo de esta sección.

Además de las posibilidades presentadas en la tabla, en el lenguaje C hay cuatro maneras diferentes de leer y escribir un dato, tal como se muestra en la Tabla 9.2.

Tabla 9.1. Condiciones de archivos de disco

Condición	Significado
1	El archivo de disco no existe y se quiere crearlo y añadir información.
2	El archivo de disco ya existe y se quiere extraer información de él.
3	El archivo de disco ya existe y se quiere añadir más información conservando la información que ya estaba en él.
4	El archivo de disco ya existe y se quiere destruir la información que ya existía en él y añadir información nueva.

Tabla 9.2. Distintos métodos de lectura y escritura de datos

Método	Comentarios
Un carácter cada vez.	Lee y escribe a disco un carácter cada vez.
Leer y escribir datos y cadenas de caracteres	Lee y escribe a disco una cadena de caracteres.
Método mixto	Usado para E/S de caracteres, cadenas de caracteres, reales y enteros.
Método de estructuras o bloques	Usado para E/S de vectores de elementos y estructuras.

Ya se han utilizado los dos primeros métodos especificados en la tabla: almacenar y recuperar un carácter o una cadena de caracteres. Los programas de esta última sección contienen los dos últimos métodos: mezcla de tipos de datos diferentes y operaciones con estructuras o bloques.

Formato del archivo

En la Sección 9.1 se pudo observar que un programa que realiza operaciones de E/S sobre archivos tiene una estructura característica que se muestra en la Figura 9.1.

Observando dicha figura, se puede ver que el programa utiliza el tipo `FILE` para definir un puntero a archivo. Este tipo tiene una estructura predefinida declarada en el archivo de cabecera `<stdio.h>`. Este archivo debe incluirse en todos los programas que contienen operaciones de E/S. La estructura predefinida `FILE` ayuda a establecer el enlace necesario entre el programa y el sistema operativo.

El próximo paso en el programa es abrir el archivo. Como muestra la Figura 9.1, se usa la función `fopen("NOMBRE", "modo")`. El primer argumento "NOMBRE" debe ser un nombre de archivo válido. (Nótese que el nombre del archivo podría incluir el nombre del camino al archivo, por ejemplo `\\MIDIR\\ARCHIVO.DAT`).

El segundo argumento "modo" es una cadena de un solo carácter que especifica el modo de apertura del archivo. Se han visto anteriormente los modos "r" y "w". En la Tabla 9.3 se muestran otros más.

```

<stdio.h>
.
.
FILE *ptr
.
.
ptr fopen("NOMBREDOS", "orden");
.
.
/*INTERACCIÓN CON EL ARCHIVO*/
.
.
fclose (ptr)
.
.

```

Figura 9.1. Estructura estándar de la E/S a disco en C.

Tabla 9.3. Calificación de operaciones sobre archivos en C

Calificador	Significado
"a"	Abierto para añadir datos. Los datos nuevos se añaden al final del archivo o se crea un archivo nuevo (si no existe).
"r"	Abierto para lectura. El archivo debe existir.
"w"	Abierto para escritura. Los datos son escritos desde el principio o se crea un nuevo archivo (si no existe).
"a+"	Abierto para lectura y añadir datos. Si el archivo no existe se crea.
"r+"	Abierto para lectura y escritura. El archivo debe existir.
"w+"	Abierto para lectura y escritura. Se escribe desde el principio del archivo.

Programa de ejemplo de archivos

En el Programa 9.4 se muestran varios aspectos importantes sobre los archivos. Este programa permite usar una de las opciones siguientes para crear y leer un archivo de mensajes:

1. Crear un archivo nuevo con un nombre suministrado por el usuario.
2. Crear un archivo nuevo con un nombre suministrado por el usuario y escribir un mensaje en él.
3. Leer un mensaje de un archivo existente.
4. Añadir un mensaje al final de un archivo existente.
5. Informar al usuario de que un archivo en el que quiere leer o añadir algo no existe.

Programa 9.4

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    char seleccion[2];          /* Selección para el archivo. */
    char nombre_archivo[13];    /* Nombre del archivo. */
    char eleccion[2];           /* Elección. */
    int valor_seleccion;        /* Número de selección. */
    int caracter;               /* Carácter. */
    FILE *puntero_a_archivo;    /* Puntero a archivo. */

    /* Mostrar las opciones al usuario. */
    printf("Seleccione una de las siguientes opciones:\n");
    printf("1] Crear un nuevo archivo. \n");
    printf("2] Escribir datos sobre un archivo existente.\n");
    printf("3] Añadir nuevos datos a un archivo existente.\n");
    printf("4] Leer datos de un archivo existente.\n");

    /* Leer la opción del usuario y llevar a cabo la acción. */
    do
    {
        printf("Su seleccion => ");
        gets(eleccion);
        valor_seleccion = atoi(eleccion);
        switch(valor_seleccion)
        {
            case 1 : /* Crear un nuevo archivo. */
            case 2 : /* Escribir sobre archivo existente */
                strcpy(seleccion, "w");
                break;

            case 3 : /* Añadir datos a archivo existente */
                strcpy(seleccion, "a");
                break;

            case 4 : /* Leer datos de un archivo existente */
                strcpy(seleccion, "r");
                break;

            default :
                printf("Esto no es una eleccion.\n");
                valor_seleccion = 0;
        }
    } while(valor_seleccion == 0);

    /* Leer el archivo del usuario. */
    printf("Introduzca el nombre del archivo => ");
    gets(nombre_archivo);

```

```

/* Abrir el archivo según la opción. */
if ((puntero_a_archivo = fopen(nombre_archivo, seleccion)) == NULL)
{
    printf("No puedo abrir el archivo %s!", nombre_archivo);
    exit(-1);
}
/* Escribir o leer del archivo. */
switch(valor_seleccion)
{
    case 1 : break;
    case 2 :
    case 3 :
        printf("Introduzca los caracteres a grabar: \n");
        while((caracter = getche()) != '\r')
            caracter = putc(caracter, puntero_a_archivo);

        break;
    case 4 :{
        while((caracter = getc(puntero_a_archivo)) != EOF)
            printf("%c", caracter);
        break;
    }
}
/* Cerrar el archivo abierto. */
fclose(puntero_a_archivo);
}

```

La ejecución del Programa 9.4 produce:

Seleccione una de las siguientes opciones:

- 1) Crear un nuevo archivo.
- 2) Escribir datos sobre un archivo existente.
- 3) Añadir nuevos datos a un archivo existente.
- 4) Leer datos de un archivo existente.

Su eleccion => 2

Introduzca el nombre del archivo => ARCHIVO.01

Introduzca los caracteres a grabar: Grabado por un programa C.

El mensaje anterior puede recuperarse posteriormente, o se pueden añadir nuevos mensajes al archivo o escribir sobre los ya existentes. El programa informará al usuario si el archivo no existe, siendo necesario que éste cree un archivo nuevo antes de poder usarlo.

Análisis del programa

El puntero a archivo se declara en la parte declarativa del programa:

```
FILE *puntero_a_archivo; /* Puntero a archivo. */
```

Las opciones del usuario se muestran por la pantalla usando la función `printf()`:

```
printf("Seleccione una de las siguientes opciones:\n");
printf("1] Crear un nuevo archivo. \n");
printf("2] Escribir datos sobre un archivo existente.\n");
printf("3] Añadir nuevos datos a un archivo existente.\n");
printf("4] Leer datos de un archivo existente.\n");
```

A continuación, el programa pide al usuario que introduzca su elección. Esto se hace dentro de un bucle `do` que hace que se repita la petición si el usuario no introdujo una opción válida:

```
do
{
    printf("Su seleccion => ");
```

Se usará la función `gets()` para leer la entrada del usuario, en lugar de utilizar la función `scanf()` que es más complicada de emplear. La entrada del usuario se convierte al tipo `int` mediante la función `atoi()` que convierte una cadena de caracteres al tipo entero.

```
gets(eleccion);
valor_seleccion = atoi(eleccion);
```

Seguidamente se usa una sentencia `switch` para seleccionar una de las cinco alternativas posibles. Observe que las dos primeras alternativas tienen el mismo tratamiento:

```
switch(valor_seleccion)
{
    case 1 :
    case 2 : strcpy(seleccion, "w");
              break;
    case 3 : strcpy(seleccion, "a");
              break;
    case 4 : strcpy(seleccion, "r");
              break;
    default :
              printf("Esto no es una eleccion.\n");
              valor_seleccion = 0;
```

La condición `default` se activará si no se selecciona ninguno de los cuatro casos. En ese caso se le comunicará al usuario que no ha seleccionado ninguna de las opciones válidas y la variable `valor_seleccion` se pone a cero, lo que causa la repetición del bucle y permite al usuario elegir una nueva opción.

```
while(valor_seleccion == 0);
```

Suponiendo que el usuario seleccione una opción válida, el programa le pedirá el nombre del archivo. Se utiliza de nuevo la función `gets()` para leer dicho nombre.

```
printf("Introduzca el nombre del archivo => ");
gets(nombre_archivo);
```

En la sentencia posterior se usa la función `fopen()` para abrir el archivo. La variable `nombre_archivo` contiene el nombre del archivo introducido por el usuario y la variable `seleccion` contendrá uno de los modos de apertura de un archivo (`r`, `w` o `a`).

```
if ((puntero_a_archivo = fopen(nombre_archivo, seleccion)) == NULL)
```

Si por cualquier razón el archivo no se puede abrir, `fopen()` devolverá un valor nulo (0) y entonces se ejecutará la siguiente sentencia compuesta:

```
{
    printf("No puedo abrir el archivo %s!", nombre_archivo);
    exit(-1);
}
```

En caso contrario, esta sentencia no se ejecutará.

A continuación, se usa un `switch` para determinar qué acción se llevará a cabo con el archivo abierto. Con la opción uno, el usuario elige simplemente crear un nuevo archivo, por lo tanto, no hay nada que leer o escribir:

```
switch(valor_seleccion)
{
    case 1 : break;
```

Con la opción dos o tres, el usuario selecciona almacenar una cadena de caracteres en el archivo (escribiendo sobre una cadena existente o añadiendo una nueva). Se usa un bucle `while` para leer cada carácter de la cadena y escribirlo usando la función `putc()`, que escribe un carácter en el archivo correspondiente a `puntero_a_archivo`. El bucle `while` permanece activo hasta que el programa de usuario presione la tecla RETURN/ENTER, produciendo un retorno de carro (`\r`).

```
case 2 :
case 3 :
    printf("Introduzca los caracteres a grabar: \n");
    while((caracter = getche()) != '\r')
        caracter = putc(caracter, puntero_a_archivo);
    break;
```

En el cuarto caso, el usuario elige leer una cadena de caracteres desde un archivo seleccionado. Se utiliza la función `getc()` para obtener un carácter del archivo correspondiente a `puntero_a_archivo`. Este proceso está en un bucle `while` que continuará hasta que se alcance la marca de EOF. La función `printf()` se usa para imprimir en la pantalla los caracteres leídos del archivo.


```

case 4 :
    while((character = getc(puntero_a_archivo)) != EOF)
        printf("%c", character);
    break;

```

El programa termina realizando el cierre del archivo.

```
fclose(puntero_a_archivo);
```

Mezcla de tipos de datos

Los programas previos se limitan a trabajar con cadenas de caracteres. El Programa 9.5 ilustra un método de trabajo que permite tratar tanto datos de tipo numérico como cadenas de caracteres. El programa muestra cómo se pueden almacenar en un archivo datos de diferentes tipos usando como ejemplo un archivo de piezas basado en la estructura definida en el capítulo anterior. Para ello se utilizará una nueva función denominada `fprintf()`.

Programa 9.5

```

#include <stdio.h>

main()
{
    char nombre_pieza[15];    /* Tipo de pieza. */
    int cantidad;            /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
    FILE *puntero_a_archivo;  /* Puntero a archivo. */

    /* Abrir un archivo para escritura. */
    puntero_a_archivo = fopen("PIEZAS.DAT", "w");

    /* Leer datos del usuario */
    printf("Introduzca el tipo de pieza, la cantidad y el precio");
    printf(" unitario, separado por blanco:\n");
    printf("Pulse -RETURN- para terminar la introduccion de datos.\n");

    scanf("%s %d %f", nombre_pieza, &cantidad, &precio_unitario);
    fprintf(puntero_a_archivo, "%s %d %f", nombre_pieza, cantidad,
            precio_unitario);

    /* Cerrar el archivo abierto. */
    fclose(puntero_a_archivo);
}

```

La ejecución del Programa 9.5 genera la siguiente salida:

```
Introduzca el tipo de pieza, la cantidad y el precio unitario,
separado por blancos:
Pulse -RETURN- para terminar la introduccion de datos.
Resistencia 12 0.05.
```

Suponiendo que el usuario siga cuidadosamente las instrucciones, la función `scanf()` recibirá los datos pedidos y la función `fprintf()` los almacenará en el archivo abierto.

La función `fprintf()` tiene esta forma:

```
fprintf(FILE *flujo, const char *formato, .argumentos)
```

Análisis del programa

El Programa 9.5 tiene varios puntos débiles. No protege la entrada del usuario (se usa la función `scanf()`). Tampoco deja que el usuario conozca si hay algún problema en la apertura del archivo. Sin embargo, muestra un ejemplo simple de programa que escribe datos de diferentes tipos en un archivo. La clave del programa es la función `fprintf()`, la cual permite formatear de muy diversas maneras los datos que se escriben en el disco.

Lectura de datos de tipos diferentes

En el Programa 9.6 se mostrará cómo leer datos de tipos diferentes de un archivo y escribirlos por la pantalla. La clave para este programa es la función `fscanf()`. Esta función es similar a `scanf()` excepto que tiene como primer argumento un puntero a archivo.

Programa 9.6 `#include <stdio.h>`

```
main()
{
    char nombre_pieza[15];    /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
    FILE *puntero_a_archivo;  /* Puntero a archivo. */

    /* Abrir un archivo para lectura. */
    puntero_a_archivo = fopen("PIEZAS.DAT", "r");

    /* Leer datos del archivo. */
    while(fscanf(puntero_a_archivo, "%s %d %f", nombre_pieza,
                &cantidad, &precio_unitario) != EOF)
```

```

printf("%s %d %f\n", nombre_pieza, cantidad, precio_unitario);

/* Cerrar el archivo abierto. */
fclose(puntero_a_archivo);
}

```

Suponiendo que el contenido del archivo de datos de entrada se corresponde con los datos generados por el programa previo, la ejecución del Programa 9.6 producirá el siguiente resultado:

```
Resistencia 12 0.050000
```

Archivos de texto frente a binarios

Todos los archivos que se han utilizado hasta ahora se denominan **archivos de texto**. La Figura 9.2 muestra cómo se almacena un archivo de texto en el disco.

Como se muestra en la figura, los números se almacenan como cadenas de caracteres en lugar de como valores numéricos. Debido a esto, no se usa eficientemente el espacio del disco. Una forma de aumentar la eficiencia en el almacenamiento es usar el **modo binario** en vez del **modo texto**. El **archivo binario** no almacena los números como cadenas de caracteres (como se hace en los archivos de texto). En vez de esto, se almacenan de la misma forma que en la memoria (dos octetos para un entero, cuatro para un número real, etc.). La única restricción es que si un archivo se almacena en modo binario, se tiene que leer en el mismo modo ya que, en caso contrario, carecerá de sentido lo leído. Para ello, lo único que se necesita es añadir la letra **b** al modo de

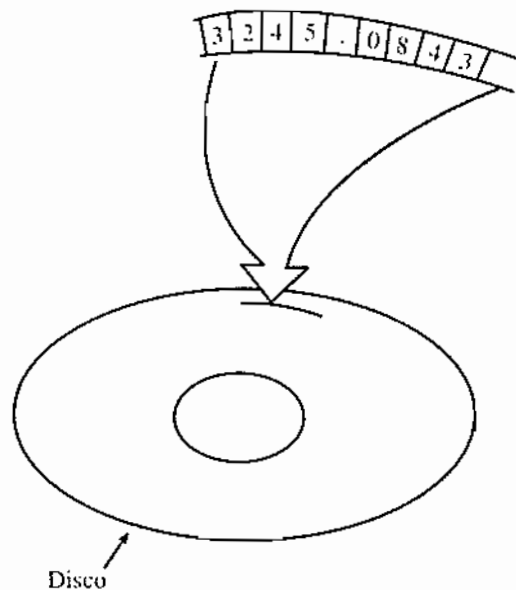


Figura 9.2. Almacenamiento de número en forma de texto.

apertura. Así, `fopen("archivo.01", "wb")` significa abrir, o crear, un archivo llamado `archivo.01` para escribir en formato binario. De la misma forma, `fopen("archivo.01", "rb")` significa abrir un archivo llamado `archivo.01` para leer en formato binario. Como se habrá imaginado el lector, `fopen("archivo.01", "ab")` significa abrir un archivo llamado `archivo.01` para añadir, en formato binario, al final del mismo.

Para trabajar con archivos de texto se deberá especificar la letra `s` en vez de `b`. Sin embargo, esto sería redundante, ya que, por defecto, si no se especifica ninguno de estos valores, se abre el archivo en modo texto.

Escritura de registros

El Programa 9.7 muestra un ejemplo de cómo almacenar en un archivo estructuras de datos de C. Este es un importante mecanismo para almacenar datos complejos en un archivo. Observe que este programa es similar al que usaba tipos de datos mixtos. Sin embargo, la principal diferencia es que almacena una estructura de datos en el archivo usando una nueva función denominada `fwrite()`.

Programa 9.7

```
#include <stdio.h>
typedef struct
{
    char nombre_pieza[15];    /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
} tipo_pieza;

main()
{
    tipo_pieza pieza;         /* Variable de tipo tipo_pieza. */
    FILE *puntero_a_archivo;  /* Puntero a archivo. */

    /* Abrir un archivo para escritura. */
    puntero_a_archivo = fopen("PIEZAS.DAT", "wb");

    /* Leer datos del usuario. */
    do
    {
        printf("\nNombre de la pieza => ");
        gets(pieza.nombre_pieza);
        printf("Numero de piezas => ");
        scanf("%d", &pieza.cantidad);
        printf("Precio de cada pieza => ");
        scanf("%f", &pieza.precio_unitario);
```

```

        /* Escribir la estructura al archivo. */
        fwrite(&pieza, sizeof(pieza), 1, puntero_a_archivo);
        printf("Quiere introducir mas piezas (s/n)? => ");
    } while (getchar() != 's');

    /* Cerrar el archivo abierto. */
    fclose(puntero_a_archivo);
}

```

Suponiendo que el usuario introduce los mismos valores que antes, la ejecución del Programa 9.7 mostrará lo siguiente:

```

Nombre de la pieza => resistencia
Número de piezas => 12
Precio de cada pieza => 0.05

```

Esta información se salvará en el disco como un bloque de datos en formato binario dentro del archivo PIEZAS.DAT.

Análisis del programa

El Programa 9.7 define primero un tipo llamado `tipo_pieza` como una estructura compuesta de tres miembros. Esta es la estructura para el inventario de piezas presentada en las primeras secciones del capítulo anterior.

```

typedef struct
{
    char nombre_pieza[15];    /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
} tipo_pieza;

```

El programa declara el puntero a archivo de tipo `FILE` y la variable `pieza` de tipo `tipo_pieza`.

```

main()
{
    tipo_pieza pieza;         /* Variable de tipo tipo_pieza. */
    FILE *puntero_a_archivo; /* Puntero a archivo. */

```

Se abre un archivo llamado `PIEZAS.DAT` para escritura en modo binario (si el archivo no existe se creará).

```

/* Abrir un archivo para escritura. */
puntero_a_archivo = fopen("PIEZAS.DAT", "wb");

```

Seguidamente se le pide al usuario que introduzca datos. Obsérvese el uso del operador (.) para acceder a cada miembro de la estructura. La lectura se realiza dentro de un bucle `do` para permitir al usuario que introduzca más de una estructura:

```
/* Leer datos del usuario. */
do
{
    printf("\nNombre de la pieza => ");
    gets(pieza.nombre_pieza);
    printf("Numero de piezas => ");
    scanf("%d", &pieza.cantidad);
    printf("Precio de cada pieza => ");
    scanf("%f", &pieza.precio_unitario);
```

Después, se escriben los datos en el archivo abierto usando la función `fwrite()`. Esta función permite escribir en un archivo un determinado número de elementos. Tiene cuatro argumentos:

```
fwrite(dir_datos, tamaño, n, puntero)
```

El primer argumento contiene la dirección donde están almacenados los elementos que se quieren escribir en el archivo. El segundo argumento es el tamaño de un elemento en octetos. El tercer argumento es el número de elementos que se desea escribir. El último argumento es un puntero a archivo que especifica donde se escribirán los datos.

```
fwrite(&pieza, sizeof(pieza), 1, puntero_a_archivo);
```

Observe el uso de la función `sizeof()` para determinar el tamaño de un elemento en octetos.

Luego se le pregunta al usuario si quiere introducir más datos. Se terminará el bucle si el usuario introduce un carácter diferente a 's':

```
    printf("Quiere introducir mas piezas (s/n)? => ");
} while (getchar() == 's');
```

Al final se cierra el archivo.

```
/* Cerrar el archivo abierto. */
fclose(puntero_a_archivo);
```

Lectura de registros

El Programa 9.8 muestra cómo recuperar los bloques de datos escritos por el programa anterior. Dése cuenta de que este programa utiliza la misma definición de tipo para las piezas que el Programa 9.7.

Programa 9.8

```

#include <stdio.h>
typedef struct
{
    char nombre_pieza[15];    /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
} tipo_pieza;
main()
{
    tipo_pieza          /* Variable de tipo tipo_pieza. */
    FILE *puntero_a_archivo; /* Puntero a archivo. */
    /* Abrir un archivo para lectura. */
    puntero_a_archivo = fopen("PIEZAS.DAT","rb");
    /* Leer datos del archivo y mostrarlos por pantalla */
    while(fread(&pieza, sizeof(pieza), 1, puntero_a_archivo) == 1)
    {
        printf("\nNombre de la pieza => %s\n", pieza.nombre_pieza);
        printf("Numero de piezas => %d\n", pieza.cantidad);
        printf("Precio de cada pieza => %f\n", pieza.precio_unitario);
    }
    /* Cerrar el archivo abierto. */
    fclose(puntero_a_archivo);
}

```

Cuando se ejecuta el Programa 9.7, abrirá el archivo `PIEZAS.DAT` para lectura en modo binario (el archivo ya debe existir). Se leen los datos usando la función `fread()` que tiene el siguiente formato:

```
fread(dir_datos, tamaño, n, puntero)
```

Es muy similar a la función `fwrite()`. Como antes, `dir_datos` apunta a donde se pretende que se copien los datos leídos, `tamaño` contiene el tamaño de un elemento en octetos, `n` es el número de elementos y `puntero` referencia al archivo que se quiere leer.

La función `fread()` devuelve el número de elementos leídos del archivo. Sólo en caso de fin de archivo o de un error de lectura, el número de elementos leídos será diferente de los que se especifican en el tercer argumento de la llamada. En este programa se lee un único elemento y, por lo tanto, se repite el bucle `while` hasta que `fread()` devuelva un valor distinto de 1.

Conclusión

Esta sección ha mostrado algunos aspectos importantes relacionados con el acceso a archivos. Se ha presentado un nuevo método para almacenar datos de diferentes tipos

en un archivo y poder recuperarlos posteriormente. También se ha visto cómo trabajar con bloques de datos, como sería el caso de una estructura de C. En la próxima sección se estudiarán algunos de los aspectos técnicos de las operaciones de entrada/salida. En la Sección 9.4 se presentarán mecanismos más sofisticados para el manejo de archivos, tales como el manejo de vectores de registros con acceso aleatorio a cada registro. Compruebe su comprensión de esta sección mediante el siguiente repaso.

Repaso de la Sección 9.2

1. Enumere las cuatro posibles condiciones que pueden aparecer cuando se trabaja con un archivo.
2. Explique los diferentes métodos existentes en C para leer y escribir datos.
3. Enumere los tres modos básicos de apertura de un archivo. Explique su significado.
4. ¿Cuál es propósito de la función `fscanf()`?
5. Explique el significado de una función de lectura o escritura de bloques.

9.3. FLUJOS DE DATOS, PUNTEROS A ARCHIVOS Y ARGUMENTOS DE LA LÍNEA DE ORDENES

Presentación

Esta sección presenta algunos detalles técnicos concernientes a las operaciones de E/S. Se estudiarán también los distintos modos de acceso a los archivos. Esta sección ayudará a prepararse para la próxima donde se desarrollará un programa de aplicación.

Aspectos internos de la E/S

Un flujo de datos puede entenderse como una secuencia de octetos de información que se mandan o reciben en serie. La Figura 9.3 muestra este concepto.

Realmente ya se ha trabajado con dos tipos de flujos de datos, un flujo de datos de texto y un flujo de datos binario (archivos de texto y archivos binarios). Un flujo de datos de texto consiste en líneas de caracteres. Cada línea termina con un carácter de nueva línea (`\n`). Un importante aspecto de los flujos de datos de texto es que lo que escribe un programa en un flujo de datos no siempre coincidirá exactamente con la forma en la que realmente está guardado. Así, en DOS, el final de una línea se indica con un carácter de nueva línea más un carácter de retorno de carro (`\n\r`). No ocurre



Figura 9.3. Concepto de flujo de datos en C.

lo mismo con los flujos de datos binarios donde coincide exactamente lo almacenado con lo escrito por el programa.

En C, un archivo representa una fuente de datos almacenada en un soporte externo. La Figura 9.4 presenta algunos ejemplos de diferentes tipos de soportes.

Flujo de datos con memoria intermedia

Cuando se realiza una operación de E/S sobre un archivo, se necesita hacer una asociación entre un flujo de datos y el archivo, usando para ello una sección de **memoria intermedia** (*buffer*). Se puede interpretar esta memoria intermedia como un lugar de almacenamiento temporal de datos. Se almacenan en ella los octetos leídos del archivo o escritos en él. Así, cuando se lee un archivo del disco, se almacena en la memoria intermedia una porción de datos de tamaño fijo. Cuando un programa lee o escribe datos realmente accede a la porción correspondiente almacenada en la memoria intermedia. Por ello se denomina **flujo de datos con memoria intermedia** (*buffered*

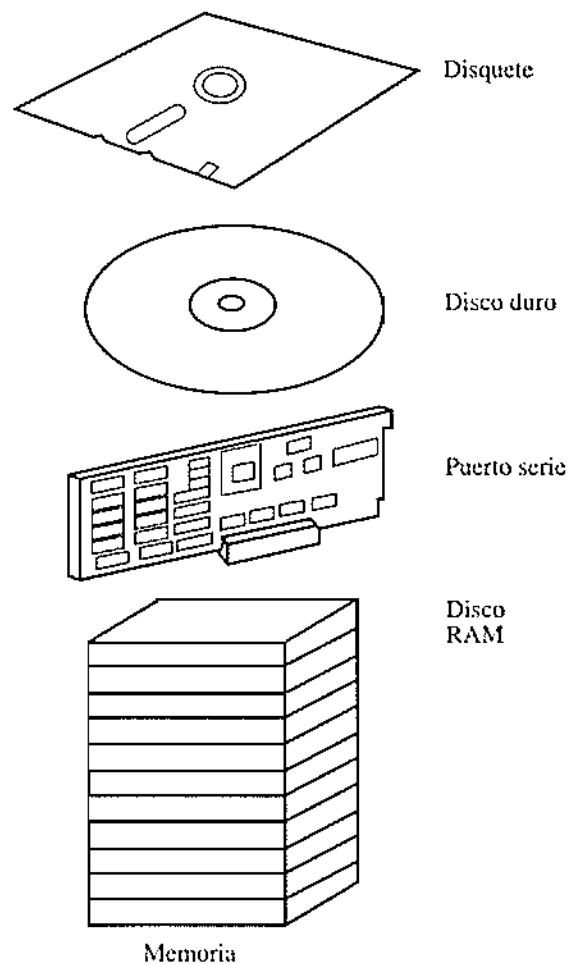


Figura 9.4. Distintos tipos de soporte de archivos en C.

stream). Sólo se almacenan realmente los datos en el disco cuando se transfieren de la memoria intermedia al disco (**limpiado** o **vaciado**). De esta forma se reduce el número de accesos al disco que hace el programa.

Si el programa termina de forma anormal, puede producirse una pérdida de datos debido a que no se haya producido el volcado de la memoria intermedia. Es interesante resaltar que el puntero a archivo que se declara en los programas que manejan archivos es realmente un puntero a esta memoria intermedia.

En el archivo de cabecera estándar `<stdio.h>`, se definen las siguientes constantes:

```
EOF = 1
NULL = 0
BUFSIZE = 512
```

Como se recordará, EOF es la marca de fin de archivo. NULL es el valor devuelto por algunas funciones, como `fopen()`, cuando ocurre un error. Por último BUFSIZE es el tamaño asignado por defecto a la memoria intermedia asociada a un flujo de datos de E/S.

Archivos estándar

Existen tres **archivos estándar** predefinidos que están automáticamente disponibles a cualquier programa: la entrada estándar (`stdin`), la salida estándar (`stdout`) y el error estándar (`stderr`). Normalmente, la entrada estándar se corresponde con el teclado, y tanto la salida como el error estándar con la pantalla. Esto se muestra en la Figura 9.5.

La existencia de una salida de error estándar permite que, aunque la salida estándar esté redirigida a un archivo, se puedan seguir imprimiendo mensajes de error por la pantalla.

Niveles de entrada/salida

Hay realmente dos niveles de entrada/salida. El primero se llama **E/S estándar** y ya se ha estado trabajando con él anteriormente. El segundo se llama **E/S de bajo-nivel**. La ventaja de la E/S estándar es que requiere menos detalles de programación. La desventaja es que tenemos menos control sobre los detalles de los procesos de E/S, y es más lento que la E/S de bajo nivel. Sin embargo, para la mayoría de las tareas de programación de E/S, el nivel estándar cumple de sobra con todas las necesidades. El nivel de E/S estándar utiliza internamente las funciones de bajo nivel. A la E/S de bajo nivel se le denomina también **E/S del sistema** y es un tema que se tratará próximamente.

Archivos de acceso aleatorio

Hasta ahora, el tipo de archivos con los que se ha trabajado han sido **archivos de acceso secuencial**, en los que no se podía acceder directamente a un dato específico.

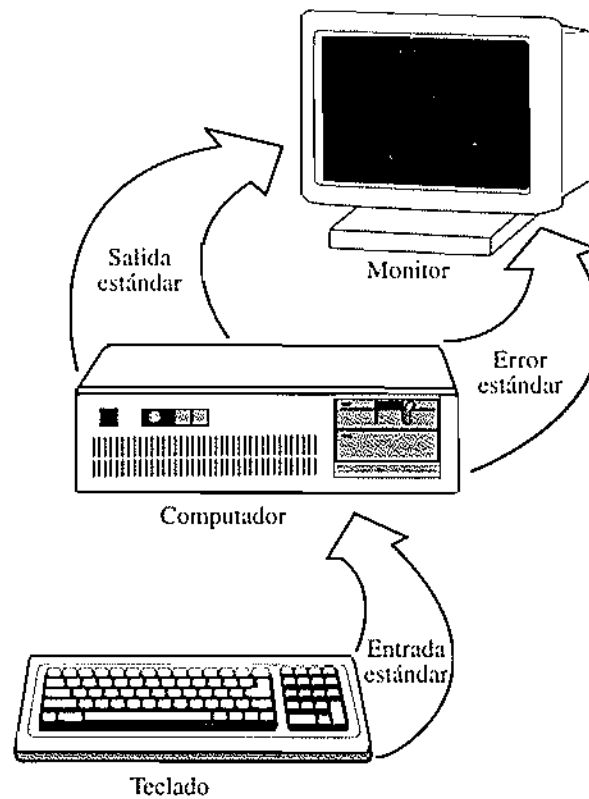


Figura 9.5. Ilustración de los tres archivos estándar.

Los **archivos de acceso aleatorio** permitirán realizar este tipo de operaciones. El Programa 9.9 muestra el uso de un archivo de acceso aleatorio, continuando con el anterior ejemplo del inventario de piezas. Para comprender este programa se debe primero saber qué significa un puntero a archivo.

Programa 9.9

```
#include <stdio.h>
typedef struct
{
    char nombre_pieza[15];    /* Tipo de pieza. */
    int cantidad;             /* Número de piezas. */
    float precio_unitario;    /* Precio de cada pieza. */
} tipo_pieza;
main()
{
    tipo_pieza pieza;         /* Variable de tipo tipo_pieza. */
    FILE *puntero_a_archivo; /* Puntero a archivo. */
    int num_registro;         /* Número de registro. */
    long int desplazamiento;  /* Desplazamiento del registro. */

    /* Abrir el archivo para lectura. */
    if ((puntero_a_archivo = fopen("PIEZAS.DAT","r")) == NULL)
```

```

    {
        printf("No puedo abrir el archivo PIEZAS.DAT\n");
        exit(-1);
    }

    /* Leer el número de registro. */
    printf("Introduzca el numero de registro => ");
    scanf("%d", &num_registro);

    /* Calcular el desplazamiento del registro seleccionado. */
    desplazamiento = num_registro * sizeof(pieza);

    /* Situar en el lugar requerido */
    if (fseek(puntero_a_archivo, desplazamiento, 0) != 0)
    {
        printf("El puntero ha superado el limite del archivo.");
        exit(-1);
    }

    /* Leer del archivo los datos seleccionados. */
    fread(&pieza, sizeof(pieza), 1, puntero_a_archivo);

    /* Mostrar los datos del archivo. */
    printf("\nNombre de la pieza => %s\n", pieza.nombre_pieza);
    printf("Numero de piezas ==> %d\n", pieza.cantidad);
    printf("Precio de cada pieza => %f\n", pieza.precio_unitario);

    /* Cerrar el archivo abierto. */
    fclose(puntero_a_archivo);
}

```

Punteros a archivos

Un **puntero a archivo** es simplemente un puntero a la posición en el archivo donde tendrá lugar el próximo acceso al archivo. Cuando se abre un archivo, el puntero apunta a la posición 0, el principio del archivo. Cada vez que se escriben datos en el archivo, el puntero termina apuntando al final de los datos escritos. Cuando se realiza una operación de añadir al final, primero se hace que el puntero señale al final del archivo antes de que se escriban los nuevos datos. La Figura 9.6 muestra el concepto de puntero a archivo.

Hay una función llamada `fseek()` que cambia de posición un puntero a archivo. Esta función moverá el puntero a la posición donde tendrá lugar el próximo acceso. El Programa 9.9 utiliza esta función para acceder a un determinado registro del archivo.

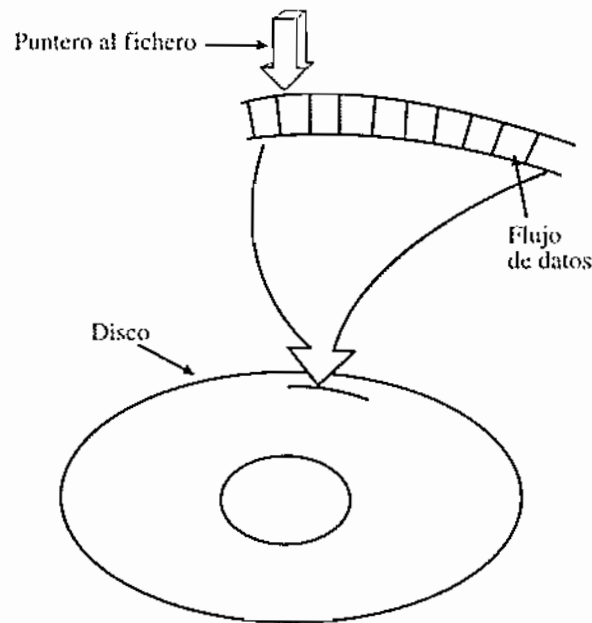


Figura 9.6. Concepto de puntero al fichero.

Análisis del programa

La primera parte del Programa 9.9 es la misma que la usada en los programas previos que utilizaban archivos de acceso secuencial. Las diferencias comienzan con la declaración de variables:

```
int num_registro;           /* Número de registro. */
long int desplazamiento;    /* Desplazamiento del registro. */
```

Se han definido dos nuevas variables `num_registro` y `desplazamiento`. La primera variable contendrá el número del registro al que el usuario quiera acceder y la segunda contendrá la posición dentro del archivo donde se almacena dicho registro. Esta variable es de tipo `long`.

Primero se abre el archivo para leer comprobando si hay error:

```
if ((puntero_a_archivo = fopen("PIEZAS.DAT", "r")) == NULL)
{
    printf("No puedo abrir el archivo PIEZAS.DAT\n");
    exit(-1);
}
```

No hay diferencia con los archivos de acceso secuencial. La diferencia comienza cuando se le pide al usuario un número de registro. Esto implica que el usuario debe saber cuántos registros hay en el archivo y lo que significa dicho número. Podría corresponder, por ejemplo, con un número identificador de la pieza.

```
printf("Introduzca el numero de registro => ");
scanf("%d", &num_registro);
```

A continuación se calcula el valor del desplazamiento multiplicando el número de registro por su tamaño. El resultado es el desplazamiento que hay que realizar para acceder al registro.

```
desplazamiento = num_registro * sizeof(pieza);
```

Ahora se usa la función `fseek()` para desplazar el puntero a la posición adecuada comprobando si hay error.

```
if (fseek(puntero_a_archivo, desplazamiento, 0) != 0)
{
    printf("El puntero ha superado el limite del archivo.");
    exit(-1);
}
```

La función `fseek()` contiene tres argumentos. El primero es el puntero a archivo, el siguiente es el desplazamiento y el último es el modo que puede tomar tres valores:

```
SEEK_SET (0) => desplazamiento desde el principio del archivo
SEEK_CUR (1) => desplazamiento desde la posición actual
SEEK_END (2) => desplazamiento desde el final del archivo
```

La función `fseek()` devuelve un valor distinto de cero si se produce un error.

En el programa se utiliza el desplazamiento desde el principio del archivo. Téngase en cuenta que un desplazamiento positivo mueve el puntero hacia el final del archivo, mientras que uno negativo lo mueve hacia el principio.

Una vez movido el puntero, se lee el registro usando la siguiente sentencia:

```
fread(&pieza, sizeof(pieza), 1, puntero_a_archivo);
```

En la parte final del programa, se muestran los datos por la pantalla y se cierra el archivo.

```
printf("\nNombre de la pieza => %s\n", pieza.nombre_pieza);
printf("Numero de piezas => %d\n", pieza.cantidad);
printf("Precio de cada pieza => %f\n", pieza.precio_unitario);
fclose(puntero_a_archivo);
```

Argumentos de la línea de órdenes

Una característica interesante del lenguaje C es que permite acceder desde el programa a los argumentos que se especifican en la línea de mandatos cuando se ejecuta el programa. De esta forma, el comportamiento del programa puede quedar influenciado por el valor de estos argumentos.

Acceso a los argumentos de la línea de órdenes

El Programa 9.10 muestra cómo se acceden a los argumentos de la línea de órdenes desde dentro de un programa. Como se puede observar, dichos argumentos son visibles en el programa a través de los parámetros de la función `main()`.

Programa 9.10 `#include <stdio.h>`

```
main(int argc, char *argv[]) /* Argumentos de la línea de mandatos. */
{
    int contador; /* Contador del número de argumentos. */

    /* Mostrar el número de argumentos */
    printf("El numero de argumentos es %dn", argc);

    /* Mostrar cada uno de los argumentos */
    for(contador = 0; contador < argc; contador++)
        printf("El argumento %d es %s\n", contador, argv[contador]);
}
```

Suponiendo que este programa se denomina `PROGRAMA`, la ejecución del siguiente mandato:

`PROGRAMA UNO DOS TRES`

Producirá la siguiente salida:

```
El argumento 0 es PROGRAMA
El argumento 1 es UNO
El argumento 2 es DOS
El argumento 3 es TRES
```

La Figura 9.7 muestra el formato de los argumentos de la línea de órdenes.

Observe que los dos argumentos `argc` y `argv` representan respectivamente el número de argumentos y un vector de punteros a dichos argumentos. El primer argumento es siempre el nombre del programa. Los nombres `argc` y `argv` son sólo una convención y, en su lugar, se podría usar cualquier identificador válido de C.

Conclusión

Esta sección ha presentado algunos de los detalles técnicos concernientes a la E/S en C. El material presentado aquí se empleará en el desarrollo del programa de aplicación de la siguiente sección. Compruebe que ha asimilado los conceptos que contiene esta sección mediante este repaso.

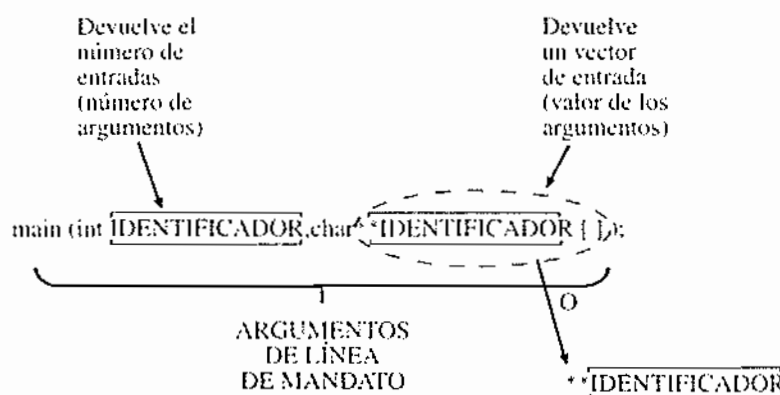


Figura 9.7. Forma de los argumentos de la línea de mandatos.

Repaso de la Sección 9.3

1. Nombre los dos tipos de flujo de datos utilizados en las operaciones de E/S.
2. ¿Qué es una memoria intermedia en E/S?
3. Nombre los tres archivos estándares utilizados por un programa C.
4. Describa la función de los argumentos de la línea de mandatos.

9.4. PROGRAMA DE APLICACIÓN: UNA BASE DE DATOS DE PIEZAS

Presentación

Esta sección muestra el desarrollo de un sistema de base de datos sencillo que ilustra los principales aspectos de un sistema real de gestión de bases de datos.

El problema

Crear un programa que almacene datos sobre piezas. Habrá un máximo de 25 piezas. Se almacenarán los siguientes datos de cada pieza: su nombre, la cantidad existente y su precio. El usuario podrá acceder de forma aleatoria a la información de cualquier pieza. Sólo un usuario autorizado podrá introducir datos en la base de datos previniendo así que usuarios no autorizados puedan modificar la información.

Primer paso: Definición del problema

La definición del problema es la siguiente:

- Propósito del programa: Crear un vector de 25 estructuras. Cada estructura contendrá el número de registro, el nombre del elemento, la cantidad de elementos y el coste de cada uno. El vector se almacenará en un archivo y sólo lo podrán

modificar los usuarios autorizados. Cualquier usuario podrá acceder a cualquier registro.

- Entrada requerida: Identificación del usuario (para seleccionar si el usuario está autorizado para modificar los datos). Selección del usuario para leer el archivo y, si está autorizado, para introducir nuevos datos.
- Proceso sobre la entrada: almacenar los datos del vector en un archivo. Leer con un acceso aleatorio los registros del mismo.
- Salida requerida: Dos menús, uno para los usuarios autorizados y otro para el resto. Mostrar el contenido de los registros accedidos por el usuario de forma aleatoria.

Desarrollo del algoritmo

Los pasos del programa son los siguientes:

1. Comprobar la autorización del usuario.
2. Explicar al usuario el funcionamiento del programa.
3. Leer las opciones del usuario.

Si autorizado => introducir nuevos datos.
Cualquier usuario => ver cualquier registro.

4. Mostrar el registro leído.

El programa usará una función que escribe un vector en un archivo y otra función que permite acceder a cualquier registro del archivo.

Desarrollo del programa

Se recomienda usar una estructura en bloques y un diseño descendente. Se desaconseja la utilización de variables globales. En primer lugar se desarrollará el bloque del programador.

A continuación se muestra el programa completo (Programa 9.11).

```
Programa 9.11 #include <stdio.h>
               #include <conio.h>
               #include <string.h>
               #include <math.h>
               #include <stdlib.h>

               #define TRUE 1          /* VERDADERO */
               #define FALSE 0        /* FALSO */

               /*
Programa: Programa de inventario de piezas
Desarrollado por: A. C. Programador
```

Descripción:

Este programa realiza el inventario de hasta 25 piezas diferentes de un surtido. Cada pieza del surtido contiene datos acerca del nombre de la pieza, el número de piezas y el precio de cada pieza.

El programa almacena y recupera esta información de un archivo denominado PIEZAS.DAT. El programa sólo permite introducir datos a usuarios autorizados. No se requiere ninguna autorización especial para leer datos del archivo.

La autorización para escribir datos en el archivo se consigue introduciendo "Piezas Acceso 123" en la línea de ordenes.

Variables:

cadena_ent[10]	= Cadena introducida por el usuario.
ch	= Caracter.
repetir	= Variable para el bucle de repetición.
indice	= Número de registro.
autorizacion	= Autorización para escribir.
*archivo	= Puntero a archivo.
vector_piezas[25]	= Hasta 25 piezas del surtido.
registro	= Estructura para leer del archivo.
desplazamiento	= Desplazamiento para acceso aleatorio.

Definición de tipos:

```
*/
```

```
typedef struct
```

```
{
    char pieza[20];      /* Tipo de pieza. */
    int cantidad;        /* Número de piezas. */
    double precio;       /* Precio de cada pieza. */
    int num_registro;    /* Número del registro de piezas. */
} registro_piezas;
/* Este tipo define cada pieza del surtido. */
/* Prototipos de funciones: */
```

```
void explicar_programa(void);
```

```
/*
```

```
Esta función explica al usuario la tarea que lleva a cabo
el programa.
```

```
*/
```

```
char leer_menu(int autorizacion);
```

```
/*
```

```
Esta función muestra por pantalla el menú de selección
del programa y devuelve el valor seleccionado.
```

```
*/
```

```

void introducir_datos(void);
/*
    Esta función permite al usuario introducir nuevos datos en el
    archivo.
*/

void leer_datos(int num_reg);
/*
    Esta función lee un registro dado del archivo
*/

main(int argc, char *argv[])
{
    char cadena_ent[10];    /* Cadena introducida por el usuario. */
    char ch;                /* Caracter. */
    int repetir = TRUE;     /* Variable para el bucle de repetición. */
    int indice;             /* Número de registro. */
    int autorizacion;       /* Autorización para escribir. */

    /* Comprueba la autorización para escribir en el archivo. */
    if (strcmp(argv[1], "Piezas Acceso 123") == 0)
        autorización = TRUE;
    else
        autorizacion = FALSE;
    explicar_programa();    /* Explicar el programa al usuario. */
    while(repetir)
    {
        /* Leer la selección del usuario. */
        ch = read_menu(autorizacion);
        printf("\n\n");
        switch(ch)
        {
            case 'R':
                printf("\n\nIntroduzca el numero de ");
                printf("registro => ");
                gets(cadena_ent);
                indice = atoi(cadena_ent);
                leer_datos(indice);
                break;
            case 'X': repetir = FALSE;    /* Salir. */
                break;
            case 'E': if (autorizacion)
                {
                    /* Introducir nuevos datos. */
                    introducir_datos();
                    break;
                }
            default: printf("Esa seleccion no es correcta:\n");
        }
    }
}

```

```

    }
}

void explicar_programa()
{
    printf("\n\n\n");
    printf(" Este programa permite almacenar y recuperar datos");
    printf("de hasta un total de 25 piezas de un surtido. Se ");
    printf("permite introducir y recuperar el nombre de la pieza");
    printf("el numero de piezas y el precio de cada una de las");
    printf("piezas. Solo es posible introducir nuevos datos");
    printf("mediante la correspondiente autorizacion.");
}

char leer_menu(int autorizacion)
{
    char ch;          /* Selección del usuario. */

    /* Imprimir el menú. */
    printf("\n\nSeleccione por letra una de las siguientes");
    printf(" opciones:\n\n");
    if (autorizacion)
        printf(" E] Introducir un nuevo dato\n");

    printf(" R] Leer datos del archivo. \n");
    printf(" X] Salir.\n");

    /* Leer la opción del usuario. */
    printf("Opcion: ");
    ch = toupper(getchar());
    return(ch);
}

void introducir_datos()
{
    FILE *archivo;          /* Puntero a archivo. */
    registro_piezas vector_piezas[25]; /* Hasta 25 piezas. */
    char cadena_ent[10];    /* Entrada del usuario. */
    int indice;             /* Número de registro. */
    int repetir;            /* Variable de repetición. */
    char ch;                /* Respuesta del usuario. */

    repetir = TRUE;
    indice = 1;
    while(repetir)
    {
        /* Leer el nombre de la pieza. */
        printf("\nNombre de la pieza => ");

```

```

    gets(vector_piezas[indice].pieza);

    /* Leer el número de piezas. */
    printf("Numero de piezas => ");
    gets(cadena_ent);
    vector_piezas[indice].cantidad = atoi(cadena_ent);

    /* Leer el precio de cada pieza. */
    printf("Precio de cada pieza => ");
    gets(cadena_ent);
    vector_piezas[indice].precio = atof(cadena_ent);

    /* Actualizar el número de registro. */
    vector_piezas[indice].num_registro = indice;

    /* Introducir más datos? */
    printf("Desea introducir mas datos (S/N) => ");
    ch = toupper(getchar());
    indice++;
    if(ch != 'S')
        repetir = FALSE;          /* Salir del bucle. */    }

    /* Escribir los datos al archivo. */
    if((archivo = fopen("PIEZAS.DAT", "wb")) == NULL)
    {
        printf("No puedo abrir el archivo de piezas.\n");
        exit(-1);
    }

    /* Introducir los datos en el archivo. */
    fwrite(vector_piezas, sizeof(vector_piezas), indice, archivo);

    /* Cerrar el archivo. */
    fclose(archivo);
}

void leer_datos(int num_reg)
{
    FILE *archivo;          /* Puntero a archivo. */
    registro_piezas registro; /* Estructura para leer del archivo. */
    long int desplazamiento; /* Desplazamiento para accesos
                               aleatorios. */

    /* Abrir el archivo para lectura. */
    if ((archivo = fopen("PIEZAS.DAT", "rb")) == NULL)
    {
        printf("No puedo abrir el archivo de piezas. \n");
        exit(-1);
    }
}

```

```

/* Calcular el desplazamiento. */
desplazamiento = num_reg * sizeof(registro);

/* Desplazar el puntero del archivo a la posición indicada. */
if(fseek(archivo, desplazamiento, SEEK_SET) != 0)
{
    printf("No puedo encontrar el registro indicado.");
    exit(-1);
}

/* Recuperar los datos del archivo. */
fread(&registro, sizeof(registro), 1, archivo);

/* Cerrar el archivo. */
fclose(archivo);

printf("\nRegistro %d\n", registro.num_registro);
printf("Pieza => %s\n", registro.pieza);
printf("Precio unitario => %f\n", registro.precio);
}

```

Análisis del programa

El usuario especificará su autorización como un argumento de la línea de órdenes.

Nótese que en la sentencia `switch` de la función `main()`, el caso correspondiente a la letra `E` continuará en la sección `default` si el usuario no tiene autorización.

Obsérvese el uso de la función `toupper()` que permite que el usuario introduzca indistintamente en mayúsculas o en minúsculas sus peticiones.

Por último, es interesante resaltar que el programa utiliza archivos binarios, como queda reflejado con el uso de la opción `b` en la función `fopen()`.

Conclusión

Esta sección ha presentado el desarrollo de un sistema de base de datos sencillo que usa acceso aleatorio a archivos y un mecanismo de seguridad. La autoevaluación que aparece al final del capítulo incluye preguntas específicas acerca de este programa.

9.5. PROGRAMAS DE APLICACIÓN ADICIONALES

El Programa 9.12 se usa para mostrar un archivo de texto pantalla a pantalla. Después de completar una pantalla, el programa se para hasta que el usuario pulse una tecla y

entonces se muestra la siguiente página. Este programa es útil para ver archivos de texto largos.

Programa 9.12 #include <stdio.h>

```
main(int argc, char *argv[])
{
    FILE *fptr;
    char fchar, temp;
    int lineas = 0;

    if (argc > 1)
    {
        fptr = fopen(argv[1], "r");
        if (fptr == NULL)
        {
            printf("No puedo abrir %s\n", argv[1]);
            exit();
        }
    }
    else
    {
        printf("No se ha especificado ningun archivo.\n");
        exit();
    }
    fchar = getc(fptr);
    while(fchar != EOF)
    {
        printf("%c", fchar);
        if (fchar == '\n')
        {
            lineas++;
            if (lineas == 23)
            {
                lineas = 0;
                printf("\nMas...");
                temp = getch();

                /* Borrar "Mas..." de la pantalla */
                printf("\b\b\b\b\b\b\b\b");
                printf(" ");
                printf("\b\b\b\b\b\b\b\b");
            }
        }
        fchar = getc(fptr);
    }
    fclose(fptr);
}
```

El programa cuenta el número de caracteres de final de línea que aparecen en el archivo. Cada vez que la cuenta llega a 23, imprime el siguiente mensaje en la parte inferior de la pantalla:

Mas...

Cuando el usuario presiona una tecla, desaparece este mensaje y se muestran las 23 líneas siguientes del archivo. Este proceso continúa hasta que se ha mostrado todo el archivo. Obsérvese que el nombre del archivo se especifica en la línea de mandatos.

El último programa de esta sección sirve para proporcionar un nivel de seguridad al contenido de un archivo de texto, mediante el uso de una técnica sencilla de cifrado de la información. El programa cambia tanto las letras mayúsculas como las minúsculas del archivo de entrada por otras letras diferentes. El archivo de entrada debe tener una extensión TXT. El archivo de salida que se crea tendrá el mismo nombre pero una extensión SCR.

Programa 9.13

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

main(int argc, char *argv[])
{
    FILE *archivo_entrada, *archivo_salida;
    char archivo_fuente[12] = "", archivo_destino[12] = "", fchar;

    if (argc > 1)
    {
        strcat(archivo_fuente, argv[1]);
        strcat(archivo_fuente, ".txt"); /* Utilizar ".txt" para
                                         leer */
        archivo_entrada = fopen(archivo_fuente, "r");
        if (archivo_entrada == NULL)
        {
            printf("No puedo abrir %s\n", archivo_fuente);
            exit();
        }
    }
    else
    {
        printf("No se ha especificado ningun archivo.\n");
        exit();
    }
    strcat(archivo_destino, argv[1]);
    strcat(archivo_destino, ".scr"); /* Utilizar ".ser" para
                                     codificar */
    archivo_salida = fopen(archivo_destino, "w");
    if (archivo_salida == NULL)
```



```

{
    printf("No puedo abrir %s\n",archivo_destino);
    exit();
}
fchar = getc(archivo_entrada);
while(fchar != EOF)
{
    if (0 != isupper(fchar))
    {
        fchar += 13;
        fchar = (0 == isupper(fchar)) ? fchar - 26 : fchar;
    }
    if (0 != islower(fchar))
    {
        fchar += 13;
        fchar = (0 == islower(fchar)) ? fchar - 26 : fchar;
    }
    putc(fchar,archivo_salida);
    fchar = getc(archivo_entrada);
}
fclose(archivo_entrada);
fclose(archivo_salida);
}

```

La técnica utilizada para cifrar los caracteres, denominada el desplazamiento Cesar, simplemente desplaza las letras del alfabeto 13 posiciones hacia delante (pero de forma circular, cuando se pasa de la z/Z empieza por la a/A).

El siguiente archivo de entrada:

He estado trabajando en el emulador del 68000.
Las instrucciones TRAP ya funcionan correctamente.

Se convertiría en:

Ur rfgnqb genonwnaqb ra ry rzhyngbe qry 68000.
Ynf vafgehppvbarf GENC ln shapvbana pbeerpgnzragr.

Evidentemente, el contenido del archivo de salida no tiene sentido y sería necesario emplear un buen rato par descifrarlo.

Para descifrar un archivo simplemente se debe ejecutar el mismo programa pero tomando como archivo de entrada el de salida. Sin embargo, debido a la rigidez del programa en el manejo de los nombres de los archivos, habría que renombrar el archivo antes de descifrarlo.

Repaso de la Sección 9.5

1. ¿Cómo comprueba un programa que no se ha producido un error en la apertura de un archivo?
2. ¿Cómo se borra de la pantalla el mensaje Mas...?

3. ¿Por qué se debe comprobar al principio de cada programa que `argc` es mayor que 1?
4. ¿Cómo realiza el Programa 9.13 el desplazamiento cuando cifra una letra?

Ejercicios interactivos

DIRECTRICES

La realización de estos ejercicios requiere tener acceso a un computador con un entorno C. Se han incluido aquí para permitirle adquirir una valiosa experiencia y, lo que es más importante, para tener una realimentación inmediata de lo que hacen los conceptos y mandatos presentados en este capítulo. Además son divertidos.

Ejercicios

1. El Programa 9.14 crea un archivo y pone algo en él. Después de ejecutar el programa, compruebe que se ha creado un nuevo archivo.

Programa 9.14 `#include <stdio.h>`

```
main()
{
    FILE *puntero_a_archivo;

    puntero_a_archivo = fopen("ARCHIVO.DAT", "w");
    putc('C', puntero_a_archivo);
    fclose(puntero_a_archivo);
}
```

2. El Programa 9.15 lee del archivo generado por el Programa 9.14. Pruébalo y compruebe si lee lo previsto.

Programa 9.15 `#include <stdio.h>`

```
main()
{
    FILE *puntero_a_archivo;
    char ch;

    puntero_a_archivo = fopen("ARCHIVO.DAT", "r");
    ch = getc(puntero_a_archivo);
    printf("El caracter es %c", ch);
    fclose(puntero_a_archivo);
}
```

3. ¿Qué cambios hace el Programa 9.16 en el archivo ARCHIVO.DAT? ¿Es necesario que exista dicho archivo para ejecutar el programa?

Programa 9.16 #include <stdio.h>

```
main()
{
    FILE *puntero_a_archivo;

    puntero_a_archivo = fopen("ARCHIVO.DAT", "a");
    putc('!', puntero_a_archivo);
    fclose(puntero_a_archivo);
}
```

Autoevaluación

Responda a las siguientes preguntas referidas al Programa 9.11.

Preguntas

1. Nombre los miembros de la estructura definida en el programa.
2. Explique de qué tipo es la función leer_menu.
3. ¿Hay variables globales en el programa? Si es así, nómbrelas.
4. Explique el propósito de los argumentos de la función main().
5. ¿Por qué en el switch de la función main() el caso correspondiente a la letra E está situado justo antes de la opción de default?
6. ¿Se usa en este programa la función scanf()?
7. ¿El programa responderá correctamente si el usuario utiliza mayúsculas o minúsculas?
8. ¿Está en modo binario o en modo texto la información en el archivo?

Problemas del final del capítulo

Conceptos generales

Sección 9.1

1. Nombre una ventaja del uso de un archivo.
2. ¿Es válido el siguiente nombre de archivo para DOS?

B:FICH.02

3. Explica lo que significa el nombre de archivo de la pregunta anterior.
4. ¿Cómo se declara un puntero a archivo? Ponga un ejemplo.
5. ¿Qué debe hacer siempre un programa que utiliza archivos antes de terminar?

Sección 9.2

6. Enumere los diversos métodos que existen para leer y escribir datos en C.
7. Enumere los tres modos básicos de apertura de un archivo.
8. ¿Qué sucede si intenta abrir un archivo para escritura y éste no existe?

Sección 9.3

9. Enumere los dos tipos de flujo de datos que hay en C.
10. ¿Cuáles son los tres archivos estándar usados en C?
11. ¿Cuáles son las representaciones físicas usuales de los tres archivos estándar usados en C?

Diseño de programas

Para el desarrollo de los siguientes programas, use un diseño descendente y una estructura en bloques sin variables globales. La función `main()` se encargará básicamente de llamar al resto de las funciones. Cuando una función necesite devolver más de una variable, se usarán punteros. No olvide incluir toda la documentación en la versión final del programa.

12. Modifique el Programa 9.11 para que muestre los registros en orden alfabético con respecto al nombre de la pieza.
13. Cambie el Programa 9.13 añadiendo otra estructura que contenga el nombre del fabricante de la pieza, su dirección (calle, ciudad y provincia) y su número de teléfono. Esta nueva estructura se convertirá en un miembro de la estructura existente.

14. Desarrolle un programa que se comporte como un procesador de texto sencillo. El programa debe permitir al usuario salvar y recuperar archivos de texto, darles nombre y borrar los archivos viejos.
15. Escriba un programa que busque en un archivo de entrada todas las ocurrencias de una cadena de caracteres. El archivo de entrada y la cadena buscada se especificarán en la línea de mandatos de la siguiente forma:

```
BUSCAR <archivo> <cadena>
```

16. Escriba un programa que haga una lista de todas las palabras encontradas en un archivo de entrada y escriba la lista ordenada en un archivo de salida, junto con el número de veces que aparece cada palabra en el archivo de entrada.

Aspectos avanzados

Objetivos

En este capítulo podrá aprender:

1. Cómo usar variables de tipo extern.
2. Cómo declarar y usar punteros a funciones.
3. Cómo pasar funciones como argumentos.
4. Cómo declarar funciones con un número variable de argumentos.
5. Cómo crear una aplicación con varios archivos fuente.
6. Cómo mantener aplicaciones con varios archivos fuente usando la utilidad make.

Palabras clave

Variable externa	Módulo fuente
Diseño modular	Argumentos dinámicos
Programación modular	Definición de función
Pila	Aplicaciones grandes
Convenciones de llamada de funciones	Utilidad make

Contenido

10.1. Punteros a funciones	10.4. Aplicaciones modulares
10.2. Funciones como argumentos	10.5. La utilidad make
10.3. Funciones con número variable argumentos	10.6. Programas de aplicación adicional

Introducción

En este capítulo final se examinarán brevemente algunos temas avanzados de la programación en C. Estos temas conciernen al uso avanzado de funciones y recursos externos, así como a la programación de aplicaciones modulares y a la herramienta MAKE que permite compilar dichas aplicaciones de forma automática. Estos temas intentan aumentar su interés y animarle a seguir haciendo programas en C para su uso particular o en su trabajo. Use los manuales de referencia para su entorno particular de programación en C para ver los detalles específicos de implementación y uso de algunos de los temas tratados en este capítulo. Algunos de ellos tienen detalles que pueden depender de dicho entorno.

En todas las secciones de este capítulo usaremos tomaremos como base para los ejemplos el Programa 3.15.

10.1. PUNTEROS A FUNCIONES

Presentación

En esta sección se muestra como definir punteros a funciones, una nueva forma de llamar a una función como si fuera un puntero y asignar sus resultados a valores de tipo puntero. Además se muestra como se declaran vectores de punteros a función, algo muy útil para realizar operaciones vectorizadas por un código (por ejemplo, interrupciones del hardware).

Cálculo de la ley de Ohm con punteros a funciones

Para mostrar el uso de los punteros a funciones se usará una modificación del Programa 3.15. La primera modificación que se hará será extraer el cálculo de los distintos parámetros de la ley de Ohm a funciones. Para ello se elaboran tres funciones

```
void calculo_voltaje(void); /* Cálculo del voltaje en voltios. */
void calculo_intensidad(void); /* Cálculo de la intensidad en amperios. */
void calculo_resistencia(void); /* Cálculo de la resistencia en ohmios. */
```

que se llaman desde el programa principal. El Programa 10.1 muestra la implementación de dichas funciones.

Además se han definido dos funciones

```
float multiplicar(float a, float b); /* Multiplica a * b */
float dividir(float a, float b); /* Dividir a / b */
```

para hacer las operaciones en las funciones anteriores.

El lenguaje C permite que los valores de un tipo se puedan convertir automáticamente a otro en ciertas circunstancias. Esta conversión determina si hay que convertir un operando, y cómo, antes de una operación. Esta conversión se puede aplicar a una expresión del tipo *función que devuelve T* para obtener una expresión del tipo *puntero*

a función que devuelve *T*, sustituyendo el puntero a la función por la función misma. La única expresión del tipo *función que devuelve T* es el nombre de la función. Lo que se convierte al tipo *puntero a función que devuelve T* es el identificador en sí mismo y no en el contexto de una llamada a función (el identificador seguido de "("). Por tanto, en el caso habitual es necesario declarar la función de forma normal y después declarar un puntero a función al cuál se puede asignar el nombre de la función. Es muy importante tener en cuenta que cuando se declara un puntero a función para asignarle nombres de funciones, el tipo puntero declarado debe tener un prototipo coincidente con el de las funciones que se le quieren asignar. En el Programa 10.1 pueden verse varias declaraciones de puntero a función. El Programa 10.3 permite al usuario ejecutar dos series matemáticas basadas en sumar a 2 varios números pares y en multiplicarlos por 2, respectivamente. Para ello se pide al usuario que elija la operación a realizar (suma o multiplicación (*float*)). El tipo puntero declarado para asignarlo a las funciones *multiplicar* y *dividir* es:

```
float (*operacion)(float a, float b); /* Operacion con a y b */
```

Esta declaración debería observarse con cuidado por dos motivos. Primero, es importante poner los paréntesis que rodean al puntero, ya que sin ellos, **operacion*, no estaríamos declarando un puntero a función, sino una función que devuelve un puntero a un *float*. Segundo, los prototipos de las funciones que se quieran asignar al puntero deben ser idénticos a los usados en la declaración del puntero a función. A la variable *operacion* se le puede asignar el nombre de cualquier variable que coincida con ese prototipo, de tal forma que, a partir del instante de la asignación, *operacion* apunta a la dirección donde está la función que se le asignó. Al ejecutar *operacion* obtenemos un comportamiento idéntico al de la función asignada. Así, la sentencia:

```
operacion = dividir;
```

apunta la variable *operacion* a la función *dividir*, de tal suerte que es idéntico ejecutar

```
resistencia = (*operacion)(voltaje , intensidad);
```

que

```
resistencia = dividir(voltaje , intensidad);
```

La sintaxis del lenguaje C para manejar punteros a funciones hace que los programas que los usan se vuelvan un tanto oscuros, especialmente para usuarios con poca experiencia en el uso del lenguaje. Este problema empeora cuando los punteros a funciones se usan como miembros de estructuras, se declaran vectores de punteros a función o se pasan como argumentos a otras funciones. Para ilustrar uno de estos casos, en el Programa 10.1 se muestra un vector de punteros a función. Las funciones apuntadas son las que calculan la intensidad, resistencia y voltaje en el caso de la ley de Ohm.

```

void (*ley_ohm[3])(void) =
{
    calculo_voltaje,
    calculo_intensidad,
    calculo_resistencia
};

```

Obsérvese este vector de funciones. Está definiendo un vector de 3 punteros a función (una vez más los paréntesis del caso `(*ley_ohm[3])` son fundamentales), todas las cuáles no devuelven ningún valor ni reciben ningún argumento. Esta característica es fundamental cuando se declaran vectores de punteros a función, siendo obligatorio que el tipo del valor que devuelven las funciones del vector, y los argumentos de las mismas, no se definan a nivel de función individual, sino a nivel de todo el vector. Las funciones de un vector se pueden manipular de la misma forma que los elementos de cualquier otro vector (índice del vector, dirección de los elementos, punteros, etc.). Un ejemplo de llamada a una función de un vector de punteros a función puede verse dentro de `main` del Programa 10.1.

Programa 9.

```

switch(eleccion)
{
    case 'A' : (*ley_ohm[0])();
               break;
    case 'B' : (*ley_ohm[1])();
               break;
    case 'C' : (*ley_ohm[2])();
               break;
} /* Fin del switch. */

```

Aunque al usuario del programa le es indiferente la estructura del vector de funciones y el orden de las mismas en el vector, es muy importante para cualquiera que desee comprender o modificar el programa que su estructura sea lo más clara posible.

Programa 10.1 `#include <stdio.h>`

```

void calculo_voltaje(void); /* Cálculo del voltaje en voltios. */
void calculo_intensidad(void); /* Cálculo de la intensidad en amperios. */
void calculo_resistencia(void); /* Cálculo de la resistencia en ohmios. */
float multiplicar(float a, float b); /* Multiplica a * b */
float dividir(float a, float b); /* Dividir a / b */

void (*ley_ohm[3])(void) =
{
    calculo_voltaje,
    calculo_intensidad,
    calculo_resistencia
};

```



```

main()
{
    char eleccion;      /* Elección del usuario.          */
    float voltaje;      /* Voltaje del circuito en voltios.          */
    float intensidad;   /* Intensidad del circuito en amperios.      */
    float resistencia;  /* Resistencia del circuito en ohmios.      */

    printf("\n\nElija la forma de la ley de Ohm que corresponda:\n");
    printf("A) Voltaje B) Intensidad C) Resistencia\n");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A' : (*ley_ohm[0])();
                    break;
        case 'B' : (*ley_ohm[1])();
                    break;
        case 'C' : (*ley_ohm[2])();
                    break;
    } /* Fin del switch. */
}

void calculo_voltaje(void) /* Cálculo del voltaje en voltios. */
{
    float (*operacion)(float a, float b); /* Operacion con a y b */
    float voltaje; /* Voltaje del circuito en voltios. */
    float intensidad; /* Intensidad del circuito en amperios. */
    float resistencia; /* Resistencia del circuito en ohmios. */

    operacion = multiplicar;
    printf("Introduzca la intensidad en amperios => ");
    scanf("%f",&intensidad);
    printf("Valor de la resistencia en ohmios => ");
    scanf("%f",&resistencia);
    voltaje = (*operacion)(intensidad, resistencia);
    printf("El voltaje es %f voltios.",voltaje);
}

void calculo_intensidad(void) /* Cálculo de la intensidad en amperios. */
{
    float (*operacion)(float a, float b); /* Operacion con a y b */
    float voltaje; /* Voltaje del circuito en voltios. */
    float intensidad; /* Intensidad del circuito en amperios. */
    float resistencia; /* Resistencia del circuito en ohmios. */

    operacion = dividir;
    printf("Introduzca el voltaje en voltios => ");
    scanf("%f",&voltaje);
    printf("Valor de la resistencia en ohmios => ");
    scanf("%f",&resistencia);
    intensidad = (*operacion)(voltaje , resistencia);
}

```

```

        printf("La intensidad es %f amperios.",intensidad);
    }
void calculo_resistencia(void) /* Cálculo de la resistencia en ohmios.*/
{
    float (*operacion)(float a, float b); /* Operacion con a y b */
    float voltaje; /* Voltaje del circuito en voltios. */
    float intensidad; /* Intensidad del circuito en amperios. */
    float resistencia; /* Resistencia del circuito en ohmios. */

    operacion = dividir;
    printf("Introduzca el voltaje en voltios => ");
    scanf("%f",&voltaje);
    printf("Valor de la intensidad en amperios => ");
    scanf("%f",&intensidad);
    resistencia = (*operacion)(voltaje , intensidad);
    printf("La resistencia es %f ohmios.",resistencia);
}
float multiplicar(float a, float b) /* Multiplica a * b */
{
    return (a*b);
}
float dividir(float a, float b) /* Dividir a / b */
{
    if (b == 0)
        return (-1);
    else
        return (a/b);
}

```

Conclusión

En esta sección se han presentado los punteros a funciones en C y se ha indicado cómo usarlos de distintas formas y dentro de distintas estructuras de datos. Ahora el usuario puede tener una idea de lo que hace falta para usar los punteros a función y de par qué sirven. En la siguiente sección se aprenderá cómo pasar funciones, y también punteros a función, como argumentos de una función. Compruebe su nivel de comprensión de esta función haciendo los siguientes ejercicios de repaso.

Repaso de la Sección 10.1

1. ¿Qué es un puntero a función?
2. Defina por qué son importantes los paréntesis que rodean al puntero a función declarado.
3. ¿Se puede usar un puntero a función como miembro de una estructura?
4. Defina un puntero `punt` a la función `int sumar (int numero);`

10.2. FUNCIONES COMO ARGUMENTOS

Presentación

En esta sección se muestra cómo definir prototipos de función que reciben a otras funciones como parámetros. Este concepto es importante cuando se quiere programar alguna función que ejecuta algo que no es conocido cuando se programa dicha función. El efecto de esta función será desconocido hasta el momento de su ejecución.

Cálculo de la ley de Ohm con funciones como argumentos

El uso de funciones como argumentos es muy útil para construir funciones de propósito general. Sin embargo, esta facilidad debe usarse con mucho cuidado puesto que los efectos de la ejecución de una función que recibe como argumento a otra función no son siempre obvios. Los programas que usan funciones como argumentos suelen ser difíciles de comprender y de depurar, pero la potencia de este mecanismo sobrepasa estos inconvenientes. La utilidad de construir funciones que ejecuten otras funciones que reciben como parámetros reside en que la función que se pasa como parámetro puede depender de la ejecución de la aplicación y, por tanto, reflejar dinámicamente las necesidades de la misma.

Para mostrar el uso de funciones como argumentos se modificará el Programa 10.1, introduciendo una nueva función cuya única misión sea ejecutar la función que reciba como parámetro. El prototipo de dicha función es:

```
void calculo_ley_ohm (void (*funcion)(void));
```

Cuando se compila esta función, sólo se sabe que recibirá como parámetro una función que se ajusta al prototipo declarado como argumento. La función específica no se conoce hasta el momento en que se llama a `calculo_ley_ohm` y se detallan sus argumentos. Cuando se llama a la función se resuelve el nombre que recibe como argumento y se ejecuta dicha función. La forma de llamar a esta función es:

```
calculo_ley_ohm(*ley_ohm[0]);
```

Internamente, se ejecuta la función recibida como parámetro sin más que llamar a dicha función siguiendo las reglas de llamadas a funciones en C.

```
void calculo_ley_ohm (void (*funcion)(void))
{
    funcion();
}
```

En el Programa 10.2 se muestra como se puede calcular la ley de Ohm usando funciones como argumento y se desarrollan los fragmentos de código mostrados anteriormente.

Programa 10.2 #include <stdio.h>

```

void calculo_voltaje(void);    /* Cálculo del voltaje en voltios. */
void calculo_intensidad(void); /* Cálculo de intensidad en amperios. */
void calculo_resistencia(void); /* Cálculo de resistencia en ohmios. */
void calculo_ley_ohm (void (*funcion)(void));
    /* Ejecuta la función que recibe como parámetro */

void (*ley_ohm[3])(void) =
{
    calculo_voltaje,
    calculo_intensidad,
    calculo_resistencia
};

main()
{
    char eleccion;    /* Elección del usuario. */
    float voltaje;    /* Voltaje del circuito en voltios. */
    float intensidad; /* Intensidad del circuito en amperios. */
    float resistencia; /* Resistencia del circuito en ohmios. */

    printf("\n\nElija la forma de la ley de Ohm que corresponda:\n");
    printf("A) Voltaje B) Intensidad C) Resistencia\n");
    printf("Su eleccion (A, B, or C) => ");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A' : calculo_ley_ohm(*ley_ohm[0]);
                    break;
        case 'B' : calculo_ley_ohm(*ley_ohm[1]);
                    break;
        case 'C' : calculo_ley_ohm(*ley_ohm[2]);
                    break;
    } /* Fin del switch. */
}

void calculo_voltaje(void) /* Cálculo del voltaje en voltios. */
{
    float voltaje;    /* Voltaje del circuito en voltios. */
    float intensidad; /* Intensidad del circuito en amperios. */
    float resistencia; /* Resistencia del circuito en ohmios. */

    printf("Introduzca la intensidad en amperios => ");
    scanf("%f",&intensidad);
    printf("Valor de la resistencia en ohmios => ");
    scanf("%f",&resistencia);
    voltaje = intensidad * resistencia;
    printf("El voltaje es %f voltios.",voltaje);
}

void calculo_intensidad(void) /* Cálculo de la intensidad en amperios. */
{

```

```

float voltaje;      /* Voltaje del circuito en voltios.      */
float intensidad;   /* Intensidad del circuito en amperios. */
float resistencia;  /* Resistencia del circuito en ohmios. */

printf("Introduzca el voltaje en voltios => ");
scanf("%f",&voltaje);
printf("Valor de la resistencia en ohmios => ");
scanf("%f",&resistencia);
intensidad = voltaje / resistencia;
printf("La intensidad es %f amperios.",intensidad);
}
void calculo_resistencia(void) /* Cálculo de la resistencia en ohmios.*/
{
    float voltaje;      /* Voltaje del circuito en voltios.      */
    float intensidad;   /* Intensidad del circuito en amperios. */
    float resistencia;  /* Resistencia del circuito en ohmios. */

    printf("Introduzca el voltaje en voltios => ");
    scanf("%f",&voltaje);
    printf("Valor de la intensidad en amperios => ");
    scanf("%f",&intensidad);
    resistencia = voltaje / intensidad;
    printf("La resistencia es %f ohmios.",resistencia);
}
void calculo_ley_ohm (void (*funcion)(void))
{
    funcion();
}

```

Conclusión

En esta sección se ha presentado el uso de funciones como argumentos. Ahora usted puede tener una idea de lo que hace falta para pasar funciones como argumentos y de para qué sirven. En la siguiente sección se aprenderá cómo usar funciones con un número variable de argumentos. Compruebe su nivel de comprensión de esta sección haciendo los siguientes ejercicios de repaso.

Repaso de la Sección 10.2

1. ¿Para qué sirve pasar una función como argumento?
2. Defina por qué son importantes los paréntesis que rodean al argumento a función declarado.
3. ¿Se puede usar cualquier función como argumento de otra función?
4. Defina el prototipo de una función `func` que recibe como parámetros un `int` y una función compatible con la función `int sumar (int numero)`;

10.3. FUNCIONES CON NÚMERO VARIABLE DE ARGUMENTOS

Presentación

En esta sección se muestra como definir prototipos de función con un número variable de argumentos y cómo usar dichas funciones. Este concepto es importante cuando se quiere programar alguna función que ejecuta algo sobre un número de argumentos no conocido hasta el momento de la ejecución. El efecto de la llamada a una de estas funciones será desconocido hasta el momento de su ejecución.

Operaciones sobre un número variable de operandos

Todas las funciones que se han visto hasta el momento tienen un número fijo de argumentos. Sin embargo, en C existe la posibilidad de declarar funciones con un número de argumentos variable, número que no es conocido hasta que no se llama a la función. El caso más conocido en C de este tipo de funciones es la función `printf`, que puede imprimir un número variable de argumentos con el formato que se especifique en el primer argumento de la función.

Las mismas utilidades usadas para implementar la función `printf` pueden ser usadas por el programador para implementar sus propias funciones con un número variable de argumentos. Para ello, es necesario usar las funciones y macros predefinidas en el archivo de cabecera `<stdarg.h>`, las cuáles proporcionan facilidades para recorrer la lista de argumentos variables aun cuando sean de tipo y número desconocido.

El Programa 10.3 ilustra el uso de funciones con un número variable de argumentos. Para ello, se han definido dos funciones

```
int sumar (int contador, ...);
int multiplicar (int contador, ...);
```

que admiten un número variable de argumentos y hacen operaciones de suma y multiplicación sobre dichos argumentos.

Si se observa con atención los prototipos de estas funciones, se verá que tienen un detalle característico: *los puntos suspensivos*. Dichos puntos, que deben aparecer obligatoriamente al final de la lista de argumentos conocidos, indican que la función puede tener además un número variable de argumentos de un tipo que todavía es desconocido. Para acceder a dicha lista de argumentos dentro de la función, se usa un nuevo tipo de C llamado `va_list` y tres funciones (macros) que operan sobre objetos de tipo lista de argumentos: `va_start`, `va_arg` y `va_end` (todas ellas definidas en `<stdarg.h>`).

El tipo `va_list` depende de la máquina en que se ejecuta la aplicación, pero habitualmente se define como un vector o un puntero. Para trabajar con los argumentos, es necesario declarar una variable de este tipo dentro de la función con número de argumentos variable:

```
va_list args; /* lista de argumentos */
```

Una vez declarada esa variable, y antes de hacer nada con ella, es necesario iniciarla con la lista (o su dirección) de argumentos variables. Dicha iniciación se hace usando la función predefinida `va_start` de la siguiente manera:

```
va_start (args, contador);
```

Es decir, se guarda en la variable `args` la dirección de comienzo de la lista de argumentos variables, o lo que es lo mismo, la posición del último argumento conocido (en este caso `contador`). Observe que esto obliga a que en cualquier función con número de argumentos variable exista al menos un argumento con nombre en la definición de la función. En caso contrario, sería imposible obtener la dirección de comienzo para la lista de los restantes argumentos.

Una vez iniciada la variable con la lista de argumentos, es necesario recorrer dicha lista para extraer todos los argumentos existentes. Para ello se usa la función predefinida `va_arg`, función que extrae el siguiente argumento de la lista. El prototipo de dicha función:

```
type va_arg (va_list args, type);
```

exige conocer el tipo del argumento que se espera obtener de la lista para poder extraerlo. Por esta razón, entre otras, se usan los especificadores de formato en la función `printf`. Así, en el Programa 10.3, donde todos los argumentos son enteros, un argumento se extrae de la siguiente manera:

```
arg = va_arg(args, int);
```

Para extraer todos los elementos de la lista, es necesario recorrerla secuencialmente y llamar a la función `va_arg` en cada paso. Finalmente, cuando todos los argumentos han sido extraídos, es necesario llamar a la función `va_end` para limpiar de la pila de llamada de la función todos los elementos variables. Esta limpieza es necesaria para asegurar que la función con número de argumentos variables termina normalmente y vuelve a la dirección de retorno incluida en la pila cuando se la llamó. La función de limpieza, se llama en el Programa 10.3 de la siguiente manera:

```
va_end(args);
```

Obsérvese que la función (macro) `va_arg` sólo funciona correctamente cuando se conoce el tipo de dato a extraer y si dicho tipo se puede convertir fácilmente a un puntero añadiendo un `*`. Los tipos sencillos, como `char` o `int`, funcionan sin ninguna dificultad. Si los argumentos son objetos más complejos, como vectores o estructuras, hay que pasarlos de forma que la conversión a puntero sea correcta. Por ejemplo, para un vector de enteros, `int[]`, la forma correcta de uso es `int *` y no `int[]`, ya que `*int[]` no es un tipo de datos válido en C.

El número de argumentos en funciones de este tipo debe ser igual o mayor que el especificado en los parámetros con nombre. Si se sobrepasa el número definido, los parámetros excedentes no serán tenidos en cuenta. Si hay menos parámetros de los indicados se pueden obtener resultados impredecibles dependiendo de la implementación real de `va_arg` y `va_end`.

El Programa 10.3 muestra un ejemplo completo del uso de funciones con un número variable de argumentos.

Programa 10.3

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

int sumar (int contador, ...);
/* Suma los números que indica contador*/
int multiplicar (int contador, ...);
/* Multiplica los números que indica contador */

main()
{
    int solucion; /* Solucion de la operacion */
    char eleccion; /* Operacion elegida por el usuario */
    int parametros; /* Indica cuantos numeros participan */
    int i;

    printf("\n\nElija la operacion que corresponda:\n");
    printf("A] Sumar B] Multiplicar \n ");
    printf("Su eleccion (A, o B) => ");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A' :
            printf ("Cuantos numeros pares quiere sumar a 2? ");
            printf("Su eleccion entre 1 y 3 =>");
            scanf("%d",&parametros);
            switch(parametros)
            {
                case 1: solucion = sumar(parametros,4);
                    break;
                case 2: solucion = sumar(parametros,4,6);
                    break;
                case 3: solucion = sumar(parametros,4,6,8);
                    break;
                default: printf("Eleccion erronea \n");
                    break;
            } /* Fin del switch. */
            printf("\n La suma es => %d \n", solucion);
            break;
        case 'B' :
            printf ("Cuantos numeros pares multiplicar por 2? ");
            printf("Su eleccion entre 1 y 3 =>");
            scanf("%d",&parametros);
            switch(parametros)
            {
                case 1: solucion= multiplicar(parametros,4);
                    break;
```



```

        case 2: solucion= multiplicar(parametros,4,6);
                break;
        case 3: solucion= multiplicar(parametros,4,6,8);
                break;
        default: printf("Eleccion erronea \n");
                break;
    } /* Fin del switch. */
    printf("\n La multiplicacion es => %d \n", solucion);
    break;
default:
    printf ("Operacion invalida \n");
    break;
} /* Fin del switch. */
}
int sumar (int contador, ...)
/* Suma los números que indica contador*/
{
    va_list  args; /* lista de argumentos */
    int  arg;      /* argumento individual */
    int total = 2; /* variable de almacenamiento de resultados */
    int i;

    va_start (args, contador);
    for (i=0; i<contador; i++)
    {
        arg = va_arg(args, int);
        total = total + arg;
    }
    va_end(args);
    return(total);
}
int multiplicar (int contador, ...)
/* Multiplica los números que indica contador */
{
    va_list args; /* lista de argumentos */
    int  arg;      /* argumento individual */
    int total = 2; /* variable de almacenamiento de resultados */
    int i;

    va_start (args, contador);
    for (i=0; i<contador; i++)
    {
        arg = va_arg(args, int);
        total = total * arg;
    }
    va_end(args);
    return(total);
}

```

Análisis del programa

El Programa 10.3 permite calcular dos series numéricas basadas en sumar o multiplicar por 2, respectivamente, un número variable de términos pares. Para ello, se pide al usuario que elija el tipo de serie a calcular

```
printf("\n\nElija la operacion que corresponda:\n");
printf("A] Sumar B] Multiplicar \n ");
printf("Su eleccion (A, o B) => ");
scanf("%c",&eleccion);
```

y el número de términos que hay en la serie respectiva:

```
printf ("Cuantos numeros pares quiere sumar a 2? ");
printf("Su eleccion entre 1 y 3 =>");
scanf("%d",&parametros);
```

Una vez conocido el número de terminos de la serie, se usa este valor para indicar a las funciones con número de parámetros variable cuántos términos deben introducir en el cálculo. Según la elección del usuario se hacen distintas llamadas a función:

```
switch(parametros)
{
    case 1: solucion = sumar(parametros,4);
            break;
    case 2: solucion = sumar(parametros,4,6);
            break;
    case 3: solucion = sumar(parametros,4,6,8);
            break;
    default: printf("Eleccion erronea \n");
            break;
} /* Fin del switch. */
```

Una vez realizada la correspondiente llamada a función, se calcula la serie, se devuelve el resultado y se imprime el total calculado en la serie.

Conclusión

En esta sección se ha presentado el uso de funciones con un número variable de argumentos. Ahora usted puede tener una idea de lo que hace falta para usar dichas funciones y de para qué sirven. En la siguiente sección se aprenderá a programar aplicaciones con varios módulos independientes. Compruebe su nivel de comprensión de esta sección haciendo los siguientes ejercicios de repaso.

Repaso de la Sección 10.3

1. ¿Para qué sirve una función con un número variable de argumentos?
2. Defina por qué es importante el primer parámetro de una función de este tipo.

3. ¿Se puede usar cualquier argumento dentro de la parte variable del prototipo?
4. ¿Se pueden usar distintos tipos de argumentos dentro de la parte variable del prototipo?

10.4. APLICACIONES MODULARES

Presentación

En esta sección se muestra como construir un programa compuesto por varias unidades distintas, cada una de las cuales está almacenada en un archivo fuente. Así mismo se verá cómo se puede referenciar las variables y funciones de una unidad de programa desde otra, y cómo agrupar los elementos que exporta una unidad de programa en un archivo de cabecera que pueda ser usado desde otras unidades de programa. Este concepto, denominado programación modular, es importante cuando se quiere programar aplicaciones grandes o aplicaciones cuyos elementos se pueden agrupar con criterio lógico en módulos distintos.

Uso de varias unidades de programa

En los programas desarrollados hasta el momento en este texto, siempre se ha supuesto que todo el programa residía en un único archivo fuente de C. Sin embargo, C proporciona un soporte muy amplio para desarrollar el concepto de programación modular, según el cual no es necesario que todo el código fuente esté en un único archivo, sino que dicho código puede repartirse en distintos archivos fuente. Estos módulos separados se corresponden con la idea de unidad de programación.

La posibilidad de construir una aplicación en C como un conjunto de unidades de programa separadas, presenta varias ventajas:

- Permite hacer frente a la complejidad de un problema grande mediante la división en partes más pequeñas y aplicando criterios de caja negra.
- Permite usar tipos de datos abstractos, cada uno de los cuales se agrupa, junto con sus operaciones asociadas, en una unidad de programa.
- Permite compilar cada unidad de programa por separado, lo que las hace más fáciles de programar, depurar y probar.
- Permite ocultar dentro de una unidad de programa aquella información que no debería ser vista fuera del módulo, de forma que desde fuera del módulo sólo se puede usar la funcionalidad exportada por el mismo.

El lenguaje C nos permite aplicar principios de diseño modular, construyendo aplicaciones con distintos niveles de complejidad. No es un lenguaje perfecto para la aplicación estricta de reglas de la programación modular tales como el principio de ocultación o el tipado fuerte, pero proporciona facilidades para expresar objetos y las operaciones sobre ellos en el mundo real. Además, ofrece facilidades de empaquetamiento con las que se pueden construir aplicaciones complejas mediante su división en entidades abstractas cuyas operaciones son usadas por otros elementos de la aplicación. El empaquetamiento de programas en distintas unidades es la piedra angular de la abstracción, la ocultación de información, la visibilidad y la localidad.

Cuando se construyen distintas unidades de programa, los elementos exportables de cada unidad deben ser especificados claramente, siendo un principio de programación importante, que los nombres de los objetos de un programa reflejen su contenido y sean visibles solamente a aquellas partes que los necesiten. Esto se logra en C mediante las reglas de ámbito, que indican dónde y cómo serán visibles las declaraciones realizadas en C.

La declaración de una variable en un bloque (automática o local) restringe su ámbito de visibilidad a ese bloque. La memoria asociada a estas variables se asigna dinámicamente en tiempo de ejecución y se libera cuando termina la ejecución del bloque. El anidamiento de bloques permite que las variables locales puedan ser declaradas cerca de donde se necesitan, pudiendo los bloques más internos acceder a variables de bloques más externos. Esta localidad multinivel contribuye mucho a la legibilidad de los programas, ya que permite declarar nombres ajustados a la aplicación y evita la tentación de reutilizar múltiples veces el mismo nombre con significados distintos. Estas prácticas, habituales en lenguajes sin estructura de bloques, son una causa corriente de errores de programación.

Para ver un ejemplo de descomposición en unidades de programa de una aplicación, considere el Programa 10.1. Dicho programa puede dividirse fácilmente en dos unidades de programa `principal.c` y `ley_ohm.c`. La función `main` reside en la unidad de programa `principal.c`. Las utilidades de cálculo de la ley de Ohm están en el módulo `ley_ohm.c`.

Las tres funciones de cálculo de la ley de Ohm son exportadas desde este módulo e importadas dentro del programa cliente en la unidad de programación `principal.c`. El módulo `principal` `main` ejecuta un bucle infinito que lee el código de utilidad y llama a función adecuada de la unidad de utilidad, haciendo referencias externas a sus funciones. La codificación de ambas unidades de programa se muestra en el Programa 10.4.

Programa 10.4 `/* ARCHIVO principal.c */`

```
#include <stdio.h>

extern void calculo_voltaje(void);
extern void calculo_intensidad(void);
extern void calculo_resistencia(void);

main()
{
    char eleccion;      /* Elección del usuario.          */
    float voltaje;      /* Voltaje del circuito en voltios.      */
    float intensidad;   /* Intensidad del circuito en amperios.  */
    float resistencia;  /* Resistencia del circuito en ohmios.   */
    int i;

    do
    {
```

```

printf("\n\nElija la forma de la ley de Ohm que");
printf(" corresponda:\n");
printf("A) Voltaje B) Intensidad C) Resistencia S) Salir\n");
printf("Su eleccion (A, B, C, o S) => ");
scanf("%c",&eleccion);
switch(eleccion)
{
case 'A' : (*ley_ohm[0])();
break;
case 'B' : (*ley_ohm[1])();
break;
case 'C' : (*ley_ohm[2])();
break;
} /* Fin del switch. */
} while (eleccion != 'S');
}

/* ARCHIVO ley_ohm.c */
#include <stdio.h>

float multiplicar(float a, float b); /* Multiplica a * b */
float dividir(float a, float b); /* Dividir a / b */

int veces_ley_ohm = 0; /* Veces que se ha llamado a este modulo */

void calculo_voltaje(void) /* Cálculo del voltaje en voltios. */
{
float voltaje; /* Voltaje del circuito en voltios. */
float intensidad; /* Intensidad del circuito en amperios. */
float resistencia; /* Resistencia del circuito en ohmios. */
static int veces_vol; /* Veces que se ha calculado voltaje */

printf("Introduzca la intensidad en amperios => ");
scanf("%f",&intensidad);
printf("Valor de la resistencia en ohmios => ");
scanf("%f",&resistencia);
voltaje = multiplicar(intensidad, resistencia);
printf("El voltaje es %f voltios.",voltaje);

veces_ley_ohm ++;
veces_vol ++;
printf("Llamadas al modulo: %d para voltaje %d \n",
veces_ley_ohm, veces_vol);
}

void calculo_intensidad(void) /* Cálculo de la intensidad en amperios. */
{
float voltaje; /* Voltaje del circuito en voltios. */
float intensidad; /* Intensidad del circuito en amperios. */
float resistencia; /* Resistencia del circuito en ohmios. */

```

```

    static int veces_int; /* Veces que se ha calculado intensidad */

    printf("Introduzca el voltaje en voltios => ");
    scanf("%f",&voltaje);
    printf("Valor de la resistencia en ohmios => ");
    scanf("%f",&resistencia);
    intensidad = dividir(voltaje , resistencia);
    printf("La intensidad es %f amperios.",intensidad);

    veces_ley_ohm ++;
    veces_int ++;
    printf("Llamadas al modulo: %d para intensidad %d \n",
           veces_ley_ohm, veces_int);
}

void calculo_resistencia(void) /* Cálculo de la resistencia en ohmios. */
{
    float voltaje;      /* Voltaje del circuito en voltios.      */
    float intensidad;   /* Intensidad del circuito en amperios. */
    float resistencia;  /* Resistencia del circuito en ohmios. */
    static int veces_res; /* Veces que se ha calculado resistencia */

    printf("Introduzca el voltaje en voltios => ");
    scanf("%f",&voltaje);
    printf("Valor de la intensidad en amperios => ");
    scanf("%f",&intensidad);
    resistencia = dividir(voltaje, intensidad);
    printf("La resistencia es %f ohmios.",resistencia);

    veces_ley_ohm ++;
    veces_res ++;
    printf("Llamadas al modulo: %d para resistencia %d \n",
           veces_ley_ohm, veces_res);
}

float multiplicar(float a, float b) /* Multiplica a * b */
{
    return (a*b);
}

float dividir(float a, float b)      /* Dividir a / b */
{
    if (b == 0)
        return (-1);
    else
        return (a/b);
}

```

La estructura de bloques no es en sí un mecanismo adecuado para controlar totalmente la visibilidad de los nombres. Es necesario que exista algún mecanismo adicional para indicar que variables locales pueden ser accedidas desde fuera de la unidad de

programa donde son declaradas. Además, debería ser posible especificar que una variable local a una unidad de programa mantenga su valor desde una activación a la siguiente. Las construcciones de C que permiten lograr estos objetivos tienen que ver con las clases de declaración, ámbito y enlace de objetos. La clase de declaración determina el tiempo de vida de un objeto. El ámbito indica la región del programa en que es conocido. El enlace indica si el mismo nombre se refiere al mismo o a distintos objetos en el mismo ámbito. A continuación estudiaremos cada caso siguiendo el ejemplo del Programa 10.4.

Clases de declaración y ámbito

El estándar ANSI C define solamente dos clases de almacenamiento:

- *Automático*: variables locales al objeto en que declaran (función o módulo) y que perduran solamente mientras dura una iniciación del objeto.
- *Estático*: variables locales o globales cuyo valor persiste entre las distintas iniciaciones de un objeto. Se declaran con la palabra reservada `static`.

Además, en cuanto al ámbito, las variables pueden ser locales o externas. El ámbito de las declaraciones externas se extiende hasta el fin de la unidad de programa. La palabra reservada para definir objetos externos es `extern`, aunque puede ser omitida en algunos compiladores.

Las variables declaradas en el cuerpo de una función o unidad de programa son, por defecto, de clase `auto`. Estas variables son internas al objeto y aparecen en el programa cuando se utiliza dicho objeto, desapareciendo cuando se sale del ámbito del mismo. Los argumentos de las funciones se procesan de forma similar. Ejemplos de variables automáticas en el Programa 10.4 son:

```
float voltaje;      /* Voltaje del circuito en voltios.      */
float intensidad;  /* Intensidad del circuito en amperios. */
float resistencia; /* Resistencia del circuito en ohmios.  */
```

```
int veces_ley_ohm;
```

Estas variables se inician de forma automática cuando se llama al objeto al que pertenecen y desaparecen con él. Es pues interesante definir a que objeto pertenecen, es decir su ámbito, para saber cuando aparecen y desaparecen y donde son visibles. Las tres primeras (`voltaje`, `intensidad`, `resistencia`) se definen dentro de funciones y sólo duran lo que la llamada a dicha función. Ahora bien, la variable `veces_ley_ohm` se declara en la cabecera de la unidad de programa `ley_ohm` por lo que dura mientras dicha unidad de programa esté activa, es decir, todo el tiempo de ejecución. Por tanto esta variable, es local al módulo, pero global para todos sus elementos.

Las variables estáticas proporcionan una nueva forma de almacenamiento que, independientemente de su ámbito, pervive al objeto en que son declaradas. Proporcionan almacenamiento privado y permanente para una función u objeto y sirven como *memoria* del objeto entre distintas ejecuciones del mismo. Como las demás variables,

pueden ser iniciadas con un valor constante cuando son declaradas, pero a diferencia de ellas, esa iniciación sólo tiene lugar la primera vez que se llama al objeto y se usa la variable. Ese es el caso de las variables

```
static int veces_volt; /* Veces que se ha calculado voltaje */
static int veces_int; /* Veces que se ha calculado intensidad */
static int veces_res; /* Veces que se ha calculado resistencia */
```

que se usan para controlar las veces que se ha llamado a cada función del módulo desde el principio del programa. Así, la ejecución del Programa 10.4 durante 4 bucles, en el cuarto de los cuales se llama a calcular resistencia sería:

Llamadas al modulo: 4 para resistencia 1

Para comprobar mejor la diferencia entre los tipos de variables anteriores, ejecute varias veces el Programa 10.4.

Uso de recursos externos

Un programa C puede tener una colección de objetos externos, adjetivo que define a los objetos que no son argumentos ni variables automáticas definidas como funciones, que se definen como internos. Los elementos internos están definidos fuera de las funciones, siendo las propias funciones objetos externos ya que en C no puede definirse una función dentro de otra.

Los objetos externos de un programa son referenciados como tal por el compilador y el montador, por lo que cuando una unidad de programa referencia un objeto externo, debe indicarlo expresamente mediante una *declaración de referencia externa*. La declaración informa al compilador de que una función se va a llamar de forma separada a su definición, de que devuelve un resultado de un determinado tipo y de que la definición de la función está en otra unidad de programa. Estas declaraciones son similares a la declaración del prototipo de la variable o función referenciada, pero llevan delante la palabra reservada *extern*.

Considere el módulo principal del Programa 10.4. Dicho módulo usa tres referencias externas, las de las funciones del módulo que calcula la ley de Ohm.

```
extern void calculo_voltaje(void);
extern void calculo_intensidad(void);
extern void calculo_resistencia(void);
```

La palabra reservada *extern* indica que esas funciones son exportadas por su módulo e importadas por el módulo principal de la aplicación. Esa declaración puede aplicarse a funciones y variables, sirviendo en este último caso para declarar variables globales de forma explícita, ya que pueden ser referenciadas desde cualquier otra unidad del programa. Debido a ello, proporcionan una forma alternativa a la comunicación de datos entre funciones. Cualquier función declarada después de la variable, o que importe la misma, puede acceder a dicha variable a través de su nombre. Sin embargo, no debe usarse la aparentemente fácil solución de declarar todos las varia-

bles del programa como un objeto externo. Aunque puede parecer que esto facilita la comunicación dentro del programa, hace que los programas sean más difíciles de entender y modificar debido a que hay muchas funciones modificando los valores de las variables. Además, la autonomía y la generalidad de las funciones que sólo dependen de sus argumentos y sus variables internas, se pierde totalmente si se utiliza esta técnica. Con ella, las funciones quedan inextricablemente ligadas a las variables globales externas. Otro problema adicional puede venir de los conflictos con variables internas a funciones que usan el mismo nombre que una externa global, caso en que prevalece siempre el valor de la variable local.

La iniciación de variables externas globales es una buena práctica de programación para evitar fallos en la inicialización automática de las mismas, que según el estándar ANSI C debe ser 0. Para que no haya conflictos de nombres, una variable externa global debe ser definida únicamente una vez. En esa instancia de la variable no aparece la referencia `extern` asociada a la misma. En todas las demás dicha referencia debe estar presente.

Archivos de cabecera

En toda unidad de un programa C se puede especificar lo que se desee exportar para facilitar el uso de estos recursos desde fuera y favorecer la ocultación de información que no debe verse fuera del módulo. Para ello, se construye un *archivo de cabecera* que contenga todas las declaraciones externas que sean necesarias. En el caso del Programa 10.4, se puede definir un archivo de cabecera `ley_ohm.h` como el siguiente:

```
/* ARCHIVO ley_ohm.h
** Archivo de cabecera que proporciona la especificacion de
** las funciones que calculan la ley de Ohm.
*/
extern void calculo_voltaje(void);
extern void calculo_intensidad(void);
extern void calculo_resistencia(void);
```

Las referencias externas a estas funciones del módulo `principal.c` pueden sustituirse por la inclusión (con la palabra reservada `include`) del archivo anterior, de la siguiente manera:

```
/* ARCHIVO principal.c */
#include <stdio.h>
#include "ley_ohm.h";

main()
{
    char eleccion;      /* Elección del usuario.          */
    float voltaje;       /* Voltaje del circuito en voltios.      */
    float intensidad;    /* Intensidad del circuito en amperios.  */
    float resistencia;   /* Resistencia del circuito en ohmios.   */
}
```

```

int i;

do
{
    printf("\n\nElija la forma de la ley de Ohm que");
    printf(" corresponda:\n");
    printf("A] Voltaje B] Intensidad C] Resistencia S] Salir\n");
    printf("Su eleccion (A, B, C, o S) => ");
    scanf("%c",&eleccion);
    switch(eleccion)
    {
        case 'A' : (*ley_ohm[0])();
                    break;
        case 'B' : (*ley_ohm[1])();
                    break;
        case 'C' : (*ley_ohm[2])();
                    break;
    } /* Fin del switch. */
} while (eleccion != 'S');
}

```

Observe que el archivo de cabecera `ley_ohm.h` se ha incluido en el nivel más externo del módulo donde se va a usar. Esto significa que las referencias externas son de ámbito global al módulo y pueden ser usadas en cualquier parte del mismo. Esta técnica no es general, ya que un archivo de cabecera puede incluirse en cualquier parte de un programa C donde comience un bloque de programación (módulos, funciones, otros archivos de cabecera, etc.).

Observe también que en el ejemplo anterior se han incluido dos archivos de cabecera usando dos definiciones distintas:

```

#include <stdio.h>
#include "ley_ohm.h"

```

El primero `stdio.h` es un archivo predefinido del entorno C, cosa que se indica cuando se incluye poniendo su nombre entre `< y >`. Estos archivos se buscan dentro de las bibliotecas que proporciona el compilador (habitualmente en los directorios `/usr/include` o `/usr/include/sys`). El segundo archivo, `ley_ohm.h`, es un archivo construido por el programador de la aplicación, lo que se indica cuando se incluye poniendo su nombre entre `" y "`. Dichos archivos se buscan en el directorio de trabajo o en el camino absoluto que se especifique en la sentencia `include`.

Como conclusión se observa que un programa C se puede dividir en varios archivos fuente cuyos recursos exportados deberían aparecer en sus respectivos archivos de cabecera. Con esta técnica se pueden exportar funciones, variables y definiciones a través de toda la aplicación, lo que origina un grave problema de posible existencia de declaraciones y definiciones compartidas. Puesto que C no ayuda a resolver este problema, se puede tratar de paliarlo usando la siguiente heurística en la construcción de archivos de cabecera:

- Un archivo de cabecera puede contener:

```

definiciones de macros      #define MES "Enero"
declaraciones de variables  extern int dia;
declaraciones de función    extern void f(void);
otras directivas de inclusión #include "ley_ohm.h"
definiciones de tipo        typedef int mes;
comentarios                 /* Heurística para archivo de cabecera */

```

- Un archivo de cabecera nunca debe tener:

```

definiciones de variables    int dia;
definiciones de funciones    void f(void);

```

Otro mecanismo que se puede usar para lograr que dos variables con el mismo nombre se refieran a objetos distintos cuando están en distintos módulos es *ligarlas* al módulo en que están definidas. Esta ligazón se consigue usando la palabra reservada *static* cuando se definen las variables. Frente a las variables externas globales, que tienen un *enlace externo* al módulo en que están declaradas, una variable de tipo estático tiene un *enlace interno* al módulo en que está declarada, no siendo visible desde fuera del mismo. De esta forma, dos variables que se llamen igual y sean globales a sendos módulos del programa no originarán problemas puesto que se referirán a distintos objetos de cada módulo. Por ejemplo, las definiciones:

```

/* archiv1.c */
static int      mes = 5;

/* archiv2.c */
static int      mes = 3;

```

no originan problemas porque ambas variables se ven como objetos internos a cada módulo y por tanto distintos. Sin embargo, las declaraciones:

```

/* archiv1.c */
int      mes = 5;

/* archiv2.c */
int      mes = 3;

```

darían un error de compilación por redeclaración de un objeto ya declarado. Esta forma de usar la palabra reservada *static* permite ocultar información a los usuarios externos del módulo. Por eso, algunos autores recomiendan definir las dos macros siguientes:

```

#define PUBLIC extern
#define PRIVATE static

```

y usarlas para calificar las funciones exportables y ocultas, en lugar de poner específicamente las palabras reservadas de C. Estas macros facilitan la legibilidad de los programas con varios módulos.

Conclusión

En esta sección se han visto varios aspectos interesantes acerca de la construcción de aplicaciones modulares en C. Dichos aspectos incluyen la división de una aplicación en distintas unidades de programa, el ámbito y el tiempo de vida de las variables declaradas en los distintos módulos y el uso de archivos de cabecera para exportar objetos fuera de una unidad de programa. Con todo ello se pueden especificar en C tipos abstractos de datos y aprovechar los distintos niveles de abstracción de las aplicaciones para llevar a cabo un desarrollo más eficaz y ordenado de las mismas.

Compruebe su nivel de comprensión de esta sección haciendo los ejercicios de repaso.

Repaso de la Sección 10.4

1. Explique lo que significan los siguientes términos:
 - (a) abstracción
 - (b) unidad de programa
 - (c) exportable
 - (c) reglas de ámbito
2. ¿Cuál es la diferencia entre la definición de una variable y su referencia?
3. Si dos variables globales tienen el mismo nombre, ¿se produciría un error de compilación?
4. En que se distingue si se incluye un archivo de cabecera de las bibliotecas de C o de usuario.

10.5. LA UTILIDAD `make`

Presentación

En esta sección se verá como compilar un programa escrito en C para obtener, a partir del código fuente, un programa que se pueda ejecutar. Además se verá como compilar de forma semiautomática aplicaciones formadas por varios archivos fuente usando la utilidad `make`. Con ello, se considera que el programador ya está listo para desarrollar sus propias aplicaciones modulares, cualquiera que sea el tamaño de ésta, igual que lo haría cualquier profesional que desarrolle programas tecnológicos en C.

Compilación de programas en C

Un programa escrito en C está compuesto por, al menos, un archivo fuente (`*.c`) y archivos de tipo cabecera (`*.h`). En UNIX, los archivos se compilan con la orden `cc`, que incluye la secuencia de pasos necesarios para compilar y enlazar el programa hasta llegar al archivo ejecutable. La forma general de una orden `cc` es:

```
cc opcion archivo1 archivo2 ...
```

donde las opciones se distinguen por estar precedidas por el símbolo `(-)` y afectan el comportamiento de la orden `cc`.

Un programa, como el 10.3, compuesto por un único archivo fuente (`p10_3.c`) se compila con la siguiente orden:

```
cc p10_3.c
```

Todos los errores que ocurran se muestran en la salida estándar del sistema (habitualmente la pantalla). En caso de que no haya errores, esa orden genera un archivo ejecutable cuyo nombre por defecto es `a.out` en todos los compiladores C para UNIX. El nombre del archivo ejecutable final puede especificarse con la opción `-o` de la orden de la siguiente manera:

```
cc p10_3.c -o p10_3
```

Las distintas etapas por las que pasa la orden `cc` son:

1. Fase de compilación. Se crea un archivo objeto (`*.o`) a partir del fuente (`*.c`).
2. Fase de montado. Se crea un archivo ejecutable a partir de los objetos (`*.o`).

La orden se emplea en una u otra etapa indicando las opciones adecuadas. Por ejemplo, con `-c` sólo se ejecuta el paso 1 y se obtienen los archivos objeto pero no el ejecutable. Además permite añadir otras opciones como inclusión de bibliotecas (`-l`), definición de macros de compilación (`-D`), inclusión de información de depuración (`-g`), etc. La descripción exacta de las opciones de la orden `cc` puede verse en cualquier sistema UNIX ejecutando la orden `man cc`.

Si la aplicación incluye varios módulos, tal y como ocurre con el Programa 10.4, es necesario incluir en la orden de compilación todos los archivos fuentes que forman la aplicación. La orden para compilar el Programa 10.4 sería:

```
cc principal.c ley_ohm.c -o p10_4
```

Hay tres cosas interesantes a resaltar de esta orden:

- Sólo se genera un ejecutable de la aplicación (`p10_4`), cuyo nombre puede no coincidir con el de ningún archivo fuente.
- Es necesario recompilar todos los módulos de la aplicación cada vez.
- Puede que no todos los módulos tengan las mismas opciones de compilación.

Una forma alternativa de compilación sería compilar cada archivo fuente por separado y, posteriormente, enlazar todos los objetos para obtener el ejecutable. La secuencia de órdenes sería:

```
cc -c principal.c cc
-c ley_ohm.c
cc principal.o ley_ohm.o -o p10_4
```

Obsérvese que en este caso, el ejecutable se obtiene a partir de los objetos previamente compilados por separado. Esta modalidad de compilación nos permite recompilar sólo aquellos módulos de la aplicación que se han modificado y generar posteriormente el ejecutable. Sin embargo, queda el problema de saber qué módulos deben

recompilarse cuando se trabaja con aplicaciones grandes y qué opciones deben aplicarse a cada uno. Para este problema hay dos posibles soluciones:

- Crear archivos de control en los cuales se indique los que hay que recompilar.
- Usar la utilidad `make` presente en todos los sistemas C.

La primera opción es engorrosa y origina muchos problemas con el control de las versiones, por lo que en este texto se estudiará el uso de la utilidad `make` tanto en el sistema operativo UNIX como en MS/DOS.

Cómo funciona la utilidad `make`

Durante el desarrollo de una aplicación en C, es norma habitual dividir el código fuente en diversos archivos atendiendo a distintos criterios (lógica de la aplicación, diseño, tamaño, etc.). La Figura 10.1 ilustra este concepto mostrando los módulos de un programa de elaboración de inventario que está en construcción. Como puede verse en la figura, el archivo ejecutable (INVENT.EXE) está compuesto por tres archivos objeto. Además, uno de los archivos objeto (INVENT.o) usa funciones contenidas en

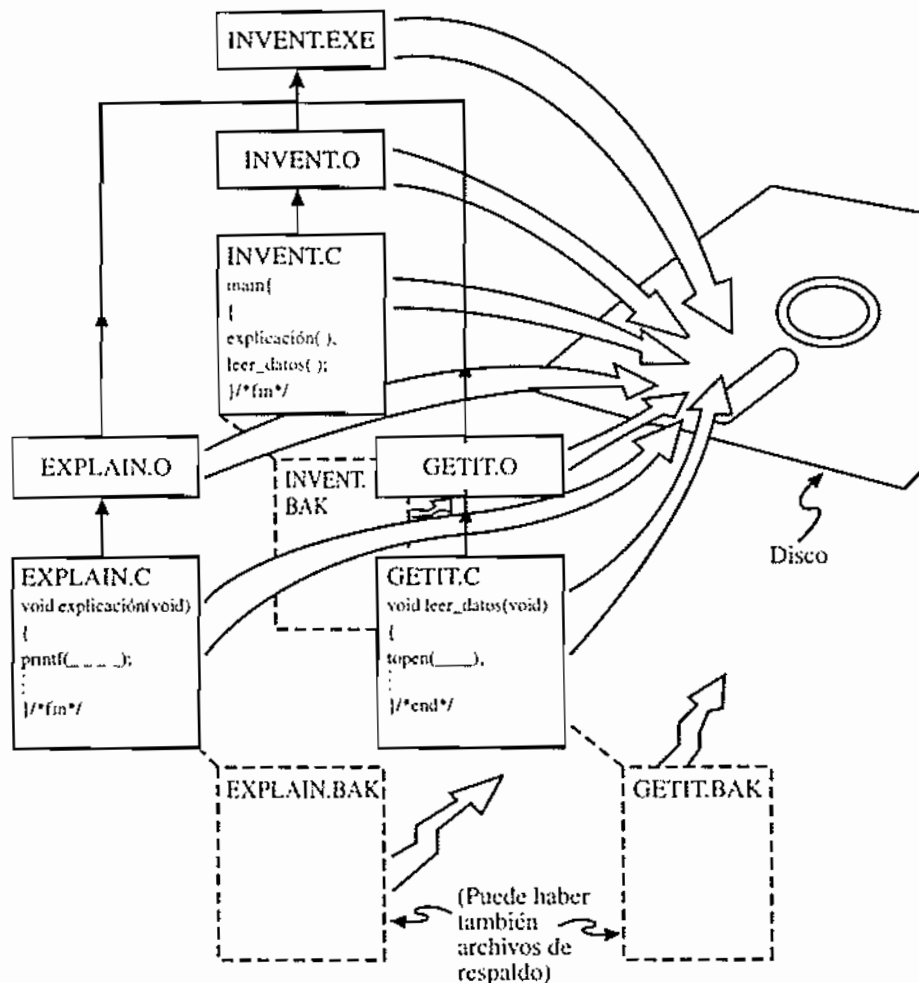


Figura 10.1. Archivos para construir un programa de inventario.

los otros. Los archivos objeto se obtienen compilando sus respectivos archivos fuente (.c). En cambio, el archivo ejecutable se obtiene montando todos los objetos.

Un directorio del sistema de archivos que incluyera dichos archivos sería como el de la Figura 10.2. Dentro de este directorio es importante resaltar que como parte de los datos de un archivo se guardan la fecha de creación y de última modificación del mismo.

Si se quisiera mantener esta aplicación habría que hacerlo manualmente o mediante la utilidad `make`. Esta utilidad mecaniza muchas de las etapas del desarrollo y mantenimiento de un programa, proporcionando mecanismos simples para obtener versiones actualizadas de los programas, por complicados que estos sean. Esto se logra indicando a la utilidad `make` la secuencia de mandatos que crean ciertos archivos y la lista de archivos que necesitan otros archivos para ser actualizados antes de que se hagan las operaciones (lista de dependencias). Una vez especificadas las dependencias entre los distintos módulos del programa, cualquier cambio en uno de ellos provocará la creación de una nueva versión por la utilidad `make`, reduciendo al mínimo posible el número de módulos a recompilar para crear el nuevo archivo ejecutable.

La utilidad `make` usa tres fuentes de información: definiciones del usuario, nombres de archivos y fecha de última modificación de los mismos y reglas de construcción predefinidas. El mecanismo fundamental para generar un nuevo ejecutable consiste en examinar las fechas de última actualización de los archivos fuente. Esta fecha se compara con la de la última actualización del archivo objeto correspondiente por la utilidad `make`. Si ninguno de los archivos fuente u objeto ha sido modificado desde la ejecución del último `make`, no se hace nada. En caso contrario, se compilan los archivos que sea necesario y se genera el nuevo ejecutable. En cualquier caso, se actualiza la fecha de los archivos objeto que sea necesario. La Tabla 10.1 muestra algunas de las funciones que desempeña esta potente utilidad.

Tabla 10.1. Principales funciones de la utilidad `make`

Desarrollo de programas	Actualiza automáticamente los archivos ejecutables cuando se modifica cualquier archivo fuente u objeto.
Gestión de biblioteca	Reconstruye automáticamente una biblioteca cuando cambia alguno de los módulos que la componen.
Gestión de red	Copia automáticamente cualquier archivo local que esté en la red cuando cambia la copia maestra.

Nombre archivo	Extensión	Tamaño del archivo	Fecha	Hora
INVENT	EXE	10187	9-12-96	1:15p
INVENT	O	430	9-12-96	1:08p
INVENT	C	642	9-12-96	1:05p
EXPLAIN	O	983	9-12-96	12:45p
EXPLAIN	C	1246	9-12-96	12:36p
GETIT	O	10083	9-12-96	10:23a
GETIT	C	12436	9-12-96	10:15a

Figura 10.2. Directorio de archivos para el programa de inventario.

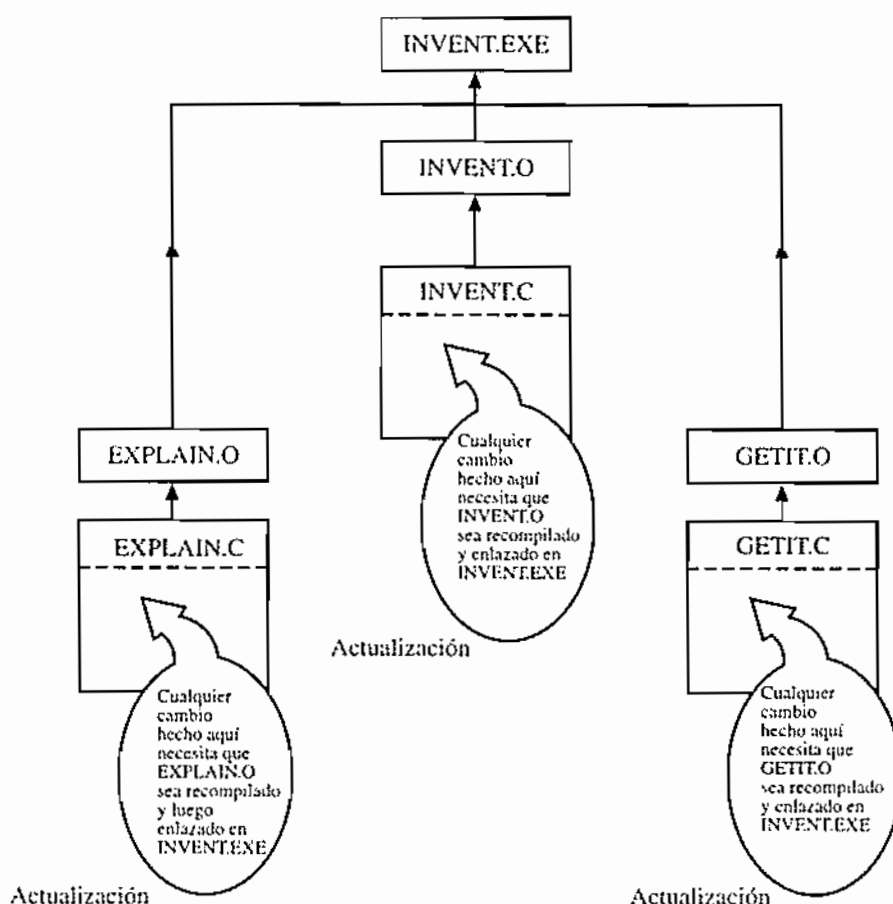


Figura 10.3. Cómo depende un archivo de otro.

La Figura 10.3 muestra las dependencias entre archivos del ejemplo anterior y como el cambio de cualquier archivo fuente necesita la actualización de los archivos que dependen de él. En cualquier momento en que se modifique un archivo fuente, su fecha de última actualización será más moderna que la de su objeto correspondiente y que la del archivo ejecutable de la aplicación. Esto significa que el sistema de archivos debe ser actualizado y que se debe generar una nueva versión de la aplicación. Un ejemplo de aplicación cuyas fechas no son coherentes puede verse en la Figura 10.4.

En la Figura 10.4 se observa que el ejecutable no puede incluir los últimos cambios incorporados a los archivos fuente, puesto que su fecha es más antigua que la de

INVENT	EXE	10187	9-12-96	1:15p	No contiene del cambio más reciente
INVENT	O	430	9-12-96	1:08p	
INVENT	C	642	9-12-96	1:05p	No está activo
EXPLAIN	O	983	9-12-96	12:45p	
EXPLAIN	C	1287	9-12-96	3:06p	Actualización reciente
GETIT	O	10083	9-12-96	10:23a	
GETIT	C	12436	9-12-96	10:15a	

Figura 10.4. Diferencias en las etiquetas de fecha/hora para archivos recientemente cambiados.

INVENT	EXE	10195	9-12-96	3:15p	Actualizado para incluir cambios
INVENT	O	430	9-12-96	1:08p	
INVENT	C	642	9-12-96	1:05p	Ahora activo
EXPLAIN	O	998	9-12-96	3:11p	
EXPLAIN	C	1287	9-12-96	3:06p	Archivo actualizado
GETIT	O	10083	9-12-96	10:23a	
GETIT	C	12436	9-12-96	10:15a	

Figura 10.5. Directorio resultante de archivos actualizados.

algunos de estos. Si se usa este ejecutable, no se obtendrá la última versión de la aplicación, sino una anterior. La ejecución de la utilidad `make` sobre el sistema de la Figura 10.4, daría lugar a una nueva versión del sistema donde todas las fechas de los archivos están coherentes (véase Figura 10.5) y cuya ejecución permitiría observar la última versión del código fuente.

La utilización de la orden `make` exige la creación previa de un archivo de descripción, llamado `makefile`, que contenga la orden que debe ejecutar el mandato `make`, así como las dependencias entre los distintos módulos de la aplicación. La Figura 10.6 ilustra la relación entre el archivo de descripción `makefile` y los archivos de un programa.

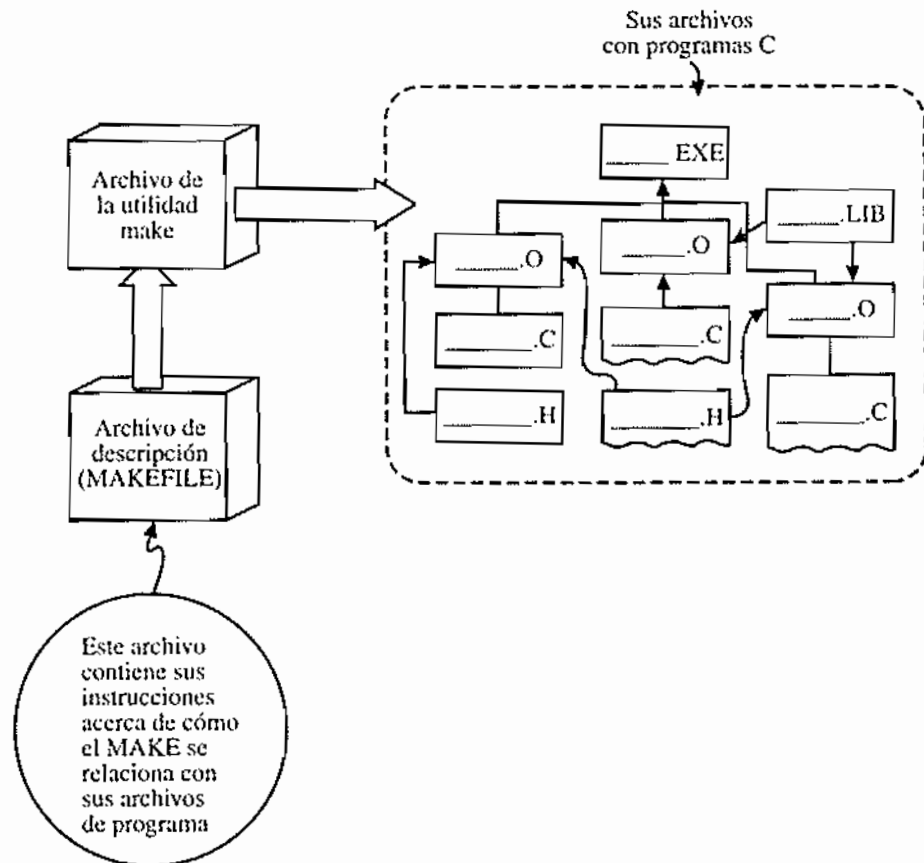


Figura 10.6. Relación entre el archivo descripción MAKE (MAKEFILE) y archivos C en desarrollo y modificación.

El archivo de descripción se crea mediante un editor de textos y es un archivo en código ASCII que contiene las instrucciones en el formato que entiende la utilidad `make`. Un ejemplo de archivo `makefile` para el Programa 10.4 sería el siguiente:

```
#
# Archivo makefile para el programa 10.4
# Esto son comentarios
#
p10_4:    principal.o ley_ohm.o
          cc principal.o ley_ohm.o -lm -o p10_4
#
principal.o ley_ohm.o: stdio.h
```

Este archivo de descripción indica que el ejecutable `p10_4` depende de dos módulos objeto y dice como se genera dicho ejecutable. Además indica que ambos objetos dependen del archivo de cabecera `stdio.h`, por lo que cualquier modificación de este archivo originará una nueva versión del sistema. Además, se especifica que el ejecutable depende de una biblioteca predefinida (en este caso `libm`). A partir de este archivo, la utilidad `make` busca los archivos fuente (`*.c`) asociados a los objetos y usa reglas predefinidas en el sistema para compilarlos y generar el objeto. Estas reglas predefinidas se pueden incluir también en el archivo de descripción, lo cual es especialmente útil en caso de que quieran ser modificadas con distintas opciones de compilación. Con la más básica de estas reglas, el archivo anterior quedaría de la siguiente manera:

```
#
# Archivo makefile para el programa 10.4
# Esto son comentarios
#
# Reglas generales
%.o: %.c
      cc -c *.c
# Dependencias
p10_4:    principal.o ley_ohm.o
          cc principal.o ley_ohm.o -lm -o p10_4
#
principal.o ley_ohm.o: stdio.h
```

Conclusión

En esta sección se ha visto como compilar automáticamente aplicaciones en C usando la herramienta `make`. El uso de esta herramienta nos da una utilidad potente y flexible para actualizar las versiones de la aplicación, incluyendo de forma automática las últimas actualizaciones del código fuente. La posibilidad de construir archivos de descripción de la aplicación y de sus dependencias, abre la puerta a desarrollos incrementales de grandes aplicaciones y al mantenimiento automático de la misma.

10.6. PROGRAMAS DE APLICACIÓN ADICIONAL

Presentación

En esta sección se muestran dos ejemplos un poco más complejos que los vistos hasta el momento. El objetivo es hacer que el lector pueda ver una aplicación con varios módulos y que aprenda a leerla y a relacionar los módulos para poder entender el programa. Asimismo, se pretende que el lector entienda más a fondo cómo se usan las funciones con número variable de argumentos mediante la programación de una utilidad similar a `printf`.

Tabulación de un texto

El programa de aplicación que se muestra sustituye cadenas de blancos por tabuladores para dejar un texto bien ajustado y viceversa. Para ello define dos funciones básicas `entabula`, para poner tabuladores, y `desentabula`, para quitar tabuladores. Ambas funciones usan otra denominada `treat` que realiza el procesamiento básico para las dos.

Programa 10.5 `#include "misfun.h"`

```
/* Autor: Vicente Luque
 * entabula.c: Sustituye cadenas de blancos por el numero
 * correspondiente de tabuladores y blancos y viceversa.
 */

int main (int argc, char *argv[])
{
    char      bufin[SIZEBUF];
    char      bufout[SIZEBUF];
    FILE      *filein = NULL;
    int       initial = 1;
    boolean   exiting = FALSE;
    boolean   desentab = FALSE;
    boolean   stdinend = TRUE;

    while (!exiting && initial < argc) {
        if (strcmp (argv[initial], "-n") == 0) {
            desentab = TRUE;
        } else {
            /* It's a file */
            stdinend = FALSE;
            if (strcmp (argv[initial], "-") == 0) {
                filein = stdin;
            } else {
                filein = fopen (argv[initial], "r");
            }
        }
    }
}
```

```

        if (!filein) {
            perror (argv[0]);
        }
    }
    if (filein) {
        treat (desentab ? desentabula : entabula,
            sizeof (bufin), bufin, bufout, filein, stdout);
        if (filein != stdin) {
            fclose (filein);
        }
    }
    }
    initial++;
}
if (stdinend) {
    treat (desentab ? desentabula : entabula, sizeof (bufin),
        bufin, bufout, stdin, stdout);
}
exit (0);
}

```

El archivo de cabecera "misfun.h" incluye los prototipos de las tres funciones e incluye a su vez otro archivo de cabecera con más definiciones ("mistdef.h"). Lo más interesante de "mistdef.h" es que tiene definiciones de macros que hacen procesamiento de forma similar a una función. El archivo "misfun.c" incluye el cuerpo de las tres funciones indicadas anteriormente. El lector puede mirar el código fuente completo en el disco que se adjunta con el libro para comprender la aplicación completa.

A continuación se muestra el archivo makefile de la aplicación, en el cuál se pueden ver claramente las dependencias entre módulos.

Archivo makefile. all: entabula

```

entabula:entabula.o misfun.o
    cc -pipe entabula.o misfun.o -o entabula

entabula.o:entabula.c
    cc -pipe -c entabula.c -o entabula.o

misfun.o:misfun.c
    cc -pipe -c misfun.c -o misfun.o

```

Programación de la utilidad `mi_printf`

Para ilustrar el uso de funciones con un número variable de argumentos, se muestra en el Programa 10.6 como se constuye un servicio equivalente al `printf` de las bibliotecas de C. Para distinguirlo del de C lo llamaremos `mi_printf`.

Esta utilidad de ejemplo no es tan completa como la del C estándar porque no incluye todos los formatos posibles del `printf`. Los contemplados en este ejemplo son:

- `c` para caracteres.
- `s` para cadenas de caracteres.
- `p` para direcciones de memoria.
- `n` para enteros, y dentro de estos:
 - `o` para octal.
 - `x`, `X` para hexadecimal.
 - `d` para decimales.
 - `u` para enteros sin signo.

La función de utilidad principal es la función `mi_printf`, cuyo esbozo es el siguiente:

```
void mi_printf(char *fmt, ...)
{
    /* extraccion de la lista de argumentos con
       va_start(...);
    procesamiento de la lista de argumentos con
       procesa_printf(...);
    impresion de los argumentos con un bucle y putchar
    final de comprobacion de la lista de argumentos con
       va_end();
    */
}
```

El procesamiento real de la entrada se hace en la función `procesa_printf`, donde se comprueban los especificadores de formato existentes en la cadena que se recibe como entrada y se rellena la cadena de salida con el valor del parámetro de entrada adaptado al formato especificado en cada caso. El cuerpo de esta función recorre la cadena de entrada y ejecuta una sentencia `switch` que hace una cosa u otra en función del especificador de formato recibido. Como ejemplo de una entrada del `switch` se muestra el caso de formateo de una cadena de caracteres:

```
case 's':
    s = va_arg(args, char *);
    if (!s)
        s = "<NULL>";
    len = strlen(s);
```

```

    if (!(flags & IZQUIERDA))
        while (len < ancho_campo-)
            *str++ = ' ';
    for (i = 0; i < len; ++i)
        *str++ = *s++;
    while (len < ancho_campo-)
        *str++ = ' ';
    continue;

```

A continuación se muestra el código completo de la aplicación de ejemplo. Se recomienda al lector que estudie cuidadosamente el Programa 10.6 y que pruebe a modificarlo para ver si ha entendido como funciona.

Programa 10.6

```

#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <ctype.h>

static char ppbuf[1024];

/*
   Funciones de utilidad diversas para convertir partes de formatos.
   Se usan desde las funciones principales de la aplicación.
*/
static int mi_atoi(const char **s)
{
    int i=0;

    while (isdigit(**s))
        i = i*10 + *((*s)++) - '0';
    return i;
}

#define PONCERO 1      /* rellena con cero */
#define SIGN 2         /* largo con y sin signo */
#define PLUS 4         /* muestra mas */
#define ESPACIO 8      /* espacio si hay mas */
#define IZQUIERDA 16   /* justificado a la izquierda */
#define ESPECIAL 32    /* 0x */
#define LARGO 64       /* usar 'ABCDEF' en lugar de 'abcdef' */

#define dividir(n,base) ({ \
    int __res; \
    __res = ((unsigned long) n) % (unsigned) base; \
    n = ((unsigned long) n) / (unsigned) base; \
    __res; })

```

```

/*
  Funcion de utilidad para convertir formatos numericos.
*/

static char * numero(char * str, long num, int base, int tamanyo,
  int precision,int type)
{
  char c,sign,tmp[66];
  const char *digitos="0123456789abcdefghijklmnopqrstuvwxyz";
  int i;

  if (type & LARGO)
    digitos = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  if (type & IZQUIERDA)
    type &= ~PONCERO;
  if (base < 2 || base > 36)
    return 0;
  c = (type & PONCERO) ? '0' : ' ';
  sign = 0;
  if (type & SIGNO) {
    if (num < 0) {
      sign = '-';
      num = -num;
      tamanyo--;
    } else if (type & PLUS) {
      sign = '+';
      tamanyo--;
    } else if (type & ESPACIO) {
      sign = ' ';
      tamanyo--;
    }
  }
  if (type & ESPECIAL) {
    if (base == 16)
      tamanyo -= 2;
    else if (base == 8)
      tamanyo--;
  }
  i = 0;
  if (num == 0)
    tmp[i++]='0';
  else while (num != 0)
    tmp[i++] = digitos[dividir(num,base)];
  if (i > precision)
    precision = i;
  tamanyo -= precision;
  if (!(type&(PONCERO+IZQUIERDA)))
    while(tamanyo-->0)
      *str++ = ' ';

```

```

        if (sign)
            *str++ = sign;
        if (type & ESPECIAL)
            if (base==8)
                *str++ = '0';
            else if (base==16) {
                *str++ = '0';
                *str++ = digitos[33];
            }
        if (!(type & IZQUIERDA))
            while (tamanyo-- > 0)
                *str++ = c;
        while (i < precision--)
            *str++ = '0';
        while (i-- > 0)
            *str++ = tmp[i];
        while (tamanyo-- > 0)
            *str++ = ' ';
        return str;
    }

/* Funcion procesa_printf con el cuerpo principal del tratamiento del
   conversión de formato de los argumentos variables.
   Se apoya en la utilidad numero que es la que hace el
   trabajo de procesamiento de los numeros.
*/
int procesa_printf(char *buf, const char *fmt, va_list args)
{
    int len;
    unsigned long num;
    int i, base;
    char * str;
    const char *s;

    int flags;          /* flags para numero() */

    int ancho_campo;    /* ancho del campo de salida */
    int precision;      /* min. # de digitos para enteros;
                           max. numero de caracteres para cadenas */
    int calificador;    /* 'h', 'l', o 'L' para campos enteros */

    for (str=buf ; *fmt ; ++fmt) {
        if (*fmt != '%') {
            *str++ = *fmt;
            continue;
        }

        /* flags del proceso*/
        flags = 0;

```



```

. repeat:
    ++fmt;          /* salta el primer '%' */
    switch (*fmt) {
        case '-': flags |= IZQUIERDA; goto repeat;
        case '+': flags |= PLUS; goto repeat;
        case ' ': flags |= ESPACIO; goto repeat;
        case '#': flags |= ESPECIAL; goto repeat;
        case '0': flags |= PONCERO; goto repeat;
    }

    /* obtener anchura del campo */
    ancho_campo = -1;
    if (isdigit(*fmt))
        ancho_campo = mi_atoi(&fmt);
    else if (*fmt == '*') {
        ++fmt;
        /* es el proximo argumento */
        ancho_campo = va_arg(args, int);
        if (ancho_campo < 0) {
            ancho_campo = -ancho_campo;
            flags ^= IZQUIERDA;
        }
    }

    /* obtener la precision */
    precision = -1;
    if (*fmt == '.') {
        ++fmt;
        if (isdigit(*fmt))
            precision = mi_atoi(&fmt);
        else if (*fmt == '*') {
            ++fmt;
            /* es el siguiente argumento */
            precision = va_arg(args, int);
        }
        if (precision < 0)
            precision = 0;
    }

    /* obtener el calificador de conversion */
    calificador = -1;
    if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
        calificador = *fmt;
        ++fmt;
    }

    /* base numerica por defecto */
    base = 10;

```

```

switch (*fmt) {
case 'c':
    if (!(flags & IZQUIERDA))
        while (--ancho_campo > 0)
            *str++ = ' ';
    *str++ = (unsigned char) va_arg(args, int);
    while (--ancho_campo > 0)
        *str++ = ' ';
    continue;

case 's':
    s = va_arg(args, char *);
    if (!s)
        s = "<NULL>";
    len = strlen(s);

    if (!(flags & IZQUIERDA))
        while (len < ancho_campo--)
            *str++ = ' ';
    for (i = 0; i < len; ++i)
        *str++ = *s++;
    while (len < ancho_campo--)
        *str++ = ' ';
    continue;

case 'p':
    if (ancho_campo == -1) {
        ancho_campo = 2*sizeof(void *);
        flags |= PONCERO;
    }
    str = numero(str,
        (unsigned long) va_arg(args, void *), 16,
        ancho_campo, precision, flags);
    continue;

case 'n':
    if (calificador == 'l') {
        long * ip = va_arg(args, long *);
        *ip = (str - buf);
    } else {
        int * ip = va_arg(args, int *);
        *ip = (str - buf);
    }
    continue;

/* formato de numero entero - activar el flag y terminar */
case 'o':
    base = 8;
    break;

```

```

    case 'X':
        flags |= LARGO;
    case 'x':
        base = 16;
        break;

    case 'd':
    case 'i':
        flags |= SIGNO;
    case 'u':
        break;

    default:
        if (*fmt != '%')
            *str++ = '%';
        if (*fmt)
            *str++ = *fmt;
        else
            --fmt;
        continue;
    }
    if (calificador == 'l')
        num = va_arg(args, unsigned long);
    else if (calificador == 'h')
        if (flags & SIGNO)
            num = va_arg(args, short);
        else
            num = va_arg(args, unsigned short);
    else if (flags & SIGNO)
        num = va_arg(args, int);
    else
        num = va_arg(args, unsigned int);
    str = numero(str, num, base, ancho_campo, precision, flags);
}
*str = '\0';
return str-buf;
}
/*
Funcion con el cuerpo principal de la rutina printf.
Se apoya en la utilidad procesa_printf que es la que hace el
trabajo real.
*/
void mi_printf(char *fmt, ...)
{
    va_list args;
    char ch, *bptr;
    int i;

    /* extraccion de la lista de argumentos */

```

```

        va_start(args, fmt);
    /* procesamiento de la lista de argumentos */
    i = procesa_printf(ppbuf, fmt, args);
    bptr = ppbuf;
    /* impresion de argumentos */
    while((ch = *(bptr++)) != 0) {
        if(ch == '\n')
            putchar('\r');
        putchar(ch);
    }
    va_end(args);
    return;
}
/* Programa principal para demostrar como funciona la
   utilidad printf programada.
*/
void main(int argc, char ** argv)
{
    mi_printf("Esta es una prueba de argumentos variables con %s \n",
              argv[0]);
    mi_printf("Se imprime un entero %4d, un hexadecimal %x y un octal %o\n",
              12, 0xbd34, 16 );
}

```

Conclusión

En esta sección se ha visto una aplicación modular con características novedosas. Lo más interesante es la relación de módulos vista en el makefile y el uso de macros que tienen varias sentencias asociadas y cerradas entre llaves. se recomienda al lector que estudie cuidadosamente estos conceptos. Además se ha visto más a fondo cómo programar y usar funciones con número variable de argumentos. Ambas aplicaciones son similares a las que el lector podría encontrar en un sistema real programado en C.

Problemas del final del capítulo

Diseño de programas

Para el desarrollo de los siguientes programas, use un diseño descendente y una estructura en bloques sin variables globales. La función `main()` se encargará básicamente de llamar al resto de las funciones. Cuando una función necesite devolver más de una variable, se usarán punteros. No olvide incluir toda la documentación en la versión final del programa. En los casos en que la aplicación sea grande, o sea lógico hacerlo así, no dude en aplicar criterios de descomposición modular para dividir la aplicación en distintos archivos fuente.

1. Modifique el Programa 10.1 para romperlo en tres módulos. En el primero debe quedar únicamente el módulo principal de la aplicación. En el segundo las funciones de cálculo de la ley de Ohm y en el tercero las funciones de aplicación matemática multiplicar y dividir ¿Afecta esto al uso de punteros a funciones?
2. Construya una aplicación que manipule una máquina expendedora de dulces. Dicha aplicación tiene cuatro entradas de petición (caramelos, chicles, gominolas y pipas), cada una de las cuales dispara un

mecanismo de selección y aceptación de dinero. Suponiendo que la máquina no para nunca, use un vector de punteros a función para construir el programa principal de dicha máquina.

3. Construya un programa que genere una secuencia de números enteros pseudoaleatorios. Dicha secuencia comienza con un número que se recibe como argumento y su valor se transmite mediante variables de valor permanente a través de sucesivas ejecuciones del generador de números.

4. Modifique el Programa 10.6 para introducir el especificador de formato `%f` en la función `mi_printf`. Dicho formato debe admitir calificadores de longitud tales como `%4.5f`.
5. La Figura 10.7 muestra una interfaz hardware entre una impresora con conector DB25 y un circuito eléctrico que maneja relés. Observe en dicha figura que hay dos puertos de salida y uno de entrada. Desarrolle un programa C que le permita al usuario activar cualquiera de los dos puertos de salida.

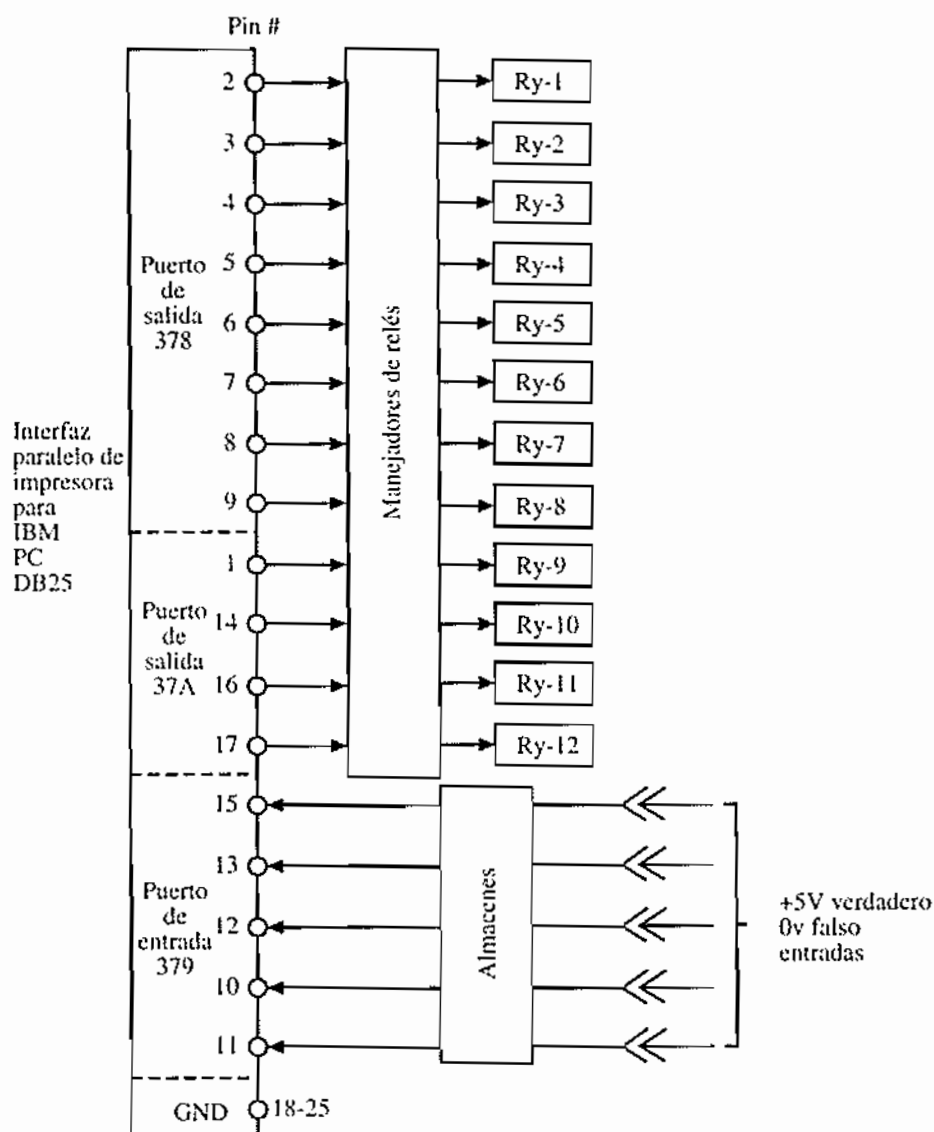


Figura 10.7. Interfaz hardware de una impresora.



Apéndices

Apéndice A: Manual de consulta de C

Palabras reservadas de C:

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Sentencias de C

En C una sentencia está formada por palabras reservadas, expresiones y otras sentencias. Se usan para controlar el flujo de ejecución de un programa. Incluyen sentencias para ejecutar ciclos, transferencias de control (toma de decisiones) y selección de otras sentencias. Lo que sigue es un resumen de las sentencias de C, listadas en orden alfabético.

Asignación =

Asigna el valor de la expresión de la derecha a la variable de la izquierda.

Ejemplo:

```
resistencia = 12;
```

Composición

Una sentencia dentro del cuerpo de otra sentencia. El ejemplo del break que se muestra a continuación ilustra las sentencias compuestas.

break

Termina la ejecución del bucle do, for, switch o while más interior.

Ejemplo:

```
while (contador > 0)
{
    if (contador > 5) break;
    /* El bucle termina cuando contador > 5. */
}
```

continue

Usado en las instrucciones do, for y while. Transfiere el control a la siguiente iteración de estas instrucciones.

Ejemplo:

```
while (contador > 0)
{
    if (contador == 5) continue;
    /* El bucle se salta cuando contador igual a 5. */
}
```

do instruccion while (expresion)

El cuerpo de la instrucción do se ejecuta una o más veces hasta que la expresión del while es FALSO (cero), pasándose entonces el control a la siguiente instrucción.

Ejemplo:

```
do
{
    valor = valor + 1;
}
while (valor < 5)
```

Bucle for

```
for (expresion1; expresion2; expresion3)
    <instrucciones>
```

El proceso es como sigue:

expresion1 se evalúa primero (sólo una vez).

expresion2 se evalúa y si es VERDADERO se ejecutan las instrucciones.

expresion3 se evalúa en cada pasada del bucle.

Ejemplo:

```
for (contador = 1; contador < 5; contador++)
{
    printf ("El contador es %d. \n", contador);
}
/* el bucle continúa mientras contador < 5 */
```

Instrucción if

```
if (expresion)
    instruccion1
else
    instruccion2
```

Si *expresion* es VERDADERO (no cero), entonces se evalúa *instruccion1*; en caso contrario se evalúa *instruccion2*.

Ejemplo:

```
if (valor != 5)
    printf ("Valor distinto de 5.");
else
    printf ("Valor igual a 5.");
/* Ejecuta el primer printf() si el valor no es igual a 5.
   En caso contrario ejecuta el segundo printf() */
```

Instrucción return

Termina la ejecución de la función en la que es activada. El flujo del programa vuelve a la función que la llamó.

Ejemplo:

```
return (valor);
/* Devuelve el valor de valor a la función que llamó */
```

Instrucción switch

Produce la transferencia de control a una instrucción dentro del cuerpo del switch.

Ejemplo:

```
switch (seleccion)
{
    case 1: {
        resistencia = 10;
        printf ("La resistencia es de 10 Ohmios.");
    }
    break;
    case 2: printf ("No hay valor asignado.");
    break;
    default: printf ("Selección inválida.");
}
/* El valor de seleccion determina una de los
   tres casos */
```

Bucle while

```
while (expresion)
    <instruccion>
```

El cuerpo de <instruccion> se ejecuta cero o más veces hasta que *expresion* sea FALSO (cero).

Ejemplo:

```
while (valor < 5)
{
    valor ++;
    printf ("El valor es %d. \n", valor);
}
/* El bucle continúa mientras valor < 5. */
```

Apéndice B: Funciones matemáticas estándar de ANSI C

Este apéndice muestra las bibliotecas de rutinas matemáticas de ANSI C. Están listadas alfabéticamente en la Tabla B.1.

Tabla B.1. Funciones matemáticas estándares en ANSI C

Función	Archivos a incluir y descripción	Ejemplo
<code>abs ()</code>	Devuelve el valor absoluto de un argumento entero <i>n</i> . #include <stdlib.h> int abs (int n);	<code>v = abs (-3);</code> devuelve <code>v = 3</code>
<code>acos ()</code>	Calcula el arco coseno del argumento cuyo valor está entre 1 y -1. #include <math.h> double acos (double x);	<code>angulo = acos (0.5);</code> devuelve <code>angulo = pi / 3</code>
<code>asin ()</code>	Calcula el arco seno del argumento cuyo valor está entre 1 y -1. #include <math.h> double asin (double x);	<code>angulo = asin (0.707);</code> devuelve <code>angulo = pi / 4</code>
<code>atan ()</code>	Calcula el arco tangente del argumento cuyo valor está entre -pi/2 y pi/2. #include <math.h> double atan (double x);	<code>angulo = atan (1.0);</code> devuelve <code>angulo = pi / 4</code>
<code>atan2 ()</code>	Calcula el arco tangente de la razón entre dos argumentos. Esta función puede usar el signo del resultado para determinar a qué cuadrante del sistema de coordenadas cartesianas pertenece el ángulo. #include <math.h> double atan2 (double x, double y);	<code>angulo = atan2 (1, 2);</code> devuelve el arco tangente cuyo ángulo es en radianes.
<code>ceil ()</code>	Calcula el techo de un argumento real. Es decir, el menor entero que es igual o está más próximo por encima al valor del argumento. Es útil cuando se redondea valores reales al siguiente valor entero. #include <math.h> double ceil (double x);	<code>techo = ceil (3.2);</code> devuelve <code>techo = 4.0</code>
<code>cos ()</code>	Calcula el coseno del argumento, cuyo valor está en radianes. #include <math.h> double cos (double x);	<code>ang_cos = cos (0);</code> devuelve <code>ang_cos = 1</code>
<code>cosh ()</code>	Calcula el coseno hiperbólico del argumento. Si el valor es mayor de lo que el sistema puede manejar, ocurrirá un error de rango. #include <math.h> double cosh (double x);	<code>valor = cosh (x);</code> devuelve el valor del coseno hiperbólico de <i>x</i>
<code>div ()</code>	Devuelve el valor de la división entera en forma de una estructura que tiene el cociente y el resto resultantes de la división. Está definida en el fichero de cabecera como:	<code>valor = div (14, 3);</code> devuelve <code>valor.quot = 4</code> y <code>valor.rem = 2</code>

Tabla B.1. (Continuación)

Función	Archivos a incluir y descripción	Ejemplo
	<pre>typedef struct { int quot; /* Cociente */ int rem; /* Resto */ } div_t; #include <stdlib.h> div_t div (int number, int denom);</pre>	
exp ()	Devuelve el exponencial e^x (base e , donde $e = 2,7182828$) del argumento. <pre>#include <math.h> double exp (double x);</pre>	<pre>valor = exp (1); devuelve valor = 2.7182818 ...</pre>
fabs ()	Devuelve un tipo <code>double</code> que es el valor absoluto del argumento. <pre>#include <math.h> double fabs (double x);</pre>	<pre>valor = fabs (-3.94); devuelve valor = 3.94</pre>
floor ()	Devuelve el mayor entero que es menor o igual el argumento. <pre>#include <math.h> double floor (double x);</pre>	<pre>valor = floor (8.25); devuelve valor = 8.0</pre>
fmod ()	Devuelve el valor del resto de la división real de los argumentos, asegurándose de que el valor devuelto es el mayor entero posible. <pre>#include <math.h> double fmod (double x, double y);</pre>	<pre>valor = fmod (x, y); devuelve n = floor(x/y) y valor = n - n*y;</pre>
frexp ()	Devuelve la mantisa m y el exponente entero n del argumento real. <pre>#include <math.h> double frexp (double x, int *exp_ptr);</pre>	<pre>man = frexp (x, &exp); devuelve man = mantisa y exp = exponente</pre>
labs ()	Devuelve el valor absoluto de un argumento de tipo entero largo. <pre>#include <stdlib.h> long labs (long n);</pre>	<pre>valor = labs (-63490L); devuelve valor = 63490</pre>
ldexp ()	Devuelve el valor real de un real de tipo doble con un exponente en base 2. <pre>#include <math.h> double ldexp (double x, int exp);</pre>	<pre>valor = ldexp (x, exp); devuelve valor = x * 2^{exp}</pre>
ldiv ()	Devuelve el valor de la división de enteros largos en forma de una estructura que tiene el cociente y el resto resultantes de la división. Está definida en el fichero de cabecera como: <pre>typedef struct { long quot; /* Cociente */ long rem; /* Resto */ } ldiv_t; #include <stdlib.h> ldiv_t ldiv (long number, long denom);</pre>	<pre>valor = ldiv (63463L, 63460L); devuelve valor.quot = 1 y valor.rem = 3</pre>

Tabla B.1. (Continuación)

Función	Archivos a incluir y descripción	Ejemplo
<code>log ()</code>	Devuelve el logaritmo natural del argumento real. #include <math.h> double log (double x);	valor = log (3); devuelve valor = 1.098612
<code>log10 ()</code>	Devuelve el logaritmo en base 10 del argumento real. #include <math.h> double log10 (double x);	valor = log10 (3); devuelve valor = 0.47712
<code>modf ()</code>	Devuelve la parte decimal y entera del argumento real. #include <math.h> double modf (double x, double *intptr);	valor = modf (x, &int_pt); devuelve un valor real en la dirección dada por int_pt
<code>pow ()</code>	Devuelve el valor x^y para los argumentos x e y #include <math.h> double pow (double x, double y);	valor = pow (2, 3); devuelve valor = 8
<code>rand ()</code>	Devuelve un número entero pseudoaleatorio entre 0 y RAND_MAX, definido en el fichero de cabecera. #include <stdlib.h> int rand (void);	valor = rand (); devuelve un valor aleatorio
<code>sin ()</code>	Calcula el seno del argumento, cuyo valor está en radianes. #include <math.h> double sin (double x);	valor = sin (x); devuelve el seno de x
<code>sinh ()</code>	Calcula el seno hiperbólico del argumento real. #include <math.h> double sinh (double x);	valor = sinh (x); devuelve el seno hiperbólico de x
<code>sqrt ()</code>	Devuelve la raíz cuadrada de un argumento entero positivo. #include <math.h> double sqrt (double x);	valor = sqrt (9.0); devuelve valor = 3
<code>srand ()</code>	Pone la semilla para la función generadora de números aleatorios. La semilla depende del valor del argumento entero sin signo. #include <stdlib.h> int srand (unsigned n);	srand (n);
<code>tan ()</code>	Calcula el seno del argumento, cuyo valor está en radianes. #include <math.h> double tan (double x);	valor = tan (x); devuelve la tangente de x
<code>tanh ()</code>	Calcula la tangente hiperbólica del argumento real. #include <math.h> double tanh (double x);	valor = tanh (x); devuelve la tangente hiperbólica de x

Apéndice C: Juego de caracteres ASCII

Tabla C.1. Juego de caracteres de ASCII

Carácter*	Código	Carácter	Código	Carácter	Código	Carácter	Código
NUL	0	blanc	32	@	64	'	96
SOH	1	!	33	A	65	a	97
STX	2	"	34	B	66	b	98
ETX	3	#	35	C	67	c	99
EOT	4	\$	36	D	68	d	100
ENQ	5	%	37	E	69	e	101
ACK	6	&	38	F	70	f	102
BEL	7	'	39	G	71	g	103
BS	8	(	40	H	72	h	104
HT	9	)	41	I	73	i	105
LF	10	*	42	J	74	j	106
VT	11	+	43	K	75	k	107
FF	12	,	44	L	76	l	108
CR	13	-	45	M	77	m	109
SO	14	.	46	N	78	n	110
SI	15	/	47	O	79	o	111
DLE	16	0	48	P	80	p	112
DC1	17	1	49	Q	81	q	113
DC2	18	2	50	R	82	r	114
DC3	19	3	51	S	83	s	115
DC4	20	4	52	T	84	t	116
NAK	21	5	53	U	85	u	117
SYN	22	6	54	V	86	v	118
ETB	23	7	55	W	87	w	119
CAN	24	8	56	X	88	x	120
EM	25	9	57	Y	89	y	121
SUB	26	:	58	Z	90	z	122
ESC	27	;	59	[	91	{	123
FS	28	<	60	\	92		124
GS	29	=	61	]	93	}	125
RS	30	>	62	↑	94	-	126
US	31	?	63	_	95	DEL	127

* Estos 32 caracteres (números de código 0 a 31) se conocen como **caracteres de control**.

Soluciones

Respuestas de las revisiones de las secciones — Capítulo 1

Sección 1.1

1. El propósito de un editor es permitir la introducción de código fuente. Se necesita para conseguir este propósito.
2. La razón para usar un compilador es que el computador tenga acceso a programas que pueda entender.
3. Un compilador convierte el código fuente en un código que entiende el computador.

Sección 1.2

1. Tres ventajas del lenguaje de programación C son: transportabilidad, control del computador y flexibilidad. Otras ventajas pueden verse en la Tabla 1.1.
2. La transportabilidad significa que un programa escrito para un computador ejecutará en otro con muy pocos o ningún cambio.
3. Todos los programas C deben empezar con la palabra reservada `main`.
4. Un ejemplo de un comentario en C es: `/* Esto es un comentario */`
5. Las llaves indican el comienzo y el fin de las instrucciones de un programa.

Sección 1.3

1. La estructura del programa está relacionada con la apariencia del código fuente. Una buena estructura de programa hará el código fuente fácil de leer, modificar y depurar.
2. No es necesario estructurar un programa C para que compile sin errores. Una buena estructura hace las cosas más fáciles al programador, no al compilador.
3. La estructura de bloques hace que el programa sea más fácil de leer y comprender. Un ejemplo es la estructura usada para una carta de negocios.
4. La razón de ser de la programación estructurada es hacer los programas más fáciles de leer, modificar y depurar.
5. El bloque del programador se usa para explicar todos los puntos importantes del programa. Contiene:
 - a) Nombre del programa.
 - b) Programador.
 - c) Descripción del programa.

- d) Explicación de las variables.
- e) Explicación de las constantes.

Sección 1.4

1. Se usa un conjunto de caracteres para escribir los programas en C. Estos caracteres son las letras mayúsculas y minúsculas del alfabeto inglés, los diez dígitos del sistema numérico arábigo y el subrayado (_). Los espacios en blanco se usan para separar los elementos de un programa C.
2. En C, un componente léxico es el elemento más básico reconocido por el compilador.
3. Los tipos principales de datos usados en C son números, caracteres y cadenas de caracteres.
4. En C, una palabra reservada es un componente léxico predefinido que tiene un significado especial para el compilador.
5. El tipo de datos que maneja el número más largo es `double`.

Sección 1.5

1. El propósito de la función `printf()` es escribir información por la salida estándar.
2. Los caracteres tienen comillas simples, las cadenas de caracteres comillas dobles.
3. El especificador de formato en una función `printf()` que indica cómo convertir, imprimir y formatear sus argumentos.
4. Un argumento es el valor real entre los paréntesis de una función.
5. Deben existir tantos argumentos como especificadores de formato. Los argumentos extra se ignoran.

Sección 1.6

1. Una función C es una colección independiente de declaraciones y sentencias.
2. Un identificador es el nombre que se da a las partes claves de un programa.
3. Todos los identificadores deben comenzar con una letra del alfabeto (mayúscula o minúscula) o el subrayado (_). El resto del identificador puede usar cualquier arreglo de letras (mayúsculas o minúsculas), dígitos (0 a 9) y subrayados. Eso es todo, no se permiten más caracteres (los espacios por ejemplo). No se debe usar palabras reservadas como identificadores.
4. Los primeros 32 caracteres de un identificador son reconocidos por C.

Sección 1.7

1. Se puede pensar en una variable como en una posición específica de memoria reservada para un tipo de datos específico y con un nombre especial para referenciarla fácilmente.
2. En C, hay que declarar las variables antes de usarlas. Para declarar una variable, debe declarar el tipo y el identificador de la variable.
3. Tres especificadores de tipo fundamentales en C son: `char`, `int` y `float`.
4. Iniciar una variable significa combinar su declaración con un operador de asignación.
5. Para evitar que aparezca en el programa una variable nueva como resultado de un error de teclado.

Sección 1.8

1. Los operadores aritméticos más frecuentemente usados en C son: `+` => (suma), `-` => (resta), `*` => (multiplicación), `/` => (resto) y `%` => resto.
2. En la división entera, C truncará el resto. Esto significa que una división como `3/5` dará 0.
3. En C, `3 - 2 = resultado;` no está permitido. No se puede hacer una asignación sobre una expresión.
4. La prioridad de una operación indica el orden en que se realizan las operaciones aritméticas.
5. Un ejemplo de asignación compuesta es: `valor -= 5;`

Sección 1.9

1. La secuencia de escape en la función `printf()` produce una salida de la interpretación normal de una cadena de caracteres.
2. Cuando se usa una función `printf()`, el carácter *backslash* se denomina carácter de escape.

3. Tres secuencias de escape de la función `printf()` son: `\n` => (nueva línea), `\t` => (tabulador) y `\b` => (espacio atrás).
4. Un especificador de campo, según se usan en la función `printf()`, determina el número mínimo de blancos que se dejan a la izquierda del punto decimal y el máximo número de espacios a la derecha del número decimal.

Sección 1.10

1. La función `scanf()` permite al programa obtener información desde el teclado.
2. La función `scanf()` sabe sobre que identificador de variable almacenar la entrada porque se le indica como argumento precedido por un carácter & sin espacio intermedio (como `&variable`).
3. En la mayoría de sistemas, la función `scanf()` causa un retorno de carro a una nueva línea de forma automática.
4. Los valores de tipo real pueden especificarse como números completos o usando la notación E.
5. Para tener salida de valores por la pantalla en notación E, sólo debe cambiarse el especificador de formato en la función `printf()` de `%f` a `%e`.

Sección 1.11

1. Hay tres mensajes de error en C: fatales, de compilación y de aviso.
2. Un error fatal termina el proceso de compilación.
3. La sensibilidad al tipo de letra significa que un compilador de C distingue entre letras mayúsculas y minúsculas. Esto significa que cuando una variable se declara en mayúsculas, debe usarse de forma consistente en el resto del programa.
4. Es una buena práctica buscar y examinar los puntos y coma cuando no se entiende la causa de un mensaje de error, porque un punto y coma perdido distorsiona a veces el significado del programa de tal forma que el compilador no puede concluir el tipo de error del mismo. Por ello, se origina un mensaje de error que suele depender de lo que hay a continuación del punto y coma perdido.
5. Un comentario anidado es un comentario usado dentro de otro. No es legal y no permite que el programa se ejecute. Algunos compiladores permiten seleccionar esta opción, pero no se recomienda.

Sección 1.12

1. El primer paso en el desarrollo de un programa es definir por escrito sus requisitos.
2. Los elementos que deberían incluirse en la definición del problema son el propósito del programa, sus requisitos de entrada (fuente) y sus requisitos de salida (destino).
3. El primer paso en la codificación real del programa es esbozarlo usando solamente comentarios.
4. El proceso usado para desarrollar el programa final consistirá en la codificación de cada sección del programa por separado, su compilación y su ejecución. Cualquier fallo del programa se elimina del mismo cuando se crea la sección donde está el fallo.

Soluciones de la autoevaluación — Capítulo 1

1. Este programa puede no compilar y ejecutar en su programa porque puede hacer falta incluir una directiva `#include <stdio.h>`. Si este fuera el caso, asegúrese de usar un formato estándar y de ponerlo antes de cualquier otro código fuente, comenzando en la parte superior izquierda de la pantalla.
2. El programa resuelve la corriente de un circuito. El usuario del programa debe introducir los valores del voltaje y la resistencia del circuito. El programa imprimirá el valor de la corriente resultante. Esta explicación se ha extraído leyendo los comentarios del principio del programa.
3. Hay tres variables en el programa, todas de tipo `float`. Esto se sabe leyendo el `/* Bloque de declaración */`.
4. La función `puts()` se usa para explicar el programa al usuario porque proporciona un salto de línea automático. No se necesitan más mandatos de formato que los necesarios para mostrar una cadena de caracteres por la salida.

5. El usuario del programa puede introducir los valores del voltaje y la resistencia como números complejos, con fracción decimal, o con notación E. Esto se ve estudiando la función `scanf()`, donde se usa el formato `%f`.
6. La salida se muestra en notación E. Esto se ve estudiando la función `printf()`. Se usa el especificador `%e` (notación E) para presentar los valores de las variables del programa.

Soluciones de los problemas impares del final del capítulo — Capítulo 1

Sección 1.1

1. El programa que se usa para escribir el código fuente es un editor.

Sección 1.2

3. Todos los programas C deben empezar con `main`.
5. Las llaves de apertura y cerrado (`{` y `}`) indican el principio y el final de las instrucciones de un programa.

Sección 1.3

7. El propósito del bloque del programador es presentar toda la información importante del programa.
9. No, no es necesario tener el bloque del programador para que un programa compile.

Sección 1.4

11. El subrayado (`_`).

Sección 1.5

13. Los caracteres se representan con comillas simples y las cadenas de caracteres con comillas dobles.
15. El nombre que se da a los valores reales entre los paréntesis de una función es argumentos.

Sección 1.6

17. El nombre que se da a una colección independiente de declaraciones y sentencias es función.
19. Los primeros 32 caracteres de un identificador son reconocidos en C.

Sección 1.7

21. Los tres tipos fundamentales de especificadores de tipo en C son `char`, `int` y `float`.
23. El requisito de que todas las variables deban ser declaradas antes de usarse previene que aparezcan nuevas variables debidas a errores de tecleo.

Sección 1.8

25. La característica única de la división entera en C es que se trunca el resultado.
27. La sentencia de C `resultado *= 5;` significa `resultado = resultado * 5;`

Sección 1.9

29. La secuencia de escape en una función `printf()` produce una interpretación muy diferente de las cadenas de caracteres.
31. El especificador de ancho de campo determina el número de dígitos a la derecha del punto decimal.

Sección 1.10

33. Como se ha descrito en este capítulo, el propósito de la función `scanf()` es obtener la entrada del usuario del programa.

35. El usuario del programa puede introducir valores como números completos (sin parte decimal), en números con fracción decimal y en notación E.

Sección 1.11

37. Un mensaje de error fatal termina inmediatamente la compilación.
39. No, los mensajes de compilación no siempre identifican el control del programa. Depende del tipo de problema. Un punto y coma olvidado o mal situado puede confundir totalmente al compilador.

Sección 1.12

41. El primer paso en la codificación de un programa es escribir los comentarios que dividen el programa en sus secciones principales.

Respuestas de las revisiones de las secciones — Capítulo 2

Sección 2.1

1. El compilador de C no requiere que un programa en C esté estructurado. Un programa se estructura para facilitar la labor del programador y de aquellos que tienen que leerlo, depurarlo y modificarlo.
2. Una estructura de bloques significa que el programa se construirá de forma que tenga varios grupos de instrucciones en vez de una lista continua de instrucciones.
3. Cada bloque de programa debería comenzar con un comentario que explique lo que el bloque hará.
4. Los bloques de programas pueden separarse por medio de espacios o comentarios que formen líneas (`/* — */`) a través del código del programa.
5. El cuerpo de un bloque de programa se resalta sangrando las líneas del programa desde la izquierda.
6. Los tres tipos de bloques son: secuencial, de selección o bifurcación y de bucle o repetitivo.

Sección 2.2

1. Una función en C es una parte específica de un programa que se diseña para devolver un valor.
2. Un tipo `void` no devuelve ningún valor.
3. Un prototipo de función declara el tipo de la función, el nombre y los parámetros de la misma al comienzo de un programa C.
4. El propósito de utilizar un prototipo de función es permitir al compilador conocer las funciones que se utilizarán en un programa. Esto asegura que se asigne el tipo y la cantidad de memoria adecuada a cada función antes de ser utilizada.
5. Llamar a una función significa invocar su acción. Esto se hace utilizando el identificador de la función en la función que realiza la llamada (como se hizo en la función `main()` en esta sección).
6. La función `exit()` se utiliza como la última función a la que se llama desde `main()` para asegurar que el computador libera todos los recursos necesarios. Esto le permite utilizar otro programa después de que su programa C haya terminado. El valor utilizado en `exit()` es 0. Por convenio, esto significa una terminación del programa con éxito.

Sección 2.3

1. El parámetro de una función identifica el tipo de variable que será pasado a la función.
2. Un parámetro formal es el identificador que se utiliza para identificar el tipo de argumento. Un parámetro real es el identificador que contiene el valor que se pasa a la función. Estos deben ser del mismo tipo de datos, aunque pueden ser diferentes, es decir, pueden utilizar diferentes identificadores.
3. Pasar valores entre funciones significa que un valor obtenido de una función influya en la operación o valor de la función llamada.
4. La única diferencia que hay entre el código de un prototipo de función y la cabecera de declaración de la función es que el prototipo debe terminar con un punto y coma mientras que la declaración no.

- Un método de pasar un valor de vuelta a la función que realizó la llamada es utilizar la sentencia de `C return()`, cuyo argumento es el valor a ser devuelto.

Sección 2.4

- Sí, una función puede pasar más de un valor a la función llamada. El número de valores a pasar deben ser todos los identificados por los parámetros formales.
- Sí, una función puede llamar a más de una función. El requisito es que la función haya sido definida.
- No, no hay ninguna diferencia en el orden en el que se definen las funciones en el cuerpo de un programa.
- El significado de una función que llama a otra función es que cualquier función puede llamar a otra función sin importar cuantas funciones hayan sido activadas. Esto significa que una función puede llamarse también a sí misma.
- Recursividad significa que una función se llama a sí misma.

Sección 2.5

- Una directiva de preprocesador representa una sentencia especial al preprocesador que provoca una acción antes de que el programa sea compilado y ejecutado.
- La directiva `#include` indica al preprocesador que sustituya un conjunto de componentes léxicos por otro conjunto de componentes.
- Una macro es una instrucción del preprocesador.
- Las constantes se definen normalmente en C utilizando órdenes del preprocesador. Por convenio, las constantes suelen definirse en mayúsculas.
- Sí, se puede utilizar un parámetro en un `#define`. Por ejemplo `#define cubo(x) x*x*x`.

Sección 2.6

- La ventaja de crear sus propios archivos de cabecera es que se puede construir una biblioteca de datos específicos para su área de tecnología. Esto le evitará la necesidad de repetir el mismo código una y otra vez.
- Como se presentó en esta sección, la información contenida en sus archivos de cabecera consta de una serie de directivas `#define`.
- La extensión que se da a un archivo de cabecera en C es `.h`.
- Los archivos de cabecera son invocados en un programa C utilizando la sentencia `#include "archivo.h"`, siendo `archivo.h` el nombre correcto de su archivo de cabecera.

Sección 2.7

- El principal objetivo en el desarrollo de un programa de aplicación para esta sección fue incrementar la legibilidad y comprensión de lo que hace el programa al mismo tiempo que se reserva las características fundamentales del lenguaje C.
- La siguiente información debería incluirse en el bloque del programador:
 - Información del programa
 - Nombre del programa
 - Nombre del programador
 - Nombre de la persona que hizo la última modificación
 - Explicación del programa
 - Qué hace el programa
 - Los datos que se requieren como entrada
 - El proceso que se llevará a cabo
 - Cuáles son los resultados
 - Las unidades para todas las variables
 - Descripciones de las funciones
 - Descripciones de los prototipos de las funciones
 - Propósito de cada prototipo

3. El uso de la directiva `#define` puede suponer mucho ahorro de código en un programa. Más importante aún, estas sentencias permiten el desarrollo eventual de archivos de cabecera, por lo que estas sentencias pueden ser utilizadas en futuros programas.
4. La última cosa que normalmente se pregunta al usuario de un programa tecnológico típico es si el programa debe repetirse.

Soluciones de la autoevaluación — Capítulo 2

1. Se definen cuatro funciones en el programa: `main()`, `explicar_programa()`, `obtener_valores()` y `calcular_y_mostrar()`.
2. El número total de funciones utilizadas en el programa es siete. Las tres funciones adicionales utilizadas son `exit()`, `printf()` y `scanf()`.
3. Los identificadores utilizados como parámetros formales en el programa son `f`, `l`, `r` y `v`.
4. Los identificadores utilizados como parámetros reales en el programa son `resistencia`, `inductancia`, `frecuencia` y `voltaje`.
5. La función `obtener_valores` pasa valores a la función `calcular_y_mostrar`.
6. Los valores son pasados de una función a otra por medio de identificadores que actúan como argumentos reales del mismo tipo y número. Un ejemplo en el programa es:

```
calcular_y_mostrar(frecuencia, inductancia, resistencia, voltaje);
```
7. En el programa hay ocho identificadores de variable: `f`, `l`, `r`, `v`, `resistencia`, `inductancia`, `frecuencia` y `voltaje`.
8. El mínimo cambio que se requiere para imprimir el valor de la reactancia inductiva debería hacerse en la función `calcular_y_mostrar`, utilizando el valor devuelto por `X_L(f, l)` en una sentencia `printf()`.
9. Sí, el programa acepta notación exponencial (E) puesto que todas las variables de entrada son de tipo `float`.

Soluciones de los problemas impares del final del capítulo — Capítulo 2

Sección 2.1

1. La estructura de bloques es un método para romper un programa en grupos distintos de código de tal manera que el programa sea más fácil de leer y entender.
3. Un bloque repetitivo o de bucle tiene la capacidad de volver hacia atrás y repetir parte del programa.

Sección 2.2

5. La parte de un programa C que informa al compilador sobre las funciones que serán definidas en el programa se denomina *prototipo de función*.
7. Se llama a una función utilizando el identificador de la función dentro de otra función junto con argumentos reales.

Sección 2.3

9. El tipo que se asigna a una función cuando ésta no devuelve ningún valor es `void`.
11. La sentencia `return()` en C se utiliza para devolver un valor de una función a la función que la llamó.

Sección 2.4

13. Sí, una función puede llamar a más de una función.
15. Sí, una función llamada puede llamar a otra función.

Sección 2.5

17. La directiva de preprocesador que se ha presentado en este capítulo ha sido `#define`.
19. Un ejemplo de uso de parámetros con `#define` es:

```
#define cuadrado(x)    x*x
```

Sección 2.6

21. La forma de invocar a su propio archivo de cabecera es por medio de

```
#include "miarchivo.h"
```

(asumiendo que `miarchivo.h` es el nombre de un archivo de cabecera).

Respuestas de las revisiones de las secciones — Capítulo 3

Sección 3.1

1. Los operadores de comparación son símbolos que indican las relaciones entre dos cantidades.
2. Los operadores de comparación de C son:
 - > Mayor que
 - >= Mayor o igual que
 - < Menor que
 - <= Menor o igual que
 - == Igual a
 - != Distinto de
3. Las dos condiciones que puede tomar un operador de comparación son VERDADERO y FALSO.
4. La afirmación de que un operador de comparación devuelve un valor significa que se evaluará con un valor 1 si la relación es VERDADERA y con un valor 0 si es falsa.
5. La diferencia entre las operaciones representadas por los símbolos `=` y `==` se explica a continuación. El símbolo igual simple es un operador de asignación que asigna el valor de la parte derecha del operador a la posición de memoria de la variable de la parte izquierda. El símbolo doble igual comprueba si el valor de la parte derecha es igual al de la parte izquierda, no realizándose ninguna asignación o transferencia de información.

Sección 3.2

1. Una bifurcación abierta ofrece una alternativa que podrá ejecutarse o no dependiendo de una condición. En cualquier caso la ejecución del programa continuará con la próxima sentencia.
2. La sintaxis de la sentencia `if` es la siguiente:

```
if (expresión) sentencia
```

Esto indica que sólo se ejecutará `sentencia` si la expresión es VERDADERA.

3. Una sentencia compuesta consiste de una o más sentencias encerradas entre llaves.
4. Sí, desde una sentencia dentro de un `if` se puede llamar a una función.

Sección 3.3

1. Una bifurcación cerrada obliga al programa a tomar una elección entre dos alternativas.
2. La diferencia entre la sentencia `if` y la sentencia `if...else` es que la primera se corresponde con una bifurcación abierta, mientras que la segunda con una bifurcación cerrada.
3. Sí, se pueden usar sentencias compuestas dentro de una sentencia `if...else`.
4. Sí, se puede llamar a una función desde dentro de una sentencia `if...else`.
5. La sentencia `if...else if...else` permite seleccionar entre tres posibilidades. Las dos primeras dependerán de si se cumplen las condiciones respectivas. Si ambas son FALSAS, se ejecutará la opción correspondiente al último `else`.

Sección 3.4

1. El complemento binario de un número equivale a cambiar en la representación binaria del número los ceros por unos y viceversa.
2. La operación AND binaria entre dos números equivale a aplicar la función AND sobre las correspondientes parejas de bits de las respectivas representaciones binarias de los números.
3. La operación OR binaria entre dos números equivale a aplicar la función OR sobre las correspondientes parejas de bits de las respectivas representaciones binarias de los números.
4. La operación XOR binaria entre dos números equivale a aplicar la función XOR sobre las correspondientes parejas de bits de las respectivas representaciones binarias de los números.
5. Un desplazamiento binario de un número equivale a desplazar la representación binaria del número hacia la derecha o a la izquierda el número de bits correspondiente.

Sección 3.5

1. La operación AND lógica compara el valor lógico de dos expresiones. Si ambas son VERDADERAS, el resultado final será VERDADERO. En caso contrario será FALSO. Se usa el símbolo `&&` para representarlo.
2. La operación OR lógica compara el valor lógico de dos expresiones. Si ambas son FALSAS, el resultado final será FALSO. En caso contrario será VERDADERO. Se usa el símbolo `||` para representarlo.
3. El resultado de aplicar en una expresión lógica el operador NOT lógico es obtener la condición complementaria. Si es VERDADERA devuelve FALSO y viceversa.
4. Un ejemplo que combina operaciones de comparación y lógicas es el siguiente: `(3==3) && (5<10)`.
5. En el procesamiento de una operación AND (OR) sólo se evaluará la expresión de la parte derecha si la de la parte izquierda es VERDADERA (FALSA).

Sección 3.6

1. Mezclar tipos de datos significa realizar operaciones sobre datos de tipos diferentes.
2. El tipo `int` es convertido a `float` y el resultado de la suma es de tipo `float`.
3. La regla que usa para trabajar con tipos de datos diferentes es convertirlos al tipo de mayor rango de la expresión.
4. Una conversión fuerza un cambio en el tipo de un dato.
5. Un valor `i` es una expresión que se refiere a una posición de memoria.

Sección 3.7

1. El propósito de la sentencia `switch` es permitir seleccionar una alternativa entre varias.
2. Las otras palabras reservadas que se usan en la sentencia `switch` son `case`, `break` y opcionalmente `default`.
3. La palabra reservada `default` indica qué sentencia se ejecutará cuando no se cumple ninguna de las condiciones especificadas.
4. Se pueden hacer llamadas a funciones desde una sentencia `switch`.

Sección 3.8

1. La omisión de la sentencia `break` en un `switch` causa que se ejecuten todas las restantes sentencias del `switch`.
2. Para asegurar que se ejecuta siempre la opción asociada a `default`, se deben eliminar todas las sentencias `break`.
3. El operador condicional está compuesto por tres expresiones. Si la primera expresión es VERDADERA, se evaluará la segunda expresión; en caso contrario, se evaluará la tercera.
4. Cualquier valor distinto de 0 hace VERDADERO al operador condicional. Un valor de 0 lo hace FALSO.

Sección 3.9

1. La compilación condicional consiste en que, dependiendo de una condición, ciertas partes del programa se compilarán y otras no.
2. Una directiva de compilación es una orden al compilador especificada en el programa.
3. El propósito de la directiva `#ifdef` es especificar que una determinada sección del código sólo se compilará bajo ciertas condiciones.
4. Las directivas de compilación condicional se usan normalmente en programas grandes.

Sección 3.10

1. El primer paso en el desarrollo de cualquier programa es especificar el propósito del mismo.
2. Un diagrama de reparación sirve para ayudar al técnico durante la reparación de un determinado sistema.
3. Un suplente del programa es una parte del programa que se ha dejado deliberadamente incompleta incluyendo sólo el código necesario para asegurar que el flujo del programa se activará en el momento preciso.
4. Los datos introducidos por el usuario representan para el programa las medidas reales del robot hipotético.

Soluciones de la autoevaluación — Capítulo 3

1. El programa contiene 6 funciones: `main()`, `explicar_programa()`, `brazo()`, `unidad_de_potencia()`, `comprobar_luz()`, `desconexion_dispositivo_brazo()`.
2. No, ninguna.
3. Sí, existe una en la función `desconexion_dispositivo_brazo()`.
4. La función `comprobar_luz()` contiene un `switch`.
5. El significado de la sentencia:

```
medida = (medida > 30) ? 30 : medida;
```

es que si el valor de `medida` es mayor que 30, se fijará a 30. En caso contrario, no se modificará.

6. En el programa se usan dos variables: `medida` y `estado_luz`.
7. Sólo la función `brazo()` devuelve un valor.
8. El valor devuelto es el voltaje introducido por el usuario.
9. La variable `estado_luz` no es de tipo `float` debido a que no se puede usar una variable de este tipo en un `switch`.

Soluciones de los problemas impares del final del capítulo — Capítulo 3

Sección 3.1

1. Los seis operadores de comparación de C son:
 - > Mayor que
 - >= Mayor o igual que
 - < Menor que
 - <= Menor o igual que
 - == Igual a
 - != Distinto de
3. El signo `=` indica asignación, no igualdad.

Sección 3.2

- 5. Una bifurcación abierta ofrece una alternativa que podrá ejecutarse o no dependiendo de una condición. En cualquier caso la ejecución del programa continuará con la próxima sentencia.
- 7. Una sentencia compuesta consiste de una o más sentencias encerradas entre llaves.

Sección 3.3

- 9. Una bifurcación cerrada obliga al programa a tomar una elección entre dos alternativas. En cualquier caso la ejecución del programa continuará con la próxima sentencia.
- 11. La sentencia `if...else if...else` permite seleccionar entre más de dos posibilidades.

Sección 3.4

- 13. Los resultados del complemento binario son: A. 3; B. 1; C. AF.
- 15. Los resultados del OR binario son: A. 7; B. F; C. F.

Sección 3.5

- 17. Una operación lógica es una que producirá un valor entre dos posibles, a los que usualmente se les denomina VERDADERO y FALSO.
- 19. La operación OR lógica compara el valor lógico de dos expresiones. Si ambas son FALSAS, el resultado final será FALSO. En caso contrario será VERDADERO. Se usa el símbolo `||` para representarlo.

Sección 3.6

- 21. Es legal sumar un tipo `int` y un tipo `char`. A este tipo de operación se le denomina mezcla de tipos de datos.
- 23. Una conversión fuerza un cambio en el tipo de un dato.

Sección 3.7

- 25. La sentencia `switch` se usa para seleccionar una alternativa entre varias.
- 27. La palabra reservada `break` se usa para indicar el fin de la opción seleccionada.

Sección 3.8

- 29. La palabra reservada `default` indica qué sentencia se ejecutará cuando no se cumple ninguna de las condiciones especificadas.

Sección 3.9

- 31. A una orden dirigida al compilador se le denomina directiva del compilador.
- 33. Las directivas de compilación condicional más comunes son: `#include`, `#define`, `#ifdef`, `#ifndef` y `#else`.

Sección 3.10

- 35. La característica del lenguaje C que permite desarrollar un programa interactivo que replique el comportamiento de un diagrama de flujo de reparación es el conjunto de sentencias para la toma de decisiones disponibles en este lenguaje.

Sección 3.11

- 37. En general, se debería dividir varias veces el número por la base correspondiente para obtener cada dígito del número convertido.

Respuestas de las revisiones de las secciones — Capítulo 4

Sección 4.1

1. Las cuatro partes importantes de un bucle `for` son:
 - El valor con el que comienza el bucle.
 - La condición bajo la cual el bucle se repite.
 - Los cambios que tienen lugar en cada vuelta.
 - Las instrucciones del bucle.
2. Sí, se puede tener más de una sentencia en un bucle `for` de C. Para ello éstas deben encerrarse entre llaves `{ }`.
3. El significado de `++Y` es que `Y` será incrementado antes de cualquier operación sobre la variable.
4. El operador coma permite tener dos sentencias secuenciales en C.

Sección 4.2

1. La construcción del bucle `while` de C es la siguiente:

```
while (expresión)
    sentencia
```
2. Un bucle `while` se repetirá mientras `expresión` sea VERDADERO (distinto de cero).
3. La condición del bucle se comprueba en primer lugar, antes de ejecutar la sentencia.
4. Un buen uso del bucle `while` tiene lugar en aquellas circunstancias en las que no se sabe el número de veces que se repetirá el bucle.

Sección 4.3

1. La construcción del bucle `do while` de C es:

```
do
    sentencia
while (expresión);
```
2. El bucle `do while` se repetirá mientras `expresión` sea VERDADERO.
3. La condición en el bucle `do while` de C se comprueba después de ejecutar `sentencia`.
4. Normalmente, se prefiere el bucle `while` sobre el bucle `do while`, debido a que se considera mejor práctica de programación comprobar la condición antes de ejecutar, en vez de hacerlo al contrario.

Sección 4.4

1. Un bucle anidado es un bucle de un programa dentro de otro.
2. La estructura que se sigue con los bucles anidados debería dejar claro donde comienza y finaliza cada bucle. Esto se puede hacer sangrando el cuerpo de cada bucle y utilizando comentarios para marcar el fin de cada bucle.
3. Se pueden anidar los tres tipos de bucles de C.
4. El bucle `do` de C siempre se activa al menos una vez. Esto puede provocar problemas si se utiliza como parte de un bucle anidado, puesto que cada vez que se active su bucle externo se activará el bucle `do` al menos una vez, sin importar la condición de su contador.

Sección 4.5

1. Un error en tiempo de ejecución es un error que tiene lugar durante la ejecución del programa.
2. No, un compilador no puede detectar los errores en tiempo de ejecución. La razón es esto es que, por definición, un error en tiempo de ejecución no es un error en la sintaxis del programa, sino que es un error en el diseño del programa.

3. Un bloque de depuración normalmente contiene una sentencia que imprime el valor de algún dato junto con la capacidad de ejecutar la función paso a paso.
4. Una función de auto-depuración es una forma muy conveniente de activar o desactivar la depuración de un programa.

Sección 4.6

1. Una función recursiva es una función que se llama a sí misma.
2. Una función recursiva se implementa mediante el uso de una pila en tiempo de ejecución que contiene una zona donde se almacenan los parámetros de la función en cada invocación a la función recursiva.
3. Una de las funciones debe hacer un `return` para romper la cadena de llamadas recursivas.
4. Si no se para la recursividad, en algún momento determinado toda la memoria disponible se llenará con copias de la función recursiva dando lugar a un error en tiempo de ejecución.
5. La siguiente sentencia `for()` realiza el mismo trabajo que `fact()`:

```
for (i = 1; i <= N; i++)
    fact *= i;
```

Se debe asignar el valor inicial 1 a la variable `fact` antes de ejecutar el bucle `for`.

Sección 4.7

1. Cada letra de moneda (1, 2 o 5) incrementa el valor de la variable `cantidad`.
2. Cualquier carácter distinto de 1, 2, 5 y P será ignorado.
3. El bucle de monedas debe continuar hasta que el usuario introduzca una P con suficiente dinero en la máquina.
4. Se puede añadir dinero hasta que la variable `cantidad` se desborde (32767 es el entero positivo más grande).
5. La variable `cantidad` debe ser al menos 50 para poder seleccionar el producto y salir del bucle.
6. 4 monedas de 25, una de 10 y otra de 5.
7. Si se utiliza `>` en vez de `>=`, la última moneda de cada bucle no será restada. Por ejemplo, si el cambio es exactamente 25 pts, se devolverían dos monedas de 10 pts y un duro, en vez de una moneda de 25 pts.
8. Sí, el orden de los bucles que realizan el cambio es importante. Para obtener el cambio mínimo, es necesario restar en el siguiente orden: monedas de 25 pts, monedas de 10 pts y monedas de 5 pts.

Sección 4.8

1. Sí, los números de Fibonacci pueden generarse sin hacer uso de un bucle mediante la recursividad.
2. El programa debe asignar a la variable `estado` el valor 1 para garantizar que el usuario introduce en la secuencia de entrada un número en primer lugar. También, la sentencia `switch` asume que el sistema se encuentra en un estado conocido y de esta manera requiere que la variable `estado` sea iniciada para todo funcione correctamente la primera vez.
3. Hay 24. Estas son las siguientes (asumiendo que las letras no se repiten):

```
abc abd acb acd adb adc
bac bad bca bcd bda bdc
cab cad cba cbd cda cdb
dab dac dba dbc dca dcb
```

4. Cada una de las tres letras se puede generar por medio de llamadas recursivas, utilizando un contador de letras que detenga la recursividad en la tercera letra.
5. Debido a que cada intento elimina la mitad de los números que quedan, se utilizarán $\log_2 1024$, o 10 intentos máximos, para reducir el intervalo a un único número.

Soluciones de la autoevaluación — Capítulo 4

1. La sentencia

```
int cantidad = 0, D5 = 0, D2 = 0, D1 = 0;
```

asigna a todas las variables declaradas el valor inicial 0.

2. Permitir que 'R' provoque una salida del bucle de monedas. Entonces, sólo se resta 50 de la cantidad introducida si la moneda es distinto de 'R'. El bucle del cambio devolverá a continuación la cantidad introducida.
3. Deben añadirse nuevas variables para representar el número de monedas de cada tipo en la reserva. Los bucles utilizados para los cambios deben modificarse para que el cambio sólo se realice cuando la reserva de monedas de un tipo no se encuentre vacía.
4. Sí, utilizando un entero y el resto de la división.
5. La comprobación es necesaria para que la cantidad introducida no se imprima dos veces cuando el usuario seleccione el producto.

Soluciones de los problemas impares del final del capítulo — Capítulo 4

Sección 4.1

1. Una sentencia compuesta en C es una sentencia que consta de más de una sentencia. Una sentencia compuesta se encuentra encerrada entre llaves { y }.
3. El operador coma permite tener en C dos sentencias secuenciales.

Sección 4.2

5. La construcción del bucle while de C es la siguiente:

```
while (expresión)
    sentencia
```

7. Un bucle while de C se repetirá mientras que *expresión* tenga valor VERDADERO (distinto de cero).

Sección 4.3

9. La construcción del bucle do while de C es la siguiente:

```
do
    sentencia
while (expresión);
```

11. Un bucle do while de C se repetirá mientras que el valor de *expresión* sea VERDADERO (distinto de cero).

Sección 4.4

13. Un bucle anidado es un bucle que contiene dentro otro bucle.
15. Sí, existe un problema cuando se anida el bucle do de C. El bucle do de C siempre se será ejecutado al menos una vez por el bucle externo.

Sección 4.5

17. Una función de depuración normalmente contiene un valor a imprimir para algún dato y la capacidad de ejecutar la función paso a paso.

Sección 4.6

19. Se necesita una pila en tiempo de ejecución para dar soporte a la recursividad.
21. Para que la memoria no se llene con copias de una función recursiva y el sistema falle.

Respuestas de las revisiones de las secciones — Capítulo 5

Sección 5.1

1. La memoria de un computador puede representarse gráficamente como una pila de posiciones de memoria, cada una de ellas identificada por un número distinto.
2. Una instrucción causa una acción del computador, mientras que un dato es el valor sobre el que actúa una instrucción.
3. Una dirección es un número que representa una posición de memoria determinada.
4. En el direccionamiento inmediato, el dato se almacena inmediatamente después de la instrucción.
5. En el direccionamiento directo, la instrucción contiene la dirección de memoria dónde está almacenado el dato.

Sección 5.2

1. Una palabra es un grupo de bits que se tratan como una unidad.
2. El tamaño de un tipo `char` es de 1 octeto.
3. Los números con signo se representan con la notación complemento a 2.
4. La diferencia está en que para los números sin signo no se considera al BMS como un bit de signo. Así, aunque el número de valores representables sea el mismo en ambos casos, el máximo valor representable es más grande en el caso de la representación sin signo.
5. El tipo que ocupa menos memoria es el `char` y el que más es el `long double`.

Sección 5.3

1. Un puntero es un tipo de datos que representa la dirección de otra posición de memoria.
2. Se le denomina de esta forma puesto que se puede considerar que apunta a otra posición de memoria.
3. Se le asigna aplicando el operador `&` a dicha variable. Por ejemplo, para almacenar la dirección de una variable `x` en un puntero `p`, se ejecutaría lo siguiente: `p = &x`.
4. Para pasar un valor usando punteros, primero se debe asegurar que el puntero contiene la dirección de una variable `y`, a continuación, se debe aplicar el operador `*` precediendo a la variable puntero: `*p = 12`. Si `p` estaba apuntando a la variable `x`, de esta forma se habrá asignado un valor 12 a `x` mediante el uso del puntero.
5. Un puntero se declara colocando un `*` justo delante del nombre de la variable puntero.

Sección 5.4

1. Las dos formas de devolver un valor desde una función hacia la función que la invocó son: usar la sentencia `return` o utilizar un puntero como argumento de la función.
2. La limitación de la sentencia `return` para devolver valores a la función que realizó la llamada es que sólo puede devolver un valor.
3. Para pasar más de un valor desde una función a la función que la invocó, se deben usar punteros como argumentos de la función.
4. El mecanismo para pasar valores a una función usando punteros como argumentos consiste en utilizar las direcciones (`&`) de las variables reales que recibirán los valores devueltos por la función.
5. Las funciones separadas facilitan un buen diseño del programa.

Sección 5.5

1. El término variable local se refiere a una variable declarada en una función que sólo es visible desde la misma.

2. El término «ámbito» aplicado a una variable se refiere a la parte de programa desde donde es visible la variable.
3. Una variable global se declara al principio del programa antes de la función `main()`, mientras que una local se declara dentro de la función que la usará.
4. El uso de variables globales no se considera una buena técnica de programación ya que dificulta la comprensión y depuración del programa.
5. Se pueden pasar valores entre funciones como argumentos de las mismas.

Sección 5.6

1. La palabra reservada `const` aplicada a un dato causa que su valor no se pueda modificar durante la ejecución del programa.
2. Una variable automática es una variable local a una función cuya vida queda restringida a la activación de la función.
3. Una variable estática es una variable local a una función cuya vida coincide con la del programa.
4. Una variable de registro es una variable que indica al compilador que debe intentar almacenarla en un registro del procesador.

Sección 5.7

1. El operador de dirección se representa mediante el símbolo `&` y devuelve la dirección de la variable a la que se le aplica.
2. El operador de indirección se representa mediante el símbolo `*`. Cuando se aplica a una variable, trata el valor almacenado en ella como la dirección de un dato.
3. El operador de igualdad es `==` y el de asignación `=`.

Soluciones de la autoevaluación — Capítulo 5

1. La salida producida por cada una de las llamadas a `printf()` es la siguiente:

```
La constante es 57532.
El valor de posicion_de_memoria_1 es 375
El contenido de este_valor = b
El resultado es F.
```

2. Hay dos valores globales al programa:

```
const unsigned int numero_1 = 57532;
int *apunta_a;
```

3. Hay sólo un valor que es global a una parte del programa:

```
extern char nuevo_valor;
```

Su ámbito abarca todas las funciones que aparecen después de su declaración.

4. El valor 375 se le asigna a la variable `posicion_de_memoria_1` mediante la siguiente sentencia:

```
*apunta_a = 375;
```

5. La sentencia en `main()` que causa que la variable `posicion_de_memoria_1` valga 375 es:

```
funcion_1();
```

6. La salida del último `printf()` es F debido a que se realiza un AND binario entre:

```
1111 <= 15
1111 <= 15
-----
1111 ==> 15 (F en base 16)
```

7. La función `funcion_1` no requiere punteros como argumentos ya que usa una variable global. La función `funcion_2` usa punteros puesto que necesita devolver más de un valor a la función que la invocó.
8. No, la función `funcion_1` no necesita ser del tipo `double` ya que no devuelve ningún valor.
9. La función `funcion_1` puede acceder a la variable `apunta_a` por ser ésta global.
10. La función `funcion_1` modifica el valor de `posicion_de_memoria_1` a través del puntero `apunta_a`:

`apunta_a = &posicion_de_memoria_1;`

Soluciones de los problemas impares del final del capítulo — Capítulo 5

Sección 5.1

1. La memoria de un computador puede representarse gráficamente como una pila de posiciones de memoria, cada una de ellas identificada por un número distinto.
3. El proceso de leer una instrucción de memoria y ejecutarla realizado por la UCP se denomina ciclo lectura/ejecución.

Sección 5.2

5. Algunos de los más típicos tamaños de palabra son los siguientes: 8, 16, 32 y 64 bits.
- | | | |
|------------|---------|---------|
| 7. A. 0110 | B. 1000 | |
| 9. A. 1111 | B. 0001 | C. 0110 |
| 11. A. -6 | B. -8 | C. -100 |

Sección 5.3

13. El operador de indirección se representa mediante el símbolo `*`. Cuando se aplica a una variable, trata el valor almacenado en ella como la dirección de un dato.
15. A. El valor de puntero es la dirección de dato.
B. El valor de dato es 5.
C. El valor de `*puntero` es 5.

Sección 5.4

17. Las funciones separadas facilitan un buen diseño del programa.
19. El operador de dirección se representa mediante el símbolo `&` y devuelve el valor de la dirección de la variable a la que se le aplica.

Sección 5.5

21. Una variable declarada en una función cuya vida está restringida a la activación de la función se denomina automática.
23. Una variable declarada al principio del programa antes de la función `main()` se denomina global y su ámbito abarca todo el programa.

Sección 5.6

25. La palabra reservada `const` aplicada a un dato causa que su valor no se pueda modificar durante la ejecución del programa.
27. Una variable que indica al compilador que debe intentar almacenarla en un registro del procesador se denomina variable de registro.

Sección 5.7

29. Se puede obtener la dirección de dato usando el operador de dirección: `&dato`.
31. El operador de igualdad es `==` y el de asignación `=`.

Respuestas de las revisiones de las secciones — Capítulo 6

Sección 6.1

1. Una cadena de caracteres es un vector de caracteres.
2. Se indica una cadena de `char` en C usando llaves cuadradas `{ }`.
3. Para una cadena de caracteres de 5 caracteres, se necesitan seis elementos del vector. El último elemento del vector contiene el carácter nulo, necesario en C para que el computador sepa donde termina una tira de caracteres en memoria.
4. El número de elemento del primer elemento de una tira de caracteres de C es 0.
5. La relación entre punteros y elementos de una tira de caracteres puede usarse para acceder a elementos individuales de la cadena de la misma forma que el índice del vector puede usarse para acceder a elementos individuales.

Sección 6.2

1. Hay veintisiete caracteres en memoria. Veintiséis son los caracteres alfabéticos y uno es el carácter nulo `'\0'`.
2. Sólo la cadena de caracteres "Oh," será leída.
3. `'\0'` representa una tira con los caracteres `'\'`, `'0'` y `'\0'`, y tiene una longitud de 2. `'\0'` representa el carácter NULO y tiene una longitud de 0.
4. Usar `char string[80] = "Hola"` ; permite a la cadena de caracteres crecer hasta 80 caracteres, mientras que `char string[] = "Hola"` ; proporciona almacenamiento para los cinco caracteres entre los corchetes (más el nulo).

Sección 6.3

1. Usar `gets()` permite que se lea toda una cadena de caracteres de un golpe. Si se reemplaza por `scanf()` sólo se leería hasta el primer blanco. Si el usuario introduce su nombre completo, sólo se leería el nombre propio.
2. Cuando se pasa una cadena de caracteres a una función, sólo se pasa la dirección del primer carácter de la cadena. La cadena puede ser de cualquier longitud, pero debe terminar con un carácter nulo (`'\0'`).
3. Si una tira de caracteres no tiene un carácter nulo, cualquier función que acceda a la tira buscará en la memoria hasta que encuentre uno. Esto incrementaría artificialmente la longitud de la tira de caracteres y conduciría a ejecuciones impredecibles.

Sección 6.4

1. La función `getchar()` espera hasta que el usuario pulse una tecla del teclado. Devuelve el código ASCII de la tecla apretada.
2. La clasificación de los caracteres en C es:

alfanumérico, alfabético, control, dígito, imprimible, minúscula, puntuación,
mayúscula, espacio y hexadecimal.

Ejemplo de varios de estos tipos son:

<i>Carácter(es)</i>	<i>Clasificación (es)</i>
'a' a 'z'	Alfanumérico, alfabético, minúscula, imprimible
'A' a 'Z'	Alfanumérico, alfabético, mayúscula, imprimible
'0' a '9'	Alfanumérico, dígito, hexadecimal
'.'	Puntuación

3. El programa imprimirá todos los caracteres que no son caracteres de puntuación o dígitos.
4. Sí, si se usa la función `isalpha()`.
5. En un PC, el entero 40.000 necesita 16 bits, con el bit más significativo a la derecha. Este nivel alto es interpretado por la función `printf()` como el entero con signo -25.536.

Sección 6.5

1. La sentencia `#include <string.h>` es necesaria para que las funciones de manejo de tiras de caracteres estén disponibles en un programa.
2. `strcat`, `strlen`, `strcpy`, `strstr`, `strcmp`
3. La función `strlen()` busca en memoria hasta que encuentra un carácter NULO. Si el carácter NULO no está en la posición adecuada, la longitud de la cadena será inválida.
4. Para realizar una sustitución de cadenas, la cadena de entrada "abcde" debe ser buscada (con `strstr()`) hasta encontrar una cadena "bc". Si se encuentra, se sustituye "bc" por "hola". Esto significa usar de forma inteligente la función `strcpy()`.
5. La longitud de la primera cadena debe ser lo suficientemente larga como para contener a la cadena que va a ser concatenada.
6. La función `strchr()` es sensible a mayúsculas y minúsculas porque las considera diferentes. Por tanto, 'a' y 'A' no son el mismo carácter.
7. `strchr`, `strstr`, `strcmp`.

Sección 6.6

1. Un vector de caracteres definido como `[7][10]` necesita 70 posiciones de memoria para almacenar los caracteres.
2. Defina una variable entera nueva para seguir la pista de las comparaciones. Incremente la variable dentro del segundo bucle `for()`. Experimente con diferentes ordenaciones de las cadenas de entrada.
3. La matriz queda como sigue después de cada ejecución de bucle:

susana	miguel	javier	juan	javi
miguel	javier	juan	javi	susana
javier	miguel	juan	javi	susana
javier	juan	miguel	javi	susana
javier	juan	javi	miguel	susana
javier	javi	juan	miguel	susana
javi	javier	juan	miguel	susana

(¡no hay cambios!)

4. La ordenación por burbuja contiene un par de bucles `for` anidados. Como se hace una comparación en cada pasada del bucle interior se hacen aproximadamente N^2 .
5. La potencia de las matrices rectangulares es que se puede concatenar las cadenas hasta una longitud fija. Su debilidad es que se malgastan posiciones de memoria cuando hay cadenas de distinta longitud. La potencia de las matrices irregulares es que almacenan las cadenas de forma muy eficiente. Su debilidad es su incapacidad para usar algunas funciones predefinidas (como `strcat`) sin peligro de sobrescribir el espacio de otras cadenas de caracteres.
6. El nombre "javiju" resulta de la secuencia de sentencias `strcpy()` usadas para cambiar "javier" y "juan" en la matriz irregular. Como cada cadena tiene una longitud predefinida, C creó un puntero al principio de cada subcadena de la matriz. Estos punteros no se actualizan durante el `strcpy()` y, por tanto, apuntan a posiciones equivocadas de la matriz.

Sección 6.7

1. El formato de texto es deseable por varias razones: da a la salida un aspecto profesional, mejora su legibilidad y permite al usuario introducir texto sin formatear.
2. La longitud de la siguiente palabra se determina buscando la cadena de entrada hasta el siguiente espacio (o salto de carro).
3. No se necesita formatear nada si el almacén de salida está completamente lleno de texto.

4. Cambiar el valor ANCHO hace que la salida formateada tenga un ancho que se corresponde con ANCHO. Esto puede originar más líneas de salida si se reduce el valor de ANCHO y menos si se aumenta.
5. Si se cambia ANCHO a 10, algunas palabras del texto sobrepasarán este valor y los resultados serán impredecibles.
6. Sí, ANCHO debería ser, como mínimo, tan grande como la palabra más larga del texto. El programa puede leer el texto, determinar este valor mínimo y parar el formateo si ANCHO es demasiado pequeño.
7. Los blancos se insertan en `expandir_linea()` añadiendo un blanco entre cada dos palabras del texto de salida.
8. Una matriz irregular puede ayudar a expandir el texto si cada palabra de la salida se almacena en su propia posición de la matriz. Dicha matriz podría usarse para crear el texto formateado con múltiples operaciones `strcat()`.

Sección 6.8

1. El valor `0x30` es código ASCII asociado al dígito '0'. Restando este valor de cada dígito ASCII se obtiene el valor numérico.
2. La expresión en post-orden es: `20 2 6 4 - / 3 1 7 + * + 4 / -`
3. La pila post-fija es una cadena de caracteres. Añadir o eliminar elementos de la pila se hace a través de la variable de índice `pfsidx`.
4. Si se hace el AND de un código ASCII de minúscula con el patrón `0xdf`, se obtiene la letra mayúscula equivalente.
5. Los punteros `lchar` y `rchar` comienzan en los extremos opuestos del vector de entrada. A medida que los caracteres a los que apuntan coinciden, se avanzan una posición hacia el centro de la cadena de caracteres. Cuando se sobrepasan, la comprobación está completa.
6. Un analizador léxico rompe una expresión de entrada en sus componentes básicos. Por ejemplo, la sentencia

```
int contador = 0;
```

contiene 5 elementos léxicos: `int, count, =, 0, ;`.

7. La codificación por transposición del vector "ken esta aqui" origina el siguiente resultado:

```
ken
est
aaq  => "keaesaintq"
ui
```

Soluciones de la autoevaluación — Capítulo 6

1. La X se usa en el código de ISBN cuando el último resto es 10. La última sentencia `if` genera la X cuando ocurre esto.
2. Las prioridades se determinan examinando el valor del puntero de pila (`pfsidx`) y los datos de la cima de la pila. Si la pila está vacía no hacen falta prioridades. En caso contrario, se fijan las prioridades de acuerdo a su orden (mayor o menor): `(, *, /, +, -`.
3. Cuando se encuentra un ')' se vacía la pila hasta que se encuentra un '('. Todos los elementos extraídos de la pila se escriben en la salida.
4. Las variables `vocales` y `contador` se declaran como variables `static int` de forma que automáticamente se inician a cero.
5. Un palíndromo se ha comprobado completamente cuando `lchar` es igual a `rchar`.
6. Comprobar un componente léxico del vector dobles necesita el uso de lectura adelantada de un carácter cuando se leen los componentes. Esto implica que hay que manipular cuidadosamente el flujo de entrada.

7. Cuando se ve una comilla doble, se extrae una tira de caracteres leyendo todos los caracteres de la entrada hasta que se encuentra otra comilla doble.
8. La matriz de codificación se llena por columnas.

Soluciones de los problemas impares del final del capítulo — Capítulo 6

Sección 6.1

1. El índice del primer carácter de una tira de caracteres es [0].
3. Una tira de caracteres en C contiene cualquier número de códigos ASCII terminados por un carácter NULO.

Sección 6.2

5. Tres técnicas de iniciación de la tira de caracteres "datos" son:
 1. `char cadena[] = "datos";`
 2. `char cadena[5] = "datos";`
 3. `char cadena[] = {'d', 'a', 't', 'o', 's'};`
7. Un blanco en la entrada hace que la función `scanf()` deje sin leer el resto de la cadena de entrada.

Sección 6.3

9. El espacio de almacenamiento de una cadena de caracteres debe reservarse en la función que llama.

Sección 6.4

11. `getchar()` no hace eco de los caracteres de la entrada.

Sección 6.5

13. `strlen()` busca la memoria hasta que encuentra un carácter NULO, dando una longitud del vector errónea.
15. Javi está antes que Javier en la lista telefónica. Si dos cadenas de la distinta longitud son idénticas en su parte común, la de menor longitud está primero.
17. `strexec()` (string execute) podría ser útil. Esta función ejecutará una expresión matemática recibida como entrada. Por ejemplo, `strexec("2+3*4")` daría 14. `strdel()` (string delete) podría ser también útil, esta función borrará un vector de entrada todas las veces que lo encuentre en un texto. Así, `strdel("infeliz", "in")` daría "feliz".

Sección 6.6

19. Las llaves `{ }` se usan para separar cadenas individuales.
21. Una matriz irregular es una matriz bidimensional de cadenas de longitud desigual.

Sección 6.7

23. Para expandir una línea de texto hay que determinar el número de blancos. Luego hay que insertar los blancos entre las palabras en las líneas de texto hasta que están totalmente expandidas.

Respuestas de las revisiones de las secciones — Capítulo 7

Sección 7.1

1. En C se indica el número de elementos que tendrá un vector poniendo la dimensión del mismo entre corchetes: `[N]`, donde N es el número de elementos.
2. El índice del primer elemento de un vector en C es siempre 0.

3. Si se declara `int valor[5]`, entonces `valor`, `&valor` y `&valor[0]` son iguales. Todos contienen la dirección de comienzo del vector.
4. Un vector global está iniciado. Un vector local no.
5. Un vector `char` necesita 1 byte por elemento y un vector `int` necesita 2 bytes por elemento (en un PC) o 4 bytes por elemento en una arquitectura de 32 bits.

Sección 7.2

1. La idea básica subyacente en las aplicaciones con vectores es manipular el valor del índice del vector.
2. El método de programación usado para obtener los elementos de un vector desde el programa de usuario es usar un bucle `for` de C e incrementar el índice del vector.
3. El método usado para que una serie de valores de entrada se muestren en orden inverso al de entrada es usar un bucle `for` de C que incremente el índice del vector y otro bucle `for` de C que decremente el índice del vector que se imprime.
4. El método usado para extraer un valor mínimo de una lista de valores de entrada es comparar dichos valores con los otros. Para ello se pone el primer valor en una variable y se compara con el valor de los elementos del vector usando un bucle `for` de C. Si el elemento del vector es menor, se cambia el valor de la variable con el de este elemento.
5. Se busca la lista completa de valores para encontrar el valor mayor. Para ello, se inicia una variable con el primer elemento de la lista y se compara con el resto de los elementos. Cada vez que se encuentra un elemento mayor, se cambia el valor de la variable al del elemento.

Sección 7.3

1. La ordenación por mezcla rompe los valores iniciales en grupos individuales. A continuación ordena cada grupo de forma individual, y, posteriormente, mezcla los grupos ordenados. El número total de comparaciones es menor que el requerido por el método de la burbuja porque no todos los elementos del vector deben ser comparados con todos los demás.
2. Cualquier método de ordenación debe comparar al menos una vez todos los números. Si están en orden correcto, la ordenación termina con una sola pasada.
3. El tipo de comparaciones usadas para controlar el `switch` de los elementos determina la secuencia de la ordenación.
4. a) Ordenación por burbuja:

7	3	1	8	Primera pasada
3	1	7	8	Segunda pasada
1	3	7	8	Tercera pasada
1	3	7	8	Cuarta pasada

b) Ordenación de la cubeta: sólo hace falta una pasada para llenar las cubetas.

c) Ordenación por mezcla:

8	7		3	1	Primer nivel de recursividad
7	8		1	3	Segundo nivel de recursividad
1	3	7	8		Mezcla de primer nivel

5. La ordenación de la cubeta es eficiente porque necesita una única pasada para ordenar todos los números. Está limitada por el hecho de que el número mayor debe ser conocido por adelantado.
6. Se necesitan tres niveles de recursividad:

Nivel 1:	0	3	8	1	9		2	6	4	7	5				
Nivel 2:	0	3		8	1	9		2	6		4	7	5		
Nivel 3:	0	3		8		1	9		2	6		4		7	5

7. Las dos técnicas usan la recursividad de forma distinta para obtener una lista ordenada. La mezcla siempre divide la lista en dos partes iguales. La ordenación rápida divide la lista en dos mitades, una contiene los elementos menores que el pivote y la otra los mayores. El valor del pivote puede originar niveles extra de recursividad cuando no divide la lista en partes iguales.
8. No hay diferencia entre ordenar números y cadenas de caracteres, porque ambos se representan con valores binarios en memoria.

Sección 7.4

1. Una forma de iniciar un vector es usar llaves {}. Todos los elementos del vector, separados por comas, se ponen entre las llaves.
2. Para declarar un vector con más de una dimensión (matriz), se debe usar los corchetes [] para dimensión adicional que se necesite.
3. Un parámetro formal de una matriz en C debe tener los tamaños de las otras dimensiones además de la que se manipula.
4. El método usado en los programas de esta sección para sumar variables de tipo vector es usar el índice del vector (vector[indice]) y el operador +=.
5. Antes de usar el operador += en un vector sin iniciar es necesario estar seguro de que todos sus elementos tienen el valor 0.
6. a) [1][3] b) [3][1] c) [2][4] d) Ilegal

Soluciones de la autoevaluación — Capítulo 7

1. La sentencia de detección de intercambio hace que el programa termine cuando no ha habido ninguno en una pasada. Esto origina menos comparaciones cuando la lista está ordenada parcialmente inicialmente.
2. Si no se usa valor_temp, uno de los dos elementos que están siendo intercambiados será sobrescrito.
3. Una vez que el vector de la cubeta ha sido cargado, se imprimirán los números de entrada ordenados en una sola pasada indicando el número de índice donde se ha puesto cada elemento ordenado.
4. Es importante que el vector de la cubeta iniciado contenga un valor que no esté en la lista de los números de entrada. Para usar números positivos y negativos en la ordenación de la cubeta, se pueden usar dos vectores para mantener los grupos de números positivos y negativos. El vector de números negativos se recorre desde el índice más alto al más bajo para mostrar los números negativos ordenados.
5. La ordenación por burbuja necesita un único bloque de almacenamiento de N elementos porque los números se ordenan intercambiándolos. En la ordenación por mezcla, cada nivel nuevo de recursividad necesita espacio de almacenamiento para todo el vector.
6. El punto medio del vector es elegido sumando los valores de los índices en cada extremo y dividiéndolos por 2.
7. El pivote determina cómo se divide el vector en dos subvectores.
8. El elemento medio de un vector se determina ordenando el vector y tomando el elemento del medio.
9. La función de comparación se define de tal forma que se puede usar elementos de cualquier tipo como entrada de la función qsort(). El tipo de elemento se define en tiempo de compilación. Por tanto, qsort() es capaz de ordenar muchos tipos diferentes de números.
10. La ordenación por mezcla es más eficiente, porque rompe el vector en subvectores del mismo tamaño con un número mínimo de llamadas recursivas. Con qsort() el valor pivote no parte el vector en subvectores de la misma longitud en ningún nivel de recursión, debido a que el vector está ordenado en orden descendente. Por tanto, son necesarios N niveles de recursividad con N-1 comparaciones en cada nivel.

Soluciones de los problemas impares del final del capítulo — Capítulo 7

Sección 7.1

1. Un vector numérico es un grupo de números que se acceden a través de una variable tipo vector.
3. El índice del primer elemento en un vector de 10 elementos es [0]. El índice del último elemento es [9].

- Una declaración de vector estático inicia todos sus elementos a cero.

Sección 7.2

- Utilice un bucle para examinar cada elemento del vector. Reemplace el máximo actual con cada elemento mayor que él.

Sección 7.3

- Las diferencias son las siguientes:

La ordenación por burbuja mueve el elemento mayor al final del vector después de cada pasada. Habitualmente necesita múltiples pasadas.

La ordenación de la cubeta pone indicadores en el vector de ordenación por cada número en el vector de entrada. Necesita una sola pasada.

La ordenación por mezcla rompe el vector de entrada en múltiples subvectores del mismo tamaño. Las comparaciones se hacen sólo sobre los subvectores de dos elementos. Entonces se mezclan los subvectores con comparaciones adicionales. Habitualmente se necesitan múltiples niveles de recursividad.

La ordenación rápida parte el vector de entrada en subvectores usando un pivote. Los vectores pueden ser de distinto tamaño. Los elementos pivote terminan en su posición correcta del vector ordenado. Habitualmente se necesitan múltiples niveles de recursividad.

- Dos vectores ordenados se mezclan comparando sus elementos uno a uno. El menor elemento del vector se escribe en el vector de salida y se avanza el puntero al menor elemento. Cuando el puntero alcanza el final de cualquier vector, se copian los elementos restantes del otro.
- La elección de un pivote determina cómo se parte el vector en cada nivel de recursividad. para minimizar el número de niveles, el pivote de cada nivel debería ser el valor medio de cada subvector.

Sección 7.4

- Se pasa un puntero a la matriz.

Sección 7.5

- El enlace dinámico usa una FAT para almacenar los sectores asignados. Cada entrada de una cadena apunta al siguiente (o el último) sector.
- Se necesita un vector para almacenar los números porque no se pueden calcular las diferencias hasta que se conoce la media.

Respuestas de las revisiones de las secciones — Capítulo 8

Sección 8.1

- En C, la palabra reservada `enum` significa enumerado.
- Un tipo de datos `enum` se usa para describir un conjunto discreto de valores enteros.
- Para el código `enum numeros (primero, segundo, tercero)`, el valor entero de primero es 0.
- El propósito principal de usar un tipo `enum` en C es hacer el código más legible.
- Sí, a una constante `enum` en C se le puede asignar un valor entero. Un ejemplo es el siguiente: `enum numeros (primero = 15)`.

Sección 8.2

- La palabra reservada `typedef` de C permite dar nombres particulares a un tipo de datos existente.
- El nombre del tipo de datos resultante del código `typedef char nombre[20]` es `nombre`.
- El principal propósito de usar `typedef` en C es crear sinónimos para los tipos de datos existentes.

Sección 8.3

1. La información de comprobación de una cuenta puede ser tratada como una estructura porque contiene información que es una colección de diferentes tipos de datos relacionados lógicamente.
2. Una estructura en C es una colección de diferentes tipos de datos relacionados lógicamente.
3. La estructura de una estructura C como la presentada en esta sección se define con la palabra reservada `struct` seguida por una llave de apertura `{` y terminada con una llave de cierre seguida por un punto y coma `;`. Entre las llaves se ponen los miembros de la estructura:

```
tipo identificador de variable;
```

4. La variable de un miembro de una estructura se programa para obtener datos y mostrarlos mediante el uso del operador de miembro `.`.
5. Un ejemplo del problema 4 sería:

```
nombre_estructura.nombre_miembro
```

Sección 8.4

1. Una etiqueta de estructura es un identificador que nombra el tipo de estructura definida en lista de declaración de miembros de la estructura.
2. Un ejemplo de etiqueta de estructura sería:

```
struct etiqueta
{
    lista-miembros-declarados;
}
```

3. Una estructura puede ser de tipo variable. Esto puede usarse usando el `typedef` de C y la identificador de variable de la estructura con el nombre del tipo.
4. Un miembro de estructura se referencia con punteros mediante el uso del operador `->`.
5. Las tres operaciones que se permiten sobre estructuras son:
 - Asignar una estructura a otra con el operador asignación `=`.
 - Acceso a un miembro de la estructura `(. o ->)`.
 - Obtener la dirección de la estructura (usando el operador `&`).

Sección 8.5

1. El propósito de una `union` de C es permitir que varios tipos de datos se almacenen en la misma posición inicial de memoria.
2. Una `union` de C se declara de la misma forma que una estructura de C. La diferencia es que la palabra clave `union` se pone en lugar de `struct`.
3. Una estructura de C puede ser iniciada por el programa. Sin embargo, debe ser global o estática.
4. Un vector de estructuras puede declararse de la misma forma que cualquier vector. La diferencia es que el tipo de vector se declara como una estructura con tipos definidos.
5. Un miembro individual de un vector de estructuras se accede con `variable_estructura[N].variable_miembro`, donde N es el índice del vector.

Sección 8.6

1. Una estructura puede ser un vector. Puede tratarse como un tipo de datos de igual forma que un vector.
2. Una estructura puede contener a otra estructura. Un tipo estructura definida en C puede usarse como miembro de cualquier otra estructura.
3. Sí, es posible para un vector ser un tipo estructura que contiene un vector en uno de sus miembros. Puesto que se puede tener una estructura como un vector y una estructura puede contener a otra estructura, se pueden combinar.

Sección 8.7

1. Un nodo está formado habitualmente por dos partes: campo de datos y un puntero.
2. Una lista enlazada de caracteres puede extenderse a través de distintas posiciones de memoria. Una cadena de caracteres se almacena como un bloque único y contiguo de memoria. La cadena de caracteres tendrá un número fijo de posiciones asignadas (incluyendo una para `'\0'`) y, por tanto, no se puede extender fácilmente. Una lista enlazada de caracteres puede extenderse fácilmente añadiendo más nodos.
3. En un nodo puede almacenarse cualquier tipo de datos.
4. El puntero NULO indica el fin de una lista enlazada (o algunas veces un nodo iniciado).
5. La asignación de memoria en tiempo de ejecución se hace mediante la función `malloc()`.
6. Las operaciones básicas de cada tipo de datos avanzados son:

Lista enlazada:	Insertión, borrado, búsqueda
Pila:	Insertar, extraer
Cola:	Encolar, desencolar
Árbol binario:	Añadir nodo (árbol), borrar nodo (árbol), búsqueda

7. FIFO significa Primero en Entrar Primero en Salir (First In First Out) y describe las operaciones básicas en una cola. LIFO significa Último en Entrar Primero en Salir (Last In First Out) y describe las operaciones básicas en una pila.

Soluciones de la autoevaluación — Capítulo 8

1. El juego de instrucciones se define usando una sentencia `enum`.
2. No, no es significativo. Las instrucciones se pueden definir en cualquier orden.
3. El acumulador debe definirse como una variable global porque debe estar disponible para las llamadas recursivas a `exec()`.
4. `exec(50)` hace que MiniMicro lea la instrucción de la dirección 50 y la ejecute.
5. La variable `iop` se ignora cuando se ejecuta `PRINT`, `RET` o `STOP`.
6. Es necesario añadir `AND`, `OR` y `XOR` a `isct`, cada una con su propia sentencia de caso. Por ejemplo, `AND` necesita:

```
case AND: acc &= iop; break;
```

7. Use `acca` y `accb` para los nombres de los dos acumuladores. Añada nuevas instrucciones que especifiquen el acumulador A o el B, como `ADDA` o `ADDB`.
8. La instrucción `JNZ` se ejecuta mediante la sentencia:

```
case JNZ: ip = (acc != 0) ? iop : ip; break;
```

Si el acumulador es cero, `ip` no cambia. En caso contrario, `ip = iop` y se obtiene el efecto de un salto a la dirección de `iop`.

9. El programa muestra 10 9 8 7 6 5 4 3 2 1 0 y para.

Soluciones de los problemas impares del final del capítulo — Capítulo 8

Sección 8.1

1. El tipo de datos `enum` (enumerado) de C se usa para describir un conjunto discreto de valores enteros.
3. No, el `enum` de C no crea un nuevo tipo de datos. Su propósito es hacer el código fuente más legible.

Sección 8.2

5. No, el `typedef` de C no crea un nuevo tipo de datos. Permite dar un nombre particular a un tipo existente.
7. El nombre nuevo resultante del tipo de datos para el código dado es estructura.

Sección 8.3

9. Un ejemplo de un sistema que usa habitualmente el concepto de estructura de C es una agenda personal.
11. El operador de miembro de C identifica la estructura a la cual pertenece el miembro. El símbolo es el punto (`.`).

Sección 8.4

13. Un ejemplo de uso de una etiqueta de estructura es:

```
struct etiqueta
{
    lista_declaracion_miembros;
}
```

15. Las tres operaciones permitidas con estructuras son asignar una estructura a otra, acceder a un miembro de la estructura y obtener la dirección de la estructura con el operador `&`.

Sección 8.5

17. Una estructura de C y una unión se declaran de la misma forma, la diferencia es que se usa la palabra reservada `union` en lugar de `struct`.
19. El código representa un miembro (identificado como miembro) del tercer elemento (2) de un vector de estructuras llamado variable.

Sección 8.6

21. Sí, un vector de estructuras en C puede contener un vector entre sus elementos. Como se puede tener una estructura como un vector y una estructura puede contener vectores entre sus miembros, se pueden combinar.

Sección 8.7

23. Los tamaños de los campos de datos y del puntero determinan el tamaño de un nodo, siendo los campos de datos los más sujetos a cambios en la definición del nodo.
25. Inserción en la cabeza o la cola: hay que obtener un nuevo nodo e iniciarlo; para insertarlo en la cabeza de la lista, se asigna al puntero del nodo nuevo la dirección de la cabeza de la lista, la dirección del nodo nuevo es ahora la cabeza de la lista. Para insertar al final de la lista, el campo del puntero del nodo nuevo se pone a NULO y el puntero de la cola se apunta al nodo nuevo. La dirección del nodo nuevo es ahora la cola de la lista.
Para borrar la cabeza, se asigna al puntero de la cabeza la dirección del puntero del primer nodo y se borra el nodo de cabeza. Para borrar por la cola, se busca el nodo penúltimo y se pone su puntero a NULO, la dirección de este nodo es el fin de la cola. A continuación se borra el nodo liberado (con `free()`).
27. La recursividad se usa para acceder un nodo en un árbol binario. Para ello se pasa el puntero al hijo izquierdo o derecho en cada nueva llamada de acceso al árbol binario. La recursividad termina cuando el puntero está en NULO.

Respuestas de las revisiones de las secciones — Capítulo 9

Sección 9.1

1. Los nombres de archivos en DOS tienen el formato siguiente:

```
[Dispositivo:]Archivo[.EXT]
```

Donde

Dispositivo = El nombre del dispositivo que se corresponde con el disco al que se quiere acceder.

Archivo = El nombre del archivo (no más de ocho caracteres).

.EXT = El nombre del archivo tiene una extensión opcional (no más de tres caracteres).

Tanto el dispositivo como la extensión son opcionales. Si no se especifica el dispositivo, se usará el dispositivo activo en ese momento. El nombre del dispositivo siempre irá seguido por dos puntos (:).

2. Un puntero a archivo es un puntero de tipo FILE. Por ejemplo, FILE *ptr_arch. 3. La función para abrir un archivo es `fopen()`.
4. El puntero a archivo se asigna mediante la siguiente sentencia:

```

puntero_a_archivo = fopen("ARCHIVO.EXT", "r");

```
5. La diferencia entre cómo se abre un archivo para leer y cómo se abre para escribir es que en el primer caso se especifica "r" en su apertura, mientras que en el segundo se especifica "w".
6. Cuando se termina de usar un archivo, se debe cerrar usando la función `fclose()`.

Sección 9.2

1. Las cuatro posibles condiciones que pueden aparecer cuando se trabaja con un archivo son las siguientes:
 - El archivo no existe y se quiere crear.
 - El archivo existe y se quiere leer.
 - El archivo existe y se desea añadir información al final del mismo.
 - El archivo existe y se desea escribir la nueva información sobre la antigua.
2. Los diferentes métodos para leer y escribir datos disponibles en C son:
 - Un carácter cada vez.
 - Cadenas de caracteres.
 - Modo mixto.
 - Bloque (o estructura).
3. Las tres formas básicas de apertura de un archivo son:
 - "a" para añadir al final de un archivo existente.
 - "r" para leer de un archivo.
 - "w" para escribir en un archivo (si no existe el archivo se creará).
4. La función `fscanf()` permite leer de un archivo datos en modo mixto.
5. El propósito de una función para leer o escribir bloques es almacenar o recuperar un bloque de datos cada vez, por ejemplo, una estructura C.

Sección 9.3

1. Los dos tipos de flujo de datos usados para operaciones de E/S son: de texto y binario.
2. La memoria intermedia es una zona para el almacenamiento temporal de datos.
3. Los tres archivos estándar usados por un programa son: la entrada estándar, la salida estándar y el error estándar.
4. Los argumentos de la línea de mandatos permiten pasar información al programa que se ejecuta.

Sección 9.5

1. Si un archivo se abre correctamente, el puntero a archivo devuelto por `fopen()` será distinto de nulo.
2. El mensaje `mas...` se borra imprimiendo caracteres de retroceso y seguidamente escribiendo espacios en las mismas posiciones.
3. La comprobación (`argc > 1`) se necesita para asegurar que se especifica un nombre de archivo en la línea de mandatos.

4. Cuando se suma 13 al valor de la letra, las letras de la N a la Z tendrán valores mayor que 26 y será necesario restarles 26 para dejarlas en el intervalo adecuado.

Soluciones de la autoevaluación — Capítulo 9

1. La estructura tiene cuatro miembros: `pieza`, `cantidad`, `precio` y `num_registro`.
2. La función `leer_menu()` es de tipo `char`.
3. No se usan variables globales en el programa.
4. El propósito de los argumentos en la función `main()` es obtener la información de la línea de mandatos donde aparecerá la autorización para escribir datos en el archivo.
5. El caso correspondiente a la letra E está situado justo antes de la opción `default` para que si no hay autorización, se muestre el mensaje asociado a dicha opción.
6. No se usa la función `scanf()` en el programa.
7. El programa responderá de la misma forma con independencia de que el usuario use mayúsculas o minúsculas gracias al uso de la función `toupper()`.
8. El campo `precio` debe ser de tipo `long` puesto que la función `atoi()` devuelve un valor de este tipo.
9. La información del archivo está en modo binario, por ello en la función `fopen()` se especifica `"b"`.

Soluciones de los problemas impares del final del capítulo — Capítulo 9

Sección 9.1

1. Almacenar los datos en un archivo permite que éstos se mantengan después de que termine el programa que los generó.
3. Sí, especifica un archivo almacenado en el dispositivo B:.
5. Todos los archivos abiertos se deben cerrar antes de que termine el programa usando la función `fclose()`.

Sección 9.2

7. Los tres modos de apertura básicos de un archivo son:
 - `"a"` para añadir al final de un archivo existente.
 - `"r"` para leer de un archivo.
 - `"w"` para escribir en un archivo (si no existe el archivo se creará).

Sección 9.3

9. Los dos tipos de flujos de datos usados para operaciones de E/S son: de texto y binario.
11. La correspondencia física de los tres archivos estándar es la siguiente:

entrada estándar:	teclado
salida estándar:	pantalla
error estándar:	pantalla

Respuestas de las revisiones de las secciones — Capítulo 10

Sección 10.1

1. Un puntero a función es una declaración de variable que permite acceder a una función a través de la dirección que expresa su nombre.

2. Son importantes los paréntesis que rodean al puntero a función declarado porque si no se ponen puede que la declaración no sea de puntero a función, sino de una función que devuelve un puntero. Esa es la diferencia entre `void (*funcion) (void)` y `void *funcion (void)`.
3. Se puede usar un puntero a función como miembro de una estructura, de un vector o de cualquier otro tipo compuesto de C.
4. `int (*punt) (int numero); punt = sumar;`

Sección 10.2

1. Pasar una función como argumento sirve para poder programar funciones cuyo resultado no se conoce hasta el momento de ejecución. Además, se puede usar para programar funciones que ejecutan otras funciones que reciben como argumento, aunque no conozcan de que tipo de función se trata.
2. Son importantes los paréntesis que rodean al argumento a función declarado porque si no estuvieran se entendería como una declaración de puntero y no de una función.
3. Se puede usar cualquier función como argumento de otra función. No hay ninguna restricción excepto la compatibilidad de los prototipos de ambas.
4. `void func (int contador; int (*funcarg) (int num));`

Sección 10.3

1. Una función con un número variable de argumentos sirve para implementar cálculos o presentaciones sobre un número de parámetros que no se conoce hasta el momento de la ejecución. Ese es el caso de la función `printf`.
2. El primer parámetro de una función con número variable de argumentos es importante porque indica cuántos argumentos vienen después. Esta indicación puede ser explícita, un número, o estar implícita en el formato, los especificadores de tipo de `printf`.
3. Sí, se puede usar cualquier argumento dentro de la parte variable del prototipo.
4. Sí, se pueden usar distintos tipos de argumentos dentro de la parte variable del prototipo.

Sección 10.4

1. Significación de los siguientes términos:
 - a) abstracción: decisión consciente de ignorar los detalles irrelevantes y concentrarse solamente en los detalles relevantes.
 - b) unidad de programa: partes de un programa que dan cuerpo a una abstracción o tipo abstracto de datos.
 - c) exportable: dícese del objeto que puede ser compartido por objetos externos al que lo declaró.
 - d) reglas de ámbito: reglas que controlan la visibilidad de los objetos declarados en un programa escrito en C.
2. La definición de una variable reserva memoria para su localización y la inicia con un valor. La referencia a una variable sólo indica una intención de uso, pero no tiene efectos de asignación de memoria ni de iniciación.
3. Si dos variables globales tienen el mismo nombre se produciría un error de compilación y puede que, posteriormente, otro de elzado de la aplicación.
4. Los archivos de las bibliotecas de C se incluyen poniendo su nombre ente `<` y `>`. Los de usuario se incluyen poniendo su nombre entre `"` y `"`.

Índice

`&else`, 153
`&endif`, 153
`&ifdef`, 153
`&include`, 6
`*argv[]`, 469

A

Adición binaria, 223
Almacenamiento por filas, 344
Ámbito, 502
Ámbito de una variable, 242
Analizador sintáctico, 298
AND
 binario `&`, 125, 126
 lógico `&&`, 130
ANSI C, 46
Aplicación modular, 497
Árbol binario, 417
Archivo binario, 457
Archivo de acceso aleatorio, 464
Archivo de acceso secuencial, 464
Archivo de cabecera, 83, 503
Archivo de texto, 457
Archivos de biblioteca, 3
Archivos de cabecera de biblioteca, 504
Archivos de cabecera de usuario, 504
Archivos estándar, 464
Archivos para incluir, 3
`argc`, 469
Argumento, 15
Argumentos en la línea de órdenes, 468
Argumentos múltiples, 69

ASCII, 273
`atof()`, 274
`atoi()`, 274
`atol()`, 274
`auto`, 247
Autodepuración, 196

B

Bifurcación abierta, 108
Bifurcación cerrada, 116
Bloque de programa, 53
Bloque de selección, 56
Bloque del programador, 10
Bloque repetitivo o de bucles, 56
Bloque secuencial, 56
Bucle centinela, 183
Bucle condicional, 183
Bucle `do while`, 183
Bucle `for`, 172
Bucle `while`, 179
Bucles anidados, 188
Búsqueda binaria, 429
Byte (octeto), 221

C

Cadena de caracteres, 14, 260
Campo de enlace, 400
Campo, 17
Campos de datos, 400
Carácter nulo 0, 260

Carácter, 14
 Círculo de lectura/ejecución, 217
 Clases de declaración, 501
 Código fuente, 6
 Cola, 414
 Colisión, 359, 411
 Comentarios /* */, 6
 Compilación condicional, 152
 Compilador, 3, 334
 Complemento a dos, 224
 Complemento a uno, 224
 Componente léxico, 13, 298
 Concatenación de componentes léxicos, 88
 const, 22
 Constantes, 80, 246
 Convenio de las llamadas en C, 501
 Conversión de tipos, 137, 341
 ctype.h, 271
 Cuarteto (*nibble*), 224

D

Declaración de un puntero, 229
 Declaración, 19
 Define, 79
 Desplazamiento de bits, 129
 Dimensión, 310
 Dirección, 216
 Direcccionamiento directo, 219
 Direcccionamiento inmediato, 219
 Diseño descendente (*top-down*), 11
 DOS, 444

E

E/S de sistema, 464
 E/S estándar, 464
 Editor, 3
 Elementos, 261
 Enlace externo, 505
 Enlace interno, 505
 Enlazador, 3
 Entero, 14
 Entorno, 2
 EOF, 447
 Error en tiempo de ejecución, 194
 Especificador de formato de la función `scanf()`
 %c %d %e %f %o %u %x, 32

Especificador de formato, 16
 Especificador de tipo, 22
 Especificadores de ancho de campo, 30, 345
 Estructura de bloques, 55
 Estructura de un compilador, 299
 Estructura de un programa, 7
 Etiqueta de una estructura, 382
 exit(), 63
 extern, 247
 extern, 502

F

FALSO, 105
 fclose(), 445
 FIFO, 414
 Flujo de caracteres, 447
 Flujo de datos en memoria intermedia (*buffer*), 463
 Flujo de datos, 462
 fopen(), 445
 Formato de archivo, 449
 fprintf(), 455
 fread(), 461
 free(), 406
 fscanf(), 456
 fseek(), 566
 Función de dispersión, 356
 Función, 19, 57
 Funciones como argumentos, 489
 fwrite(), 458

G

getc(), 447
 getchar(), 181, 270
 gets(), 266

H

Hijo derecho, 417
 Hijo izquierdo, 417
 Histograma, 354

I

Identificador, 19
 Índice de un vector, 318

Iniciación de un vector, 313
 Iniciar de una cadena de caracteres, 265
 isalnum(), 271
 isalpha(), 271
 iscntrl(), 271
 isdigit(), 271
 isgraph(), 271
 islower(), 271
 isprint(), 271
 ispunct(), 271
 isspace(), 271
 isupper(), 271
 isxdigit(), 271

L

LIFO, 411
 Lista de declaración de miembros, 383
 Lista de parámetros formales, 62, 65
 Lista enlazada, 401

M

Macro, 80
 main(), 19
 malloc(), 402
 Manipulación de bits, 124
 math.h, 45
 Matrices de estructuras, 393
 Matrices numéricas, 343
 Matriz cuadrada, 349
 Matriz de dos dimensiones, 284
 Matriz multidimensional, 343
 Matriz rectangular, 284
 Matriz, 343
 Memoria del computador. Ver memoria
 Memoria, 11
 Mensajes de error, 34
 Mezcla de tipos de datos, 137, 341
 Miembros de una estructura, 376
 Modo binario, 457
 Modo texto, 457
 Montaje de mi programa, 508
 Multiplicación de matrices, 349

N

Nodo raíz, 417
 Nodo, 400

Nombre de un archivo, 444
 Notación científica, 33
 NULL, 401
 Número variable de argumentos, 493

O

Operaciones Booleanas, 125
 Operaciones de comparación, 104, 133
 Operaciones lógicas
 AND &&, 130
 NOT !, 133
 OR ||, 132
 Operaciones sobre archivos, 450
 Operaciones sobre cadena de caracteres, 275
 Operador binario AND &, 125, 126
 Operador binario de complemento a uno F0011x, 125
 Operador binario OR |, 125, 127
 Operador binario XOR ^, 125, 128
 Operador coma, 178
 Operador condicional ?, 149
 Operador de asignación =, 23, 107
 Operador de cadena &, 88
 Operador de cadena de caracteres [], 260
 Operador de decremento --, 176
 Operador de desplazamiento a la derecha, 129
 Operador de desplazamiento a la izquierda, 129
 Operador de dirección &, 31, 229
 Operador de igualdad ==, 107
 Operador de incremento ++, 176
 Operador de indirección, 229
 Operador de miembro ., 378
 Operador de puntero en una estructura -, 386
 Operador modulo %, 25
 Operador puntero *, 229
 Operador XOR ^, 125, 128
 Operadores aritméticos + - * / %, 25
 Operadores aritméticos + - * / %, 25
 Operadores de asignación compuestos += -= *= /=
 %=, 27
 OR
 binario |, 125, 126
 Lógico ||, 132
 Orden lexicográfico, 285
 Ordenación de cadena de caracteres, 284
 Ordenación por el método de la burbuja, 285, 323
 Ordenación por el método de la cubeta (*bucket*), 329
 Ordenación por fusión, 331

Ordenación rápida (*quick sort*), 336
 Organización encadenada de archivos, 352

P

Palabra, 221
 Palabras reservadas, 13
 Palíndromo, 297
 Parámetro real, 65
 Parámetros de la función `fopen()`, 450
 Paso de valores, 65
 Paso de vectores numéricos, 314
 Pila en tiempo de ejecución, 198
 Pila, 411
 Pivote, 337
 Plantilla, 382
 Plataforma, 46
 Precedencia de operadores, 26
`printf()`, 15
 Prioridad de operaciones, 26
 Programación estructurada, 1
 Programas de ejemplo
 `abin()`, 252
 `adivinar()`, 210
 `ahex()`, 160
 Analizador léxico, 300
 `bfs()`, 433
 `binarysearch()`, 430
 `cachet()`, 162
 Calculadora, 207
 Codificación por transposición, 302
 Comprobador de ISBN, 295
 Comprobador de palíndromos, 297
 Contador de nodos, 425
 Contador de vocales, 296
 Cuadrado mágico, 399
 `dfs()`, 433
 `dispersion()`, 357
 `enlaces()`, 353
 Enumerador de cadena de caracteres, 209
 `euclid()`, 161
 Formateando un texto, 288
 Generador de histogramas, 355
 Generador de la serie de Fibonacci, 206
 Generador de números pseudo-aleatorios, 252
 Inversión de cadena de caracteres, 426
 Máquina vendedora, 203
 MiniMicro, 423
 `RAMer()`, 162
 `rand()`, 252, 355

Prototipo de una función, 58
 Puerto de impresora paralelo, 523
 Puntero a función, 485
 Puntero de un archivo, 445, 466
 Puntero, 228, 262
`putc()`, 446
`puts()`, 40

Q

`qsort()`, 340

R

`rand()`, 355
 Rango, 137
 Reales, 14
 Recorrido de un árbol, 417
 Recorrido en anchura, 432
 Recorrido en orden, 418
 Recorrido en profundidad, 432
 Recorrido postorden, 418
 Recorrido preorden, 418
 Recursividad, 198, 336, 337, 430
 Referencia externa, 502
`return()`, 67

S

Salida estándar, 15
`scanf()`, 31
 Secuencia de escape, 29
 Sensibilidad a mayúsculas y minúsculas, 20, 35
 Sentencia `break`, 141
 Sentencia `case`, 141
 Sentencia compuesta `if...else`, 117
 Sentencia compuesta, 15, 111, 174
 Sentencia condicional, 109
 Sentencia `default`, 141
 Sentencia `if`, 109
 Sentencia `if...else`, 117
 Sentencia, 15
 Separador de bloques, 55
`sizeof()`, 222
`stdio.h`, 340
`stdlib.h`, 340
`strcat()`, 275, 276
`strcmp()`, 275, 279
`strchr()`, 275, 280

string.h, 275
 strlen(), 275
 strncat(), 275, 276
 strncmp(), 275, 279
 strpbrk(), 275, 282
 strrchr(), 275, 280
 strstr(), 275, 281
 strtod(), 275, 283
 strtol(), 275, 283
 strtoul(), 275, 283
 struct, 376
 Suplente de un programa, 156
 Sustracción binaria, 224
 switch(), 139

T

Tabla de asignación de archivos (FAT), 352
 Tabla de dispersión, 356
 Técnicas de ordenación
 Método de la burbuja, 285, 323
 Ordenación por el método de la cubeta (*bucket*), 329
 Ordenación por fusión, 331
 Ordenación rápida (*quick sort*), 336
 qsort(), 340
 Tiempo de vida, 243
 Tipos de datos
 char, 14, 22
 const, 22
 double, 14, 22
 enum, 14, 22, 366, 421
 float, 14, 22
 int, 14, 22
 long, 14, 22
 long double, 14, 22

rango de valores, 14, 227
 short, 14, 22
 unsigned char, 14, 22
 unsigned int, 14, 22
 unsigned long, 14, 22
 unsigned short, 14, 22
 void, 22
 volatile, 22
 Tipos enumerados, 366
 Transportable, 46
 Traza de un programa, 194
 typedef, 371

U

union, 389
 Utilidad make, 506

V

va-arg, 493
 va-end, 493
 va-list, 492
 va-start, 493
 Valor-i (*Ivalue*), 138
 Variable automática, 246
 Variable de registro, 248
 Variable estática, 247
 Variable externa, 247
 Variable global, 243
 Variable local, 242
 Variable, 22, 246
 Vector irregular, 287
 Vectores numéricos, 310
 VERDADERO, 105
 void, 63

James L. Antonakos • Kenneth C. Mansfield Jr.

PROGRAMACIÓN ESTRUCTURADA EN C

El objetivo de este libro es enseñar el lenguaje de programación C a personas *sin conocimientos previos de programación*. Cabe destacar el hecho de que el aprendizaje de esta programación se hace en forma *estructurada*, con el fin de que el lector aprenda a realizar los programas de forma más eficiente y éstos sean más fáciles de leer.

Se estudian todas las herramientas y técnicas básicas de la programación en C como son: bucles, vectores y matrices, cálculos, estructuras de datos, entrada/salida, etc., e incluye un último capítulo dedicado a temas avanzados donde se tratan tópicos como punteros a funciones, funciones con parámetros, programas con módulos y el uso de la herramienta MAKE.

Se presentan de forma completa todos los aspectos de la programación en lenguaje C con múltiples figuras y esquemas, lo que permite ofrecer un tratamiento gráfico de todos los temas estudiados. El libro contiene también más de 200 programas en lenguaje C, todos ellos comprobados y compilados.

El disquete adjunto contiene los archivos con el código fuente en lenguaje C de todos los programas presentados en el libro, cada uno compilado para poder ejecutarse tanto en un entorno MS/DOS de PC como en estaciones de trabajo en entorno UNIX.



PRENTICE HALL

ISBN 84-89660-23-9



9 788489 660236