24

ENGINYERIA
INFORMÀTICA

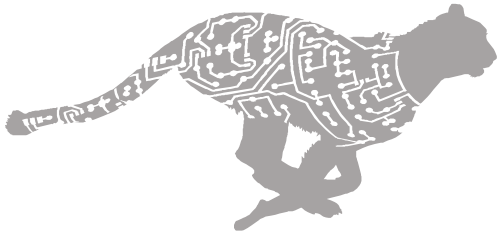
UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

UPCGRAU

Fundamentos de ordenadores: programación en C

Marta Jiménez Castells
Beatriz Otero Calviño

24
ENGINYERIA
INFORMÀTICA



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

iniciativa
digital politècnica
Publicacions Acadèmiques UPC

UPCGRAU

Fundamentos de ordenadores: programación en C

Marta Jiménez Castells
Beatriz Otero Calviño

Primera edición: diciembre de 2013

Diseño y dibujo de la cubierta: Jordi Soldevila
Diseño maqueta interior: Jordi Soldevila

© Iniciativa Digital Politècnica, 2013
Oficina de Publicacions Acadèmiques Digitals de la UPC
Jordi Girona 31,
Edifici Torre Girona, Planta 1, 08034 Barcelona
Tel.: 934 015 885
www.upc.edu/idp
E-mail: info.idp@upc.edu

DL: B-24302-2013
ISBN: 978-84-7653-996-5

Introducción

El libro *Fundamentos de ordenadores: programación en C* trata de una forma sencilla los aspectos más básicos de la programación imperativa procedural, utilizando el lenguaje C como lenguaje de programación. El libro no pretende ser un manual sino un libro que inicie al lector en la programación. Por ello, no se explica con detalle toda la potencia del lenguaje y solamente se utiliza un subconjunto básico del mismo en los programas desarrollados. Después de la lectura de este libro, el lector será capaz de diseñar programas en C de dificultad media-baja.

Este libro se creó inicialmente con la intención de ser un libro de soporte para la asignatura de *Fundamentos de Ordenadores* que se imparte en las titulaciones de la *Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona* (ETSETB) de la *Universitat Politècnica de Catalunya* (UPC). Sin embargo, pensamos que el libro también puede ser de gran ayuda a estudiantes de universidad, de bachillerato o de formación profesional que se inician en la programación y no tienen conocimientos previos en esta materia.

El libro está estructurado como un curso completo y por tanto debe ser leído en orden secuencial. Está compuesto por nueve capítulos. El primero es una breve introducción al mundo de los computadores y a su programación. El resto de los capítulos introducen los conceptos más básicos de la programación imperativa procedural: variables, constantes y sentencias básicas de asignación y llamada a función (capítulo 2), tipos de datos elementales (capítulo 3), sentencias condicionales (capítulo 4), sentencias iterativas (capítulo 5), estructuras (capítulo 6), vectores (capítulo 7) y funciones (capítulos 8 y 9).

Hemos querido darle al libro un enfoque principalmente práctico. Todos los capítulos incluyen, después del necesario estudio teórico del concepto básico que se introduce, varios ejemplos prácticos y una colección de problemas para

que el lector pueda mejorar su aprendizaje. Además, incluimos las resoluciones a los problemas para facilitar la autocorrección.

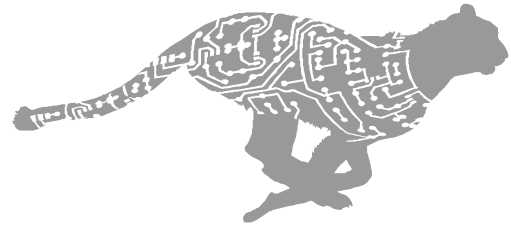
También prestamos especial atención al estilo de programación. Es muy importante adoptar un buen estilo desde el principio para que los programas sean fáciles de leer. Por ello, definimos unas normas de estilo muy simples que seguimos en todos los ejemplos y recomendamos al lector aplicarlas cuando diseñe sus programas.

Por otra parte, nuestra experiencia durante más de doce años como profesoras en distintas asignaturas de introducción a la programación en la ETSETB nos ha permitido también recopilar los errores más frecuentes que realizan los programadores noveles. En algunos capítulos describimos y mostramos con diferentes ejemplos estos errores habituales para ayudar al lector a descubrirlos en sus propios diseños.

Finalmente, queremos agradecer a todos los profesores que han impartido docencia en la asignatura de *Fundamentos de Ordenadores* por sus valiosas aportaciones en numerosas discusiones sobre cómo debemos iniciar a los estudiantes en la programación.

Barcelona, octubre de 2013

Las autoras



Índice

Conceptos básicos de programación

Este capítulo es una breve introducción al mundo de los computadores y a su programación. Comenzaremos dando al lector unas nociones de la organización de un computador y, a continuación, definiremos algunos términos básicos relacionados con la programación. Finalizaremos el capítulo explicando las distintas etapas de la elaboración de un programa, y su proceso de codificación y prueba.

1.1 Organización de un computador

Un computador es un dispositivo electrónico que permite transmitir, almacenar y manipular información mediante la ejecución de programas.

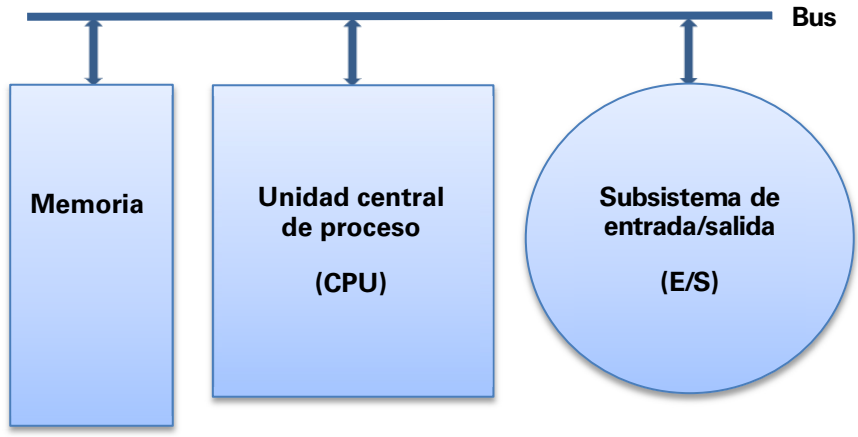
La mayoría de los computadores siguen el modelo basado en la estructura Von Neumann. John von Neumann (1903-1957)[1] fue un prestigioso matemático estadounidense de origen húngaro que colaboró en el proyecto ENIAC para la construcción de la primera computadora de propósito general. Von Neumann realizó importantes aportaciones en el área de la organización y la estructura de los computadores.

Los computadores con una estructura Von Neumann constan de tres elementos básicos: la unidad central de proceso, la memoria y el subsistema de en-

[1] ASPRAY, W. (1993): *John von Neumann y los orígenes de la computación moderna*. Gedisa.

trada/salida. Estos tres elementos básicos están interconectados mediante un bus (v. Fig. 1.1).

Fig 1.1
Estructura básica de un computador

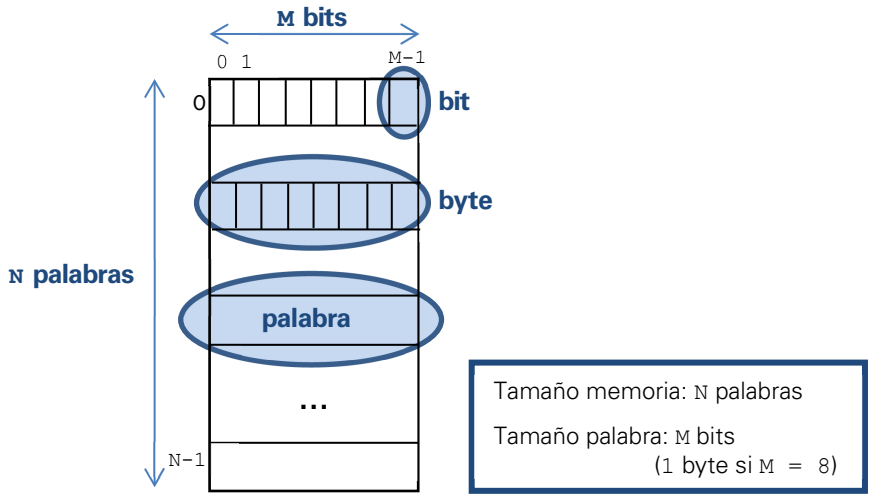


La memoria

La memoria es el dispositivo que almacena la información, es decir los programas y los datos que manipulan los programas.

La memoria está organizada como una matriz de NxM bits (v. Fig. 1.2). Un *bit* (en inglés, ***bi**nary digit*) es la unidad de información más pequeña y solamente puede tomar un valor: cero o uno. El conjunto de 8 bits se denomina *byte* y el conjunto de M bits que conforman una fila de la matriz de memoria, *palabra*. Generalmente, M suele ser igual a 8 bits, es decir, la palabra suele ser del tamaño de un byte. En el anexo 1.A se pueden consultar otras unidades de información y de medida de la memoria.

Fig. 1.2
Organización de la memoria de un computador



Los datos se almacenan en la memoria utilizando la representación binaria correspondiente al tipo de dato que representa (entero, natural, carácter, etc.) y suelen ocupar uno o varios bytes. Por otro lado, los programas se almacenan como una secuencia de instrucciones básicas que el computador entiende directamente (sumar, restar, multiplicar, comparar valores, etc.). Cada una de estas instrucciones básicas también ocupa uno o varios bytes.

A continuación se muestran algunos ejemplos de cómo se almacenan en la memoria diferentes tipos de datos e instrucciones básicas:

Tabla 1.1
Ejemplos de datos e instrucciones almacenados en la memoria

	Tipo	Valor	Representación binaria
Dato1	entero	`-3`	`11111111111111111111111111111101`
Dato2	carácter	`'C'`	`01000011`
Dato3	natural	`25`	`0000000000011001`
Instrucción1	sumar	`add %edx,%eax`	`0000000111010000`
Instrucción2	retorno	`ret`	`11000011`

La unidad central de proceso

La unidad central de proceso (CPU) es el dispositivo que procesa la información mediante la ejecución de programas, es decir, ejecuta las instrucciones de los programas en el orden indicado.

La unidad central de proceso solo es capaz de ejecutar órdenes y operaciones básicas como: sumar, restar, multiplicar, comparar valores, almacenar/recuperar un dato de la memoria, etc.

Subsistema de entrada/salida

El subsistema de entrada/salida (E/S) es el conjunto de dispositivos que conecta el computador con el mundo exterior y permite almacenar la información de manera permanente. Hay tres tipos de dispositivos: dispositivos de entrada (teclado, ratón, lector de tarjetas, etc.), dispositivos de salida (pantalla, impresora, etc.) y dispositivos de almacenamiento permanente (discos duros, DVD, CD, etc.).

El bus

El bus es el enlace entre los tres subsistemas y permite transmitir la información entre ellos.

1.2 Términos básicos en programación

El lector ha de tener muy claro que el computador no resuelve problemas por sí solo, sino que únicamente es capaz de realizar, con una gran potencia de cálculo, las operaciones necesarias para obtener la solución. Es el programador, mediante los programas, quien indica al computador los cálculos que ha de realizar para obtener la solución del problema. Por tanto, el programador debe realizar los pasos siguientes ante el planteamiento de un problema:

- Buscar o pensar un método para resolverlo, es decir, determinar las operaciones necesarias y el orden en que estas han de ser ejecutadas (algoritmo).
- Escribir el algoritmo en un lenguaje de programación determinado (programa).
- Probar, depurar y finalmente ejecutar el programa en el computador para obtener la solución al problema (proceso de codificación y prueba).

A continuación, definimos algunos términos básicos relacionados con la programación, explicamos las etapas en la elaboración de un programa y, finalmente, mostramos el proceso de codificación y prueba de este.

Algoritmo

Es un procedimiento de cálculo que ejecuta, de forma ordenada, una serie de instrucciones y conduce a la solución de un problema en un tiempo finito. Las características de un algoritmo son:

- Es **preciso**. Un algoritmo indica exactamente el orden en que deben realizarse las operaciones.
- Ha de estar **bien definido** (**no** ha de ser **ambiguo**). Si se sigue el algoritmo dos veces con los mismos datos de entrada, se obtiene el mismo resultado.
- Es **finito**. El algoritmo tiene que terminar en algún momento.

Ejemplo de algoritmo:

La figura1.3 muestra un algoritmo para dibujar un triángulo equilátero cuya longitud de lado es de `100` puntos. La figura 1.3.a muestra una versión extendida del algoritmo, mientras que la figura 1.3.b muestra el mismo algoritmo, escrito de una manera más compacta.

```
① Inicializar la posición del lápiz
② Inicializar la dirección del lápiz
③ Trazar 100 puntos
④ Girar 120°
⑤ Trazar 100 puntos
⑥ Girar 120°
⑦ Trazar 100 puntos
⑧ Girar 120°
```

(a) Versión extendida

```
① Inicializar la posición del lápiz
② Inicializar la dirección del lápiz

Repetir
  ③ Trazar 100 puntos
  ④ Girar 120°
hasta 3 veces
```

(b) Versión compacta

Fig. 1.3
Algoritmo para dibujar un triángulo equilátero

En la figura 1.4, se hace un seguimiento paso a paso del algoritmo de la figura 1.3.a y se muestra cómo se llega a la solución del problema.

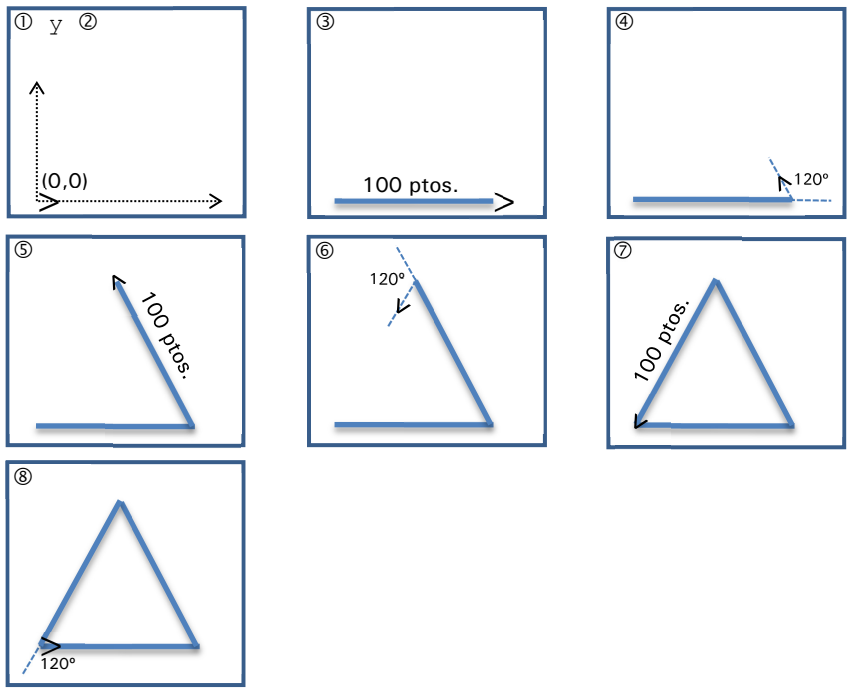


Fig. 1.4
Ejecución paso a paso del algoritmo de la figura 1.3

Programa

Un programa es la codificación de un algoritmo en un lenguaje de programación determinado.

Así, dado el enunciado de un problema, primero hay que pensar el algoritmo que lo resuelve (fase de diseño) y, a continuación, codificar el algoritmo en un lenguaje de programación determinado (fase de codificación).

La fase de diseño es un proceso creativo en que cuentan tanto la inteligencia como la intuición y la experiencia del programador. No existe un método fijo para encontrar el algoritmo que resuelve un problema determinado.

Por el contrario, la fase de codificación es mecánica y automática. Simplemente, es necesario conocer las reglas sintácticas del lenguaje de programación que se esté usando y codificar el algoritmo en ese lenguaje.

Los criterios más importantes para determinar un buen programa son:

- **La corrección.** Los programas han de funcionar correctamente y resolver el problema planteado.
- **La claridad.** Los programas han de ser fáciles de leer, entender y modificar.
- **La eficiencia.** Los programas han de llegar a la solución en el menor tiempo posible.

Lenguajes de programación

Un lenguaje de programación es un conjunto de símbolos y caracteres que se combinan entre sí siguiendo una serie de reglas sintácticas predefinidas por el lenguaje.

Existen diversas clasificaciones de los lenguajes de programación: según su evolución histórica, el propósito, el paradigma de la programación, el nivel de abstracción, etc. Nosotros nos centramos en la clasificación según el nivel de abstracción.

- **Lenguajes de alto nivel.** Son los lenguajes de programación que, por sus características, se parecen más al lenguaje humano. La característica principal de estos lenguajes es que son portables o independientes de la arquitectura. Esto quiere decir que un programa escrito en un lenguaje de alto nivel se puede ejecutar en distintos computadores sin necesidad de modificarlo.

  Los lenguajes de alto nivel, a su vez, se pueden clasificar en función del estilo de programación empleado (paradigma de programación), y así nos encontramos, entre otros, con los tipos siguientes:

- *Lenguajes de programación procedurales*. La programación consiste en dividir el problema inicial en partes más pequeñas. Los subproblemas son resueltos por subprogramas (subrutinas, funciones, procedimientos), que se llaman entre sí para llegar a las solución completa del problema. Los lenguajes C y Pascal, por ejemplo, son lenguajes de programación procedurales.
- *Lenguajes de programación orientados a objetos*. La programación consiste en crear clases y objetos, y especificar la interacción entre ellos. Los lenguajes C++ y Java son lenguajes de programación orientados a objetos.

– **Lenguajes de bajo nivel.** Son lenguajes específicos de la arquitectura, es decir, los programas escritos en lenguajes de bajo nivel no son portables, por lo que solo se pueden ejecutar en el computador para el cual fueron diseñados. Se incluyen en esta categoría el lenguaje máquina y el lenguaje ensamblador. El lenguaje máquina se caracteriza por ser el único que el computador puede interpretar directamente y se basa en la combinación de solo dos símbolos (el 0 y el 1). El lenguaje ensamblador es una representación simbólica del lenguaje máquina para facilitar la programación de bajo nivel.

En la figura 1.5, se muestra un programa que calcula el área de un cuadrado de 9 cm de lado, escrito en tres lenguajes de programación diferentes. Obsérvese cómo los programas escritos en lenguajes de bajo nivel (lenguaje ensamblador y lenguaje máquina) son más difíciles de entender que los escritos utilizando lenguajes de alto nivel (lenguaje C).

En los capítulos siguientes del libro, explicaremos cómo programar bajo el paradigma de la programación procedural utilizando el lenguaje C como lenguaje de programación.

Lenguaje C

```
main()
{
  int area,lado=9;
  area=lado*lado;
}
```

Lenguaje ensamblador

```
.bss
.comm area, 4, 4
.data
lado: .long 9
.text
.global main
main:
    movl lado, %eax
    imull %eax, %eax
    movl %eax, area
    movl $0, %ebx
    movl $1, %eax
    int $0x80
```

Lenguaje máquina

```
10100001 00100000
10010101 00000100
00001000 00001111
10101111 11000000
10100011 00101000
10010101 00000100
00001000 10111011
00000000 00000000
00000000 00000000
10111000 00000001
00000000 00000000
00000000 11001101
10000000
```

Fig. 1.5
Ejemplos de programas escritos en diferentes lenguajes de programación

1.3 Etapas en la elaboración de un programa

La metodología que vamos a seguir para la elaboración de un programa completo consta de los pasos siguientes:

Análisis

El primer paso consiste en definir bien el problema, dejando muy claro cuál es la entrada y cuál ha de ser la salida del programa. En esta fase, también se especifican, a muy alto nivel, las estructuras de datos que se utilizarán y las líneas generales para resolver el problema.

Diseño

El paso siguiente consiste en desarrollar los algoritmos para resolver el problema planteado. En un modelo de programación procedural, generalmente se utiliza la técnica de diseño descendente. Esta técnica consiste en resolver el problema mediante la resolución de problemas más sencillos. El problema inicial se descompone en subproblemas autocontenidos y de dificultad menor. Estos subproblemas, a su vez, también se dividen en subproblemas todavía más simples, hasta llegar a subproblemas triviales y fáciles de codificar.

Un ejemplo de diseño descendente ante el problema de resolver ecuaciones de segundo grado es la descomposición del problema inicial en los subproblemas siguientes (v. Fig. 1.6):

(1) Leer los coeficientes a, b y c que definen la ecuación.

(2) Calcular el valor del discriminante.

(3) En función del valor del discriminante, calcular sus soluciones. Este subproblema puede dividirse, a su vez, en los subproblemas siguientes:

   (3.1) Si el discriminante es `0`, calcular la solución real doble.

   (3.2) Si el discriminante es mayor estricto que `0`, calcular las dos raíces reales y diferentes.

   (3.3) Si el discriminante es menor estricto que `0`, calcular las dos raíces complejas.

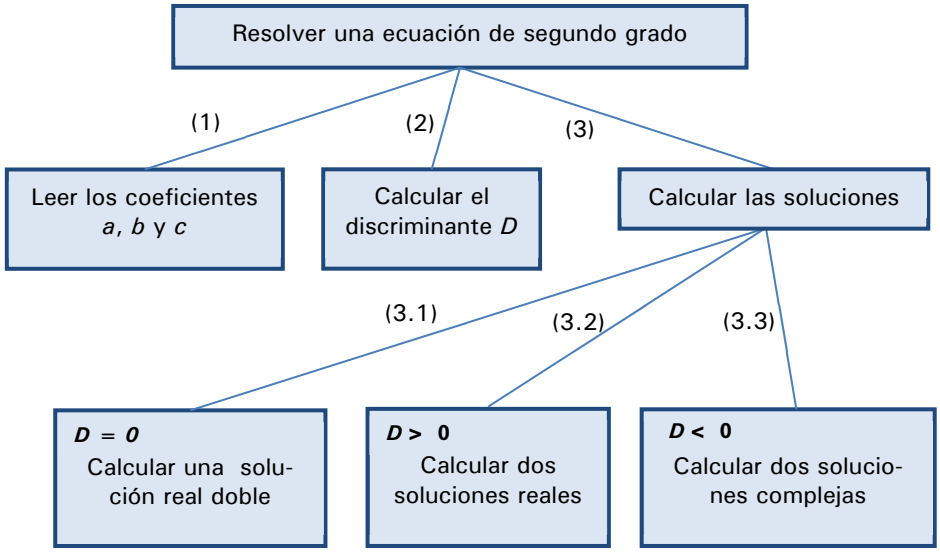


Fig. 1.6
Ejemplo de diseño descendente

Codificación

La fase de codificación consiste en escribir, en el lenguaje de programación elegido, cada uno de los algoritmos diseñados en la fase anterior. Por regla general, cada uno de los algoritmos ha de ser codificado y probado independientemente.

Prueba

Este paso consiste en ejecutar los programas en el entorno definido. El programa ha de probarse con diferentes datos de entrada, sin olvidarse de los casos especiales o menos frecuentes.

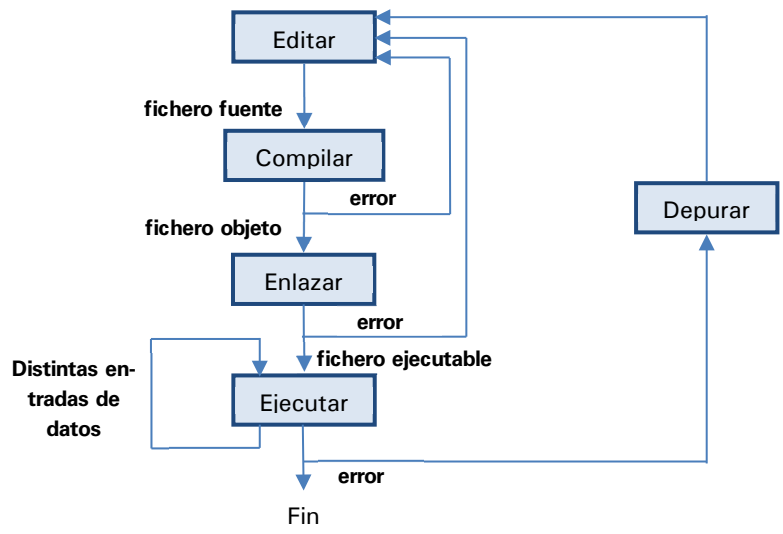
Mantenimiento

El lector ha de tener claro que en la fase anterior solo se detecta la presencia de errores, pero nunca se puede confirmar la ausencia total de estos. Es por ello por lo que la última etapa en la elaboración de un programa siempre es el mantenimiento. En esta etapa, se corrigen los errores hallados posteriormente y se añaden nuevas funcionalidades al programa.

1.4 Proceso de codificación y prueba de un programa

Independientemente del entorno de programación que se escoja para desarrollar los programas, el proceso de codificación y prueba de cada uno de los algoritmos diseñados en la fase de diseño es el que muestra la figura 1.7.

Fig. 1.7
Proceso de codificación y prueba de un programa



Edición

En primer lugar, hay que codificar el algoritmo en el lenguaje de programación escogido, siguiendo las reglas sintácticas definidas por el lenguaje. Para ello, se puede utilizar cualquier editor de texto. El resultado de esta etapa es un fichero de texto que contiene el código del programa (fichero fuente). La extensión de este fichero indica el lenguaje de programación utilizado (`.c` para códigos en C, `.f` para Fortran, `.cpp` para C++, etc.).

Compilación

El compilador es una herramienta que detecta los errores sintácticos de los programas, e indica el tipo de error y la línea del código donde se encuentra. Así, una vez escrito el código en el fichero fuente, hay que "compilar" el programa, es decir, determinar si hay errores sintácticos en el código y corregirlos. Una vez compilado el programa (ya no hay errores sintácticos), el compilador genera un fichero con un código objeto. Este fichero tiene la extensión `.o` o `.obj` y es necesario para poder generar posteriormente un fichero ejecutable.

Enlace

Cuando un programa consta de varios ficheros fuente o utiliza funciones propias del lenguaje de programación, es necesario que el enlazador combine todos los ficheros objeto correspondientes y genere un único fichero ejecutable. El enlazador es el encargado de verificar que solamente haya un programa principal, que todas las funciones estén definidas, que las llamadas a las fun-

ciones correspondan con sus definiciones, etc. Si todo es correcto, el enlazador genera un fichero ejecutable que normalmente tiene la extensión `.out`, `.exe` o no tiene extensión.

Ejecución

La etapa siguiente consiste en ejecutar el programa para determinar la presencia de errores semánticos o de ejecución, es decir, para determinar si el programa funciona correctamente o no. En esta etapa, se ha de ejecutar el programa varias veces con diferentes entradas de datos y probar todos los posibles casos del programa. Un error muy habitual en los programadores noveles es ejecutar el programa con una sola entrada de datos, la entrada en que pensaban a la hora de diseñar el algoritmo. El programa suele comportarse correctamente para esa entrada de datos, pero es posible que no haga lo mismo para cualquier otra.

Es muy importante probar exhaustivamente los programas con diferentes datos de entrada, sin olvidar las entradas especiales o menos frecuentes.

Depuración

En la etapa de ejecución, se puede detectar si el programa es incorrecto, pero esta etapa no da ninguna pista sobre qué parte del código ha causado el problema. En estos casos, hay que pasar a la etapa de depuración.

El depurador es una herramienta que permite observar el comportamiento de un programa paso a paso y facilitar así la detección de errores de ejecución. Cuando se detecta un error de ejecución, hay que ir a la etapa de edición para corregir el programa y nuevamente volver a compilar, enlazar y ejecutar.

Una vez depurado el programa, es decir, cuando ya no hay errores de ejecución (semánticos) y el programa funciona correctamente, se puede dar por finalizado el proceso de codificación y prueba.

1.5 Ejercicios

1. Escriba un algoritmo para dibujar un cuadrado cuya longitud de lado es de `100` puntos. Deje el lápiz en la posición inicial.
2. Sea `N` el número de lados de un polígono regular y `X` la longitud en puntos de cada uno de sus lados. Escriba un algoritmo que dibuje un polígono regular de `N` lados de longitud `X`. Deje el lápiz en la posición inicial.

3. Escriba un algoritmo para programar la alarma de su teléfono móvil a las `7:00` de la mañana.

4. Aplicando la técnica del diseño descendente, escriba un algoritmo que, dada la ecuación de dos rectas (`f(x) = ax + b` y `g(x) = cx + d`), determine si las rectas son iguales, paralelas o se intersecan en un punto. En este último caso, determine el punto de corte.

   Por ejemplo, si las rectas son: `f(x) = x + 2` y `g(x) = 2x + 1`, las rectas se intersecan en el punto `(x,y) = (1,3)`.

5. Aplicando la técnica del diseño descendente, escriba un algoritmo que, dada una secuencia de `5` números enteros, indique la suma de los dígitos de cada uno de los números. Por ejemplo, si la secuencia de números es: `102, 46, 21, 3000, 8888`, entonces la suma de los dígitos es, respectivamente: `3, 10, 3, 3, 32`.

1.6 Respuesta a los ejercicios propuestos

1.

Versión extendida:

```
(1)  Inicializar la posición del lápiz
(2)  Inicializar la dirección del lápiz
(3)  Trazar 100 puntos
(4)  Girar 90°
(5)  Trazar 100 puntos
(6)  Girar 90°
(7)  Trazar 100 puntos
(8)  Girar 90°
(9)  Trazar 100 puntos
(10) Girar 90°
```

Versión compacta:

```
(1) Inicializar la posición del lápiz
(2) Inicializar la dirección del lápiz
Repetir
   (3)  Trazar 100 puntos
   (4)  Girar 90°
hasta 4 veces
```

2.

(1) Inicializar la posición del lápiz

(2) Inicializar la dirección del lápiz

Repetir

(3) Trazar X puntos

(4) Girar 360°/N

hasta N veces

3.

Algoritmo para programar la alarma en un iPhone con iOS 6.0

(1) Encender el teléfono pulsando el botón de encendido (arriba a la derecha).

(2) Desplazar a la derecha la barra "Slide to unlock".

(3) Si está bloqueado, introducir el número clave en el teclado.

(4) Buscar el icono "Clock" en la pantalla y pulsarlo.

(5) Pulsar la pestaña "+" (arriba a la derecha).

(6) Escoger la hora deseada (07:00) desplazando el dedo hacia arriba y abajo sobre las horas y minutos mostrados. La hora escogida debe quedar en la zona sombreada.

(7) Pulsar la pestaña "Save" (arriba a la derecha).

(8) Pulsar el botón "Home" (abajo en el centro del terminal) para volver a la pantalla de inicio del teléfono.

4.

(1) Pedir los coeficientes a, b, c y d que definen las ecuaciones de las rectas

Si a y c son iguales y b y d también, **entonces**:

(2) Indicar que las rectas son iguales

Si a y c son iguales, pero b y d son distintas, **entonces**:

(3) Indicar que las rectas son paralelas

Si a y c son distintas, **entonces**:

(4) Indicar que las rectas se intersecan

(5) Calcular el punto de corte (x,y) tal que:

x = (d-b)/(a-c)

y = a*x +b

5.

Repetir

(1) Pedir el número siguiente de la secuencia

(2) Calcular la suma de sus dígitos

(3) Mostrar el resultado

hasta 5 veces

Si se aplica la técnica de diseño descendente, el paso (2) del algoritmo anterior se resuelve de la forma siguiente:

(2.1) Inicializar *"acumulador"* a 0

(2.2) Inicializar *"cociente"* al número de la secuencia que se está tratando

Repetir

(2.3) Guardar en *"resto"* el resto de *"cociente"*/10

(2.4) Sumar *"resto"* al *"acumulador"*

(2.5) Actualizar *"cociente"* con (*"cociente"*/10)

hasta que *"cociente"* sea 0

Si se incluye este desarrollo en el algoritmo, entonces el algoritmo completo queda escrito de la manera siguiente:

Repetir

(1) Pedir el número siguiente de la secuencia

(2.1) Inicializar *"acumulador"* a 0

(2.2) Inicializar *"cociente"* al número de la secuencia que se está tratando

Repetir

(2.3) Guardar en *"resto"* el resto de *"cociente"*/10

(2.4) Sumar *"resto"* al *"acumulador"*

(2.5) Actualizar *"cociente"* con (*"cociente"*/10)

hasta que *"cociente"* sea 0

(3) Mostrar el resultado (indicar que la suma de los dígitos del número es *"acumulador"*)

hasta 5 veces

1.7 Anexos

Anexo 1.A. Unidades de información y de medida de la memoria

Unidades de información:

Nombre	Símbolo	Equivalencia
Bit	b	1 bit
Nibble	-	4 bits
Byte	B	2 nibbles = 8 bits
Word	W	2 B = 16 bits
Long word	L	4 B = 32 bits

Unidades de medida de la memoria:

Nombre	Símbolo	Equivalencia
Bit	b	1 bit
Byte	B	8 bits
Kilobyte	KB	1024 B = 2^{10}B
Megabyte	MB	1024 KB = 2^{20}B
Gigabyte	GB	1024 MB = 2^{30}B
Terabyte	TB	1024 GB = 2^{40}B
Petabyte	PB	1024 TB = 2^{50}B
Exabyte	EB	1024 PB = 2^{60}B
Zettabyte	ZB	1024 EB = 2^{70}B
Yottabyte	YB	1024 ZB = 2^{80}B

En las unidades de medida de la memoria, se utilizan habitualmente las acepciones del Sistema Internacional de Medidas (SI): Kilo-, Mega-, Giga-, etc. Sin embargo, etimológicamente es incorrecto utilizar estos prefijos de base decimal para nombrar múltiplos en base binaria. En el SI, *kilo-* corresponde a 10^3 (1.000 unidades) y no a 2^{10} (1.024 unidades), el prefijo *mega-* corresponde a 10^6 (1.000.000 de unidades) y no a 2^{20} (1.048.576 unidades), etc. Para resolver este problema, la Comisión Electrotécnica Internacional (IEC) estableció el estándar de almacenamiento de 1.024 bytes con una nueva nomenclatura, que proviene de la contracción del prefijo establecido por el SI con la palabra *binary*; así 1.024 bytes son 1 Kibibyte (KiB), 1.024 KiB son 1 Mebibyte (MiB), 1.024 MiB son un Gibibyte (GiB), etc. Sin embargo, el uso de este estándar no se ha extendido mucho.

En este libro, se ha preferido mostrar las unidades de medida de la memoria tal como se utilizan habitualmente, en lugar del estándar IEC.

→2

Empezando a programar

Este capítulo define los elementos necesarios para que el lector pueda empezar a implementar programas sencillos utilizando el lenguaje de programación C.

Comenzamos el capítulo explicando cómo declarar e inicializar variables, y describiendo los operadores básicos que pueden utilizarse para construir expresiones y/o realizar operaciones. A continuación se describen las sentencias más básicas de programación y finalmente se muestra un primer programa escrito en lenguaje C.

2.1 Identificador

Un identificador es una cadena de caracteres que identifica un nombre. Algunos ejemplos de identificadores son: `PI`, `dato1`, `dato2`, `funcion_minimo`, etc. En el lenguaje C, el identificador puede estar formado por letras y números, pero no puede empezar por un número y no puede contener espacios. Los espacios pueden sustituirse por guiones (- o _). Las letras pueden ser cualquiera de las 26 del alfabeto latino, tanto en mayúsculas como en minúsculas. Por tanto, no se pueden utilizar letras específicas de los idiomas (como `ñ`, `ß`, `Ç`, etc.) o letras acentuadas o con diéresis (como `è`, `ü`, `â`, etc.). Además, el lenguaje C distingue entre los caracteres en mayúsculas y en minúsculas de un nombre. Así, el identificador `var1` es distinto de `Var1`.

Finalmente, existen determinadas palabras reservadas que solo pueden utilizarse para su propósito establecido, y no como identificadores, como por ejemplo: `while`, `if`, `void`, `int`, etc. El uso específico de estas palabras reservadas se explica a lo largo de este libro. La tabla 2.1 muestra todas las palabras reservadas del lenguaje C.

Tabla 2.1
Lista de palabras reservadas en el lenguaje C

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

2.2 Variable

Una variable es un identificador que guarda un único valor, que puede ser consultado y modificado durante la ejecución del programa. Las variables se declaran al inicio del programa, y antes de que se utilicen en las operaciones. Al declarar las variables, se está reservando espacio en la memoria para almacenar los valores que tomarán dichas variables durante la ejecución del programa.

Para declarar una variable, es necesario especificar:

- **Un nombre.** El nombre de la variable es un identificador que describe la información que almacena.
- **Un tipo de dato**. El tipo de dato de la variable indica el tipo de información que almacena (entero, real, natural, carácter, etc.).
- **Un valor inicial** (opcional). Con el valor inicial, se indica qué valor se quiere que tenga la variable inicialmente. Si no se especifica el valor inicial, la variable tiene un valor indeterminado (el que haya en la memoria cuando se reserva el espacio para la variable).

La sintaxis general para declarar variables es la siguiente:

```
tipo_var nom_var;              /* Declaración de una variable */

tipo_var nom_var1, nom_var2;   /* Declaración de dos variables
                                  del mismo tipo */

tipo_var nom_var=valor_inicial; /* Declaración e inicialización
                                   una variable */
```

El lenguaje C ofrece varios tipos de datos básicos o elementales. Se tratan todos estos tipos de datos detalladamente en el capítulo 3 (Tipos de datos elementales), pero ahora se muestran algunos de ellos para poder empezar a escribir programas sencillos. Los tres tipos de datos que se manipulan en este capítulo son:

- **Enteros.** El tipo de dato entero se identifica con la palabra reservada `int`. Una variable de este tipo puede almacenar valores enteros. Por ejemplo: `3,-4,35,2000,-223`

- **Reales.** El tipo de dato real se identifica con la palabra reservada `float`. Una variable de este tipo puede almacenar valores reales. Por ejemplo: `13.0,-4.213,3.354,-120.1,23.333`
- **Caracteres.** El tipo de dato carácter se identifica con la palabra reservada `char`**.** Una variable de este tipo puede almacenar un único carácter. Por ejemplo: `'a', 'Z', '1', '*', 'q', ';'`

A continuación, se proporcionan algunos ejemplos de declaración de variables.

Ejemplos de declaración de variables:

```
int f, s=0;          /* Declara las variables enteras f y s.
                        Además, la variable s la inicializa en 0 */

float area=0.0, raiz;  /* Declara las variables reales area y
                          raiz. Además, la variable area
                          la inicializa en 0.0 */

char sexo='f';         /* Declara la variable sexo de tipo
                          carácter y la inicializa en 'f' */
```

Obsérvese en el último ejemplo que la constante literal de tipo carácter (`'f'`) se escribe entre comillas simples. En el lenguaje C, las constantes de tipo carácter se escriben siempre entre comillas simples para poder distinguirlas de las variables: `'f'` es una constante literal de tipo carácter, mientras que `f` (sin comillas simples) es una variable cuyo nombre es `f`.

La figura 2.1 muestra cómo se almacena en la memoria cada una de las variables declaradas en el ejemplo anterior. El carácter `?` en la figura indica que el valor es indeterminado. Además, cada variable ocupa en la memoria `1` o más bytes, dependiendo del tipo de dato. Las variables de tipo `int` y `float` ocupan `4` bytes, mientras que las variables de tipo `char` ocupan `1` byte.

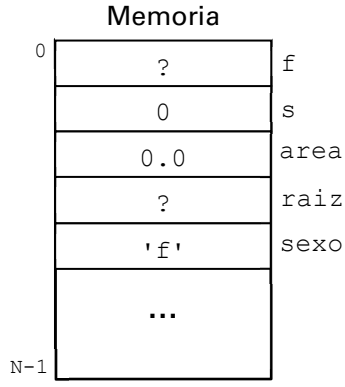


Fig. 2.1
Ejemplo de almacenamiento de variables en memoria

2.3 Constantes

Existen dos tipos de constantes: las constantes literales y las constantes simbólicas.

Constante literal

Una constante literal, también llamada *constante sin nombre*, es un valor de cualquier tipo que se utiliza directamente para operar. No es necesario definirla porque no tiene nombre. Algunos ejemplos de constantes literales de diferente tipo son los siguientes:

- **Entera en decimal**: `3`, `-8`, `210`
- **Entera en binario** (empieza siempre con `0b`): `0b110`, `0b11`, `0B111101`
- **Entera en hexadecimal** (empieza siempre con `0x`): `0x3`, `0x2A`, `0xf2e`
- **Entera en octal** (empieza siempre con `0`): `03`, `021`, `0273`
- **Real**: `34.6`, `-899.023`, `0.0044`
- **Carácter**: `'A'`, `'d'`, `';'`

Constante simbólica

Una constante simbólica es un identificador que representa un valor que puede ser consultado, pero no puede ser modificado durante la ejecución del programa. A diferencia de las variables, cuando se definen las constantes simbólicas no se reserva espacio en la memoria, sino que simplemente se define un identificador para representar un valor constante. Generalmente, el nombre o identificador de una constante describe la información que representa y se escribe en letras mayúsculas para diferenciarlo del nombre de las variables.

Para definir una constante simbólica en el lenguaje C, se utiliza la directiva `#define` y, a continuación, se especifica el nombre de la constante, seguido de su valor.

La sintaxis general para definir una constante es la siguiente:

```
#define NOM_CONS valor_cons
```

La directiva `#define` le indica al compilador que sustituya en el código fuente, todas las ocurrencias del identificador `NOM_CONS` por su valor `valor_cons`. Esta sustitución se lleva a cabo antes de que el compilador realice el análisis sintáctico del código.

Nótese también que después de la definición de una constante no se escribe el carácter punto y coma (;), algo que sí se debe hacer en la declaración de una variable.

Ejemplos de definición de constantes:

```
#define FALSO 0          /* Define la constante simbólica FALSO con
                            el valor entero 0 */

#define PI 3.141516      /* Define la constante simbólica PI con
                            el valor real 3.141516 */

#define HEX_10 0xA       /* Define la constante simbólica HEX_10
                            con el valor entero hexadecimal 0xA */

#define VERDAD 'S'       /* Define la constante simbólica VERDAD
                            con el carácter 'S' */
```

2.4 Expresión

Una expresión es una combinación de operadores y símbolos (constantes y/o variables) que especifica un valor. Algunos ejemplos de expresiones son: `3+4*8`, `2*PI*r`, `4*s+5`.

Ya hemos visto cómo declarar las variables y definir las constantes; ahora veamos los diferentes operadores que ofrece el lenguaje C para poder formar expresiones. Los operadores se pueden clasificar en tres grupos, dependiendo de su función: aritméticos, relacionales y lógicos. A continuación, describimos los operadores de cada grupo que utilizamos en este libro.

Operadores aritméticos

La tabla 2.2 muestra los operadores aritméticos y algunos ejemplos de uso considerando la siguiente declaración de variables:

```
int i=2, j=4;
float r=9.0;
```

Tabla 2.2
Operadores aritméticos

Operación aritmética	Operador	Ejemplo	Resultado
Suma	+	3 + 5	8
Resta	-	j - i r - 1.5	2 7.5
Multiplicación	*	2 * j r * j	8 36.0
División	/	j/2 r/2.0	2 4.5
Módulo	%	j % 3 j % i	1 0

El operador de módulo (%) devuelve el resto de dividir el primer operando entre el segundo. Es el resto de una división entera, por lo que requiere que sus dos operandos sean enteros y que el segundo sea no nulo.

La operación de división (/) también requiere que el segundo operando sea no nulo, pero no es necesario que sea entero. En concreto, la operación de división puede generar resultados reales o enteros: el resultado será entero cuando los dos operandos (dividendo y divisor) sean enteros y el resultado será real cuando alguno de los dos operandos (o ambos) sea real.

Operadores relacionales

Los operadores relacionales se utilizan para comparar dos operandos y formar una condición. Se denomina *condición* una expresión lógica que combina operadores relacionales y/o lógicos y cuyo resultado solo puede ser *falso* (representado con el valor `0` en el lenguaje C) o cierto (representado con el valor `1`).

La tabla 2.3 muestra los operadores relacionales que ofrece el lenguaje C y algunos ejemplos, considerando la siguiente declaración de variables:

```
int i=2, j=4;
```

Tabla 2.3
Operadores relacionales

Tipo de relación	Operador	Ejemplo	Resultado
Menor que	<	`i < j`	`1 (cierto)`
Menor que o igual a	<=	`j <= 2`	`0 (falso)`
Mayor que	>	`i > 2 * j`	`0 (falso)`
Mayor que o igual a	>=	`j >= 2 * i`	`1 (cierto)`
Igual a	==	`3 == 3` `j == i`	`1 (cierto)` `0 (falso)`
Distinto que	!=	`j != 4` `j != i`	`0 (falso)` `1 (cierto)`

Nótese en la tabla anterior que la relación "*igual a*" se representa con el operador relacional == y no con el operador =. Un error habitual en los programadores noveles es confundir el operador de asignación (=) con el operador relacional de igualdad (==).

Operadores lógicos

Los operadores lógicos permiten crear condiciones complejas a partir de condiciones simples. Las operaciones lógicas que se tratan en este libro son la conjunción (también llamada Y-lógica y habitualmente simbolizada con $\wedge$), la disyunción (O-lógica, que se representa con $\vee$) y la negación (que se simboliza con $\neg$). En la tabla 2.4, se recuerda la tabla de verdad de estas operaciones lógicas, donde A y B son dos condiciones simples cuyos resultados son cierto o falso:

Tabla 2.4
Tabla de verdad de las operaciones lógicas

A	B	Conjunción A ∧ B	Disyunción A ∨ B	Negación ¬ A
Cierto	Cierto	Cierto	Cierto	Falso
Cierto	Falso	Falso	Cierto	Falso
Falso	Cierto	Falso	Cierto	Cierto
Falso	Falso	Falso	Falso	Cierto

La tabla de verdad contiene todas las combinaciones posibles de verdad o falsedad de las condiciones simples `A` y `B`, y el resultado lógico de la condición formada al usar los operadores lógicos.

El resultado de la conjunción es `cierto` si *las dos condiciones* (`A` y `B`) conectadas por el operador son `ciertas`; en cualquier otro caso, el resultado es `falso`.

El resultado de la disyunción es `cierto` cuando *alguna de las dos condiciones (pueden ser las dos)* son `ciertas`; en cualquier otro caso (cuando todas las condiciones son `falsas`), el resultado es `falso`.

El resultado de la negación es `cierto` si la condición a la cual se le aplica la operación es `falsa`, y `falso` si la condición es `cierta`.

Una vez repasadas las operaciones lógicas, veamos los operadores que ofrece el lenguaje C para realizarlas. La tabla 2.5 muestra estos operadores y algunos ejemplos de uso, considerando la siguiente declaración de variables:

```
int i=2, j=4;
```

Tabla 2.5
Operadores lógicos

Operación lógica	Operador	Ejemplo	Resultado

Conjunción (∧)	`&&`	`(i > -2) && (i <= 5)`
		`(i > 3) && (i <= 5)`
		`(i >= 1) && (i != 2)`
		`(i >= 3) && (i > j)`
			`1 (cierto)`
			`0 (falso)`
			`0 (falso)`
			`0 (falso)`

Disyunción (∨)	`||`	`(j >= 0) || (j < i)`
		`(j <= 0) || (j > i)`
		`(j >= 0) || (j >= i)`
		`(j <= 0) || (j <= i)`
			`1 (cierto)`
			`1 (cierto)`
			`1 (cierto)`
			`0 (falso)`

Negación (¬)	`!`	`!(j>0)` `!(i>j)`	`0 (falso)` `1 (cierto)`

Nótese que el resultado de una condición o expresión lógica es siempre `0` (si la condición es `falsa`) o `1` (si la condición es `cierta`). Ahora bien, *en el lenguaje C, cualquier expresión (lógica o no) cuyo resultado es distinto de `0` se interpreta como `cierta`, mientras que las expresiones con resultado igual a `0` se interpretan como `falsas`*. Por ejemplo, si consideramos la declaración siguiente:

```
int a=3;
```

La expresión `(3+4) && (a+1)` es una expresión lógica con resultado `1` (`cierto`), ya que `3+4` es `7` (valor distinto de `0`) y `a+1` es `4` (valor distinto de `0`). De esta forma, la expresión lógica `7 && 4` tiene resultado `1` (`cierto`), ya que los valores distintos de `0` se interpretan como ciertos.

Sin embargo, la expresión `(3+4) && (a-3)` es una expresión lógica con resultado `0` (`falso`), ya que `3+4` es `7` (valor distinto de `0`), pero `a-3` es `0` (valor igual a `0`). Entonces, la expresión lógica `7 && 0` tiene resultado `0` (`falso`), ya que el primer operando (`7`) se interpreta como cierto y el segundo (`0`), como falso.

Precedencia de los operadores

Finalmente, ahora que ya sabemos cómo formar expresiones combinando variables, constantes y operadores, solamente nos queda conocer cómo se evalúan estas expresiones. Existe un orden de prioridad entre los operadores a la hora de evaluar una expresión. Así, en la expresión `3+4*5`, se evalúa primero la subexpresión `4*5` y luego se realiza la suma `3+20`, porque el operador de la multiplicación es más prioritario que el de la suma. El resultado de la expresión es `23`.

Obviamente, al igual que en las expresiones matemáticas, la precedencia natural de las operaciones se puede alterar mediante el uso de paréntesis, que permiten efectuar operaciones aritméticas de una expresión en el orden en que se desee. Siguiendo con el ejemplo anterior, se puede forzar a realizar la suma antes de la multiplicación, escribiendo la expresión de la manera siguiente: `(3+4)*5`. Ahora el resultado de la expresión es `35`. Sin embargo, si no se incluyen los paréntesis, las operaciones se realizarán respetando el orden de precedencia que se indica en la tabla 2.6.

En la tabla 2.6, la prioridad de los operadores va disminuyendo al ir descendiendo en la tabla. Por lo tanto, cuanto menor es el nivel de un operador, mayor es su prioridad. Los operadores de un mismo nivel tienen el mismo orden de prioridad entre ellos y, por tanto, han de resolverse considerando el orden de asociatividad que tenga el nivel del operador.

Por otra parte, la tabla 2.6 no contiene todos los operadores que se analizan en este libro. En capítulos posteriores, se introducirán nuevos operadores y en el anexo 9.A se mostrará la tabla de precedencia completa con todos los operadores estudiados.

A continuación, se muestran algunos ejemplos para determinar el valor de las expresiones, teniendo en cuenta la precedencia de los operadores. Para los ejemplos, se considera la siguiente declaración de variables:

```
int i=2, j=4;
```

Tabla 2.6
Precedencia de los operadores

Nivel	Operador	Símbolo	Asociatividad
1	Paréntesis	()	De izquierda a derecha
2	Negación lógica Signo negativo	! -	De derecha a izquierda
3	Multiplicación División Módulo	* / %	De izquierda a derecha
4	Suma Resta	+ -	De izquierda a derecha
5	Menor que Menor que o igual a Mayor que Mayor que o igual a	< <= > >=	De izquierda a derecha
6	Igual a Distinto que	== !=	De izquierda a derecha
7	Conjunción	&&	De izquierda a derecha
8	Disyunción	||	De izquierda a derecha

Ejemplo 1:

```
i*j%2+10/2-i     /* Equivalente a ((((i*j)%2) + (10/2)) - i)
                    El resultado de la expresión es 3 */
```

El orden en que se realizan las operaciones para determinar el valor de la expresión es el siguiente: los operadores `*`, `%` y `/` tienen mayor prioridad que los operadores `+` y `-`. Los operadores `*`, `%` y `/` tienen la misma prioridad. Por tanto, para realizar las operaciones se tiene en cuenta su orden de asociatividad (de izquierda a derecha): en primer lugar, se realiza la multiplicación `i*j` y luego el módulo. A continuación, se realiza la división `10/2`. Finalmente, después de determinar los valores de las operaciones anteriores, en la expresión quedan todavía las operaciones de suma y resta. La suma y la resta pertenecen al mismo nivel, por lo que tienen la misma prioridad.

Las operaciones se realizan teniendo en cuenta el orden de asociatividad de estos operadores, que vuelve a ser de izquierda a derecha. Por tanto, se realiza primero la suma y después la resta. El resultado de la expresión es `3`.

Ejemplo 2:

```
10*i>j*10+3     /* Equivalente a ((10*i)>((j*10)+3))
                   El resultado de la expresión es 0 (falso) */
```

Ejemplo 3:

```
i==j&&3*i<=5*j+7||3<j

/*Equivalente a: (((i==j)&&((3*i)<=((5*j)+7)))||(3<j))
  El resultado de la expresión es 1 (cierto) */
```

2.5 Sentencia

Una sentencia se define como una acción. Las sentencias más elementales de un programa son la sentencia de asignación y la llamada a una función.

Sentencia de asignación

Esta sentencia asigna un valor a una variable.

La sintaxis general de una sentencia de asignación es la siguiente:

```
nom_var  =  expresión;      /* Asigna a la variable nom_var el
                               resultado de la expresión */
```

Nótense dos detalles importantes de la sintaxis:

- El símbolo que se utiliza para expresar la asignación es el carácter =. Como ya hemos comentado anteriormente, es un error habitual en los programadores noveles confundir el operador de asignación (=) con el operador relacional de igualdad (==).
- Las sentencias de asignación finalizan siempre con el carácter punto y coma (;).

A continuación se proporcionan algunos ejemplos de sentencias de asignación considerando la siguiente declaración de variables:

```
int i;
char carac;
float base, altura, area;
```

Ejemplos de sentencias de asignación:

```
i = 0;          /* Asigna a la variable i el valor 0 */
carac = 'a';    /* Asigna a la variable carac el carácter 'a' */
base = 4.0;     /* Asigna a la variable base el valor 4.0 */
altura = 5.0/2; /* Asigna a la variable altura el valor 2.5 */
area = base*altura;   /* Asigna a la variable area el resultado
                         de la expresión base*altura, que es
                         4.0 * 2.5 = 10.0 */
```

Llamada a una función

Una función es un conjunto de sentencias al cual se le asigna un nombre y que realiza una serie de cálculos o tareas a partir de unos datos concretos. Una función, además, puede devolver o no un valor.

Para llamar a una función es necesario escribir su nombre y, a continuación y entre paréntesis, colocar adecuadamente los datos con que operará la función.

La sintaxis general para llamar a una función es la siguiente:

```
nombre_función(dato1, dato2,…);
                    /* Si la función no devuelve un valor */

nom_var = nombre_función(dato1, dato2, …);
                        /* Si la función devuelve un valor */
```

Los datos de la función son expresiones y reciben el nombre de parámetros reales. Si la función tiene más de un parámetro, entonces se separan utilizando comas. Si la función devuelve un valor, este ha de asignarse a una variable (`nom_var`) utilizando la sentencia de asignación.

Existen dos tipos de funciones: las que ofrece el propio lenguaje de programación y las que implementa el programador. Las funciones propias del lenguaje están organizadas en librerías y nos referiremos a ellas como *funciones de librería*. De momento, solo veremos cómo utilizar las funciones de las librerías del lenguaje C y, más adelante, en los capítulos 8 y 9, veremos cómo definir o crear nuestras propias funciones.

Cuando en un programa se quiere utilizar alguna de las funciones que contiene una librería, esta ha de ser "incluida" en el programa. Para ello, se utiliza la directiva `#include` y, a continuación, el nombre de la librería (`nom_lib.h`) entre los símbolos `<>`.

La sintaxis general para incluir una librería es la siguiente:

```
#include <nom_lib.h>
```

Por tanto, si se quiere utilizar las funciones de la librería matemática se escribirá la siguiente línea al inicio del fichero que contiene el código fuente:

```
#include <math.h>  /* Incluye la librería matemática math.h */
```

A continuación, se proporcionan algunos ejemplos de llamadas a funciones de la librería `math.h`, asumiendo la siguiente declaración de variables:

```
float x=3.14, y=2.3, rcos, rsen, res;
```

Ejemplos de llamadas a funciones:

```
rcos = cosf(x);     /* Llamada a la función cosf, que calcula el
                    coseno de x (en radianes). El resultado (-1.0)
                    quedará almacenado en la variable rcos */

rsen = sinf(x/2);   /* Llamada a la función sinf, que calcula el
                     seno de x/2 (en radianes). El resultado (1.0)
                     quedará almacenado en la variable rsen */
```

```
res = sqrtf(x*x + y*y);
                /* Llamada a la función sqrtf, que calcula la raíz
                   cuadrada de x*x+y*y. El resultado (3.89)
                   quedará almacenado en la variable res */
```

En la gran mayoría de los programas que diseñaremos, vamos a necesitar realizar dos acciones básicas: mostrar información por pantalla y leer información del teclado. Estas acciones se pueden realizar utilizando dos funciones de la librería `stdio.h` (*Standard Input/Output*): la función `printf()` para mostrar información por pantalla y la función `scanf()` para leer información del teclado. Veamos, a continuación, cómo utilizar estas dos funciones con detalle.

Función `printf`: mostrar mensajes por pantalla

La función que se utiliza para mostrar mensajes y valores por pantalla es `printf()`.

En su versión más sencilla, la función `printf()` requiere un solo parámetro: el mensaje que se muestra por pantalla. El mensaje se escribe siempre entre comillas dobles y los caracteres que componen el mensaje han de ser caracteres incluidos en la tabla ASCII del anexo 3.A. Esta tabla (que se explicará con más detalle en el próximo capítulo), incluye los 26 caracteres del alfabeto latino, los caracteres numéricos, signos de puntuación, caracteres de control, etc. Obsérvese que no incluye caracteres acentuados como `á`, `è`, `û`, ni caracteres con diéresis como `ü`, ni caracteres específicos de los idiomas como `ñ`, `ç`, etc.

En realidad, el lenguaje C permite usar cualquier tipo de carácter (esté o no incluido en la tabla ASCII) en los mensajes de texto. Sin embargo, en este libro no se utilizan los caracteres que no están en la tabla ASCII ya que no siempre se muestran correctamente por pantalla (dependerá de la configuración de la máquina).

Ejemplo 1:

```
printf("Hola\n");      /* Muestra por pantalla la palabra
                          Hola y salta una línea */
```

El carácter `\n` es un carácter de control que indica salto de línea.

Para mostrar por pantalla mensajes de texto que además incluyen valores de variables o de expresiones, es necesario incluir entre las comillas dobles del mensaje el especificador de formato: `%d` si queremos incluir un valor entero, `%f` si queremos incluir un valor real o `%c` si queremos incluir un carácter. Además, después de las comillas dobles, y separada por comas, es necesario indicar la variable o expresión asociada al valor que se desea mostrar por pantalla. De esta forma, el `printf` sustituirá el especificador de formato por el valor de la variable o expresión correspondiente.

Ejemplo 2:

```
int a=3, b;

b = a + 1;

printf("%d + %d = %d\n", a, b, a+b);

/* Sustituye el primer especificador de formato (%d) por el valor
   de la variable a, el segundo por el valor de la variable b y
   el tercero por el resultado de la expresión a+b. Muestra por
   pantalla: 3 + 4 = 7 y salta una línea */
```

Ejemplo 3:

```
float b=3.41, h=2.0, a;

char rectan='r';

a = b*h;

printf("El area de %c = %.2f*%.2f = %.2f\n", rectan, b, h, a);

/* Muestra por pantalla: El area de r = 3.41*2.00 = 6.82
   y salta una línea */
```

El formato `%.2f` permite mostrar por pantalla el valor de la variable real con dos decimales. Si se utiliza el especificador de formato `%f`, sin indicar el número de decimales que se desean mostrar, por defecto se muestran `6` decimales.

Ejemplo 4:

```
printf("Dia:%d\tMes:%d\tAnyo:%d\n", 12, 12, 2012);

/* Muestra por pantalla: Dia:12  Mes:12  Anyo:2012 y salta una
   línea  */
```

El carácter de control `\t` incluye una tabulación (introduce un número determinado de espacios en blanco). Este carácter de control es muy útil para mostrar tablas o matrices por pantalla, porque permite mostrar la información perfectamente alineada. En el anexo 3.A, se indican todos los caracteres de control.

Como ya se ha indicado, en los mensajes de texto, se evita siempre utilizar caracteres que no estén incluidos en la tabla ASCII del anexo 3.A. Por eso, en los ejemplos anteriores se ha escrito `area`, `Dia` y `Anyo`, en lugar de `área`, `Día` y `Año`.

Ejemplo 5:

```
float dato=35.78;

printf("El 15%% de %.2f es: %.2f\n", dato, dato*15/100);

/* Muestra por pantalla: El 15% de 35.78 es: 5.37 y salta una
   línea */
```

Para mostrar por pantalla el carácter `%`, es necesario escribir `%%` en el formato especificado en el primer parámetro. Esto es debido a que el carácter `%` se utiliza para indicar un especificador de formato (`%d`, `%f`, etc.). Para poder distinguir si queremos indicar un especificador de formato o queremos mostrar por pantalla los caracteres `%d` (por ejemplo), el carácter `%` se especifica duplicando el carácter. Así, el siguiente `printf` muestra por pantalla el mensaje: El formato `%d` se usa para mostrar enteros.

```
printf("El formato %%d se usa para mostrar enteros.\n");
```

De la misma manera, para mostrar por pantalla el carácter `\`, que se utiliza para especificar caracteres de control, es necesario escribir `\\` en el mensaje de texto. El siguiente `printf` muestra por pantalla el mensaje: El carácter de salto de línea es `\n`.

```
printf("El caracter de salto de linea es \\n.\n");
```

Función `scanf`: leer datos desde el teclado

Para leer datos introducidos desde teclado por el usuario, se utiliza la función `scanf`. Esta función tiene, en su versión más simple, dos parámetros: un especificador de formato escrito entre comillas dobles y una variable precedida por el símbolo `&`. El primer parámetro indica el tipo de dato que se quiere leer (`%d` si es entero, `%f` si es real y `%c` si es carácter) y el segundo parámetro indica la variable donde se guarda el dato introducido por el teclado. El símbolo `&` es un operador llamado *operador de referencia*, que se estudia en detalle más adelante, en el capítulo 9 (Funciones: paso de parámetros por referencia).

Ejemplo 1:

```
int dato;

scanf("%d", &dato);     /* Lee desde el teclado un valor entero y
                           lo guarda en la variable dato */
```

Si se quiere leer más de un dato, es preciso incluir en el primer parámetro todos los especificadores de formato necesarios (tantos como datos se quieran leer). Los parámetros siguientes de la función `scanf` son todas las variables donde se almacenan cada uno de los datos leídos. La primera variable

guarda el primer dato leído del teclado, la segunda variable almacena el segundo dato, etc.

Ejemplo 2:

```
int edad;

float sueldo;

char sexo;

printf("Introduzca sexo(f,m) edad y sueldo: ");

scanf("%c %d %f%*c", &sexo, &edad, &sueldo);

/* Lee desde el teclado un carácter que indica el sexo (f o m) y
   lo guarda en la variable sexo, a continuación lee un valor
   entero y lo guarda en la variable edad. Luego, lee un valor
   real y lo guarda en la variable sueldo */
```

Obsérvese en el ejemplo anterior que, antes del `scanf`, se hace un `printf`. Generalmente, antes de un `scanf` siempre se utiliza un `printf` para indicar al usuario del programa cómo se introducirán los datos (orden y formato).

Además, en el ejemplo anterior la función `scanf` incluye el formato `%*c`. Este formato indica que el carácter siguiente, que se lee del teclado después de introducir el sueldo, se descarta (el carácter asterisco indica que se ignora el carácter). Este carácter suele ser el `\n` (correspondiente a la tecla ENTER) introducido por el usuario al finalizar la entrada de datos. Debido a que el carácter se descarta, no es necesario asignarlo a ninguna variable, como en el caso de los valores introducidos para el sexo, la edad y el sueldo. El carácter asterisco se puede utilizar también con otros especificadores de formato: `%*d`, `%*f`.

La función `scanf` espera del teclado el formato exacto especificado en el primer parámetro. Si los datos no se introducen tal como se especifica en el primer parámetro, la función `scanf` deja de leer de la entrada y finaliza su ejecución, pero no la ejecución del programa. Por ejemplo, si se considera el fragmento de código siguiente:

```
int a, b;

printf("Introduzca dos valores enteros: ");

scanf("%d-%d%*c", &a, &b);
```

y el usuario introduce desde el teclado `7,8`, el programa lee el valor `7` y lo guarda en la variable `a`; sin embargo, el valor `8` no lo lee. Para esta entrada de datos, el formato de entrada no coincide con el formato especificado en el primer parámetro en la función `scanf`, ya que el usuario ha introducido una coma en lugar de un guion. Una vez finalizada la ejecución de la función `scanf`,

el programa continuará ejecutándose con la variable `a`, que vale `7`, y la variable `b`, que vale un valor indeterminado.

Si, en cambio, el usuario introduce los datos: `7-8`, entonces la lectura de los mismos se realiza de forma correcta: en la variable `a` queda almacenado el valor `7` y, en la variable `b`, el valor `8`.

Por último, queremos aclarar que, cuando se ejecuta la función `scanf`, esta queda a la espera de que el usuario introduzca los datos por teclado. Los datos o caracteres que el usuario introduce han de finalizar con el carácter `\n` (tecla `ENTER`) y quedan almacenados internamente dentro de lo que se denomina el *buffer del teclado*. La función `scanf` siempre consulta primero si quedan datos por leer del buffer del teclado. Si es así, lee los datos del buffer y continúa su ejecución. En caso contrario (cuando el buffer está vacío), la función `scanf` se queda esperando a que el usuario introduzca nuevos datos por teclado. Por tanto, hemos de tener muy claro que la función `scanf` no siempre se espera a que el usuario introduzca nuevos datos, sino que depende de si quedan datos por leer en el buffer del teclado.

Por ejemplo, si escribimos el fragmento de código siguiente:

```
char c1, c2;

printf("Introduzca una palabra de dos caracteres: ");

scanf("%c", &c1);

scanf("%c%*c", &c2);
```

el primer `scanf` se queda esperando a que el usuario introduzca los datos por el teclado, porque inicialmente el buffer está vacío. Si el usuario introduce `mi\n` (en total, 3 caracteres: `m`, `i` y el `\n`), el primer `scanf` lee un solo carácter, tal como indica el primer parámetro. Por tanto, lee el valor `m` y lo guarda en la variable `c1`. El resto de caracteres que había introducido (`i\n`) quedan almacenados en el buffer del teclado. A continuación, se ejecuta el segundo `scanf`. Como el buffer no está vacío, lee los datos del buffer. Por tanto, lee el carácter `i`, que guarda en la variable `c2`, y el valor `\n`, que no almacena. Ahora el buffer queda vacío y el siguiente `scanf` que se ejecute quedará a la espera de que el usuario introduzca nuevos datos por el teclado.

2.6 Estructura de un programa

En este capítulo, hemos visto cuatro conceptos básicos para crear programas sencillos en el lenguaje de programación C: declarar e inicializar variables, definir constantes, crear expresiones y, finalmente, escribir sentencias. Finalizamos el capítulo mostrando cómo se combinan estos cuatro conceptos y escribiendo nuestro primer programa en C.

En primer lugar, indicamos que, para escribir comentarios dentro de un programa, se utilizan los símbolos `/*` (para iniciar el comentario) y `*/` (para terminarlo). Si el comentario ocupa una sola línea, se puede utilizar el símbolo `//` (derivado de C++) al inicio del comentario. En este caso, no se requiere colocar al final del comentario ningún carácter especial.

Ejemplo de comentarios:

```
/* Esto es un comentario */

// Esto es otro comentario

/* Este comentario es muy largo
   y ocupa más de una línea */
```

Cualquier texto escrito entre comentarios es ignorado por el compilador y el enlazador a la hora de generar el fichero ejecutable. Por tanto, en los comentarios se puede utilizar cualquier tipo de carácter (acentuado, específico del idioma, etc.).

Los comentarios se utilizan para documentar los programas e indicar qué acciones concretas realizan determinadas partes del código.

La estructura general de un programa en el lenguaje C es la siguiente:

```
/* Incluir librerías */
#include <nom_lib.h>

/* Definir constantes */
#define NOM_CONS valor_cons

/* Definir tipos de datos */
/* Los veremos más adelante (v. capítulos 6 y 7) */

/* Definir prototipos de nuestras funciones */
/* Los veremos más adelante (v. capítulos 8 y 9) */

/* Programa principal */
main()
{
    /* Declarar e inicializar variables locales */
    /* Sentencias */
}

/* Definir funciones */
/* Los veremos más adelante (v. capítulos 8 y 9) */
```

En el esquema anterior, observamos lo siguiente:

– El programa principal está identificado por la palabra `main` y utiliza dos llaves (una para abrir (`{`) y otra para cerrar (`}`)) para identificar el conjunto de variables y sentencias que contiene.

– Las variables se declaran al inicio del programa principal. En la declaración, se reserva espacio en la memoria para almacenar los valores que las variables irán tomando durante la ejecución del programa. Estas variables que se declaran dentro del programa principal se denominan *variables locales*. En el lenguaje C, también se pueden declarar variables globales, que son variables que se declaran fuera del `main` y que tienen unas características distintas a las variables locales. En este libro, solo declaramos y utilizamos variables locales.

– El conjunto de sentencias del programa principal indican las acciones que se realizarán sobre las variables. Las sentencias se ejecutan (de arriba abajo) en el orden en que están escritas.

– Finalmente, nótese que las directivas para incluir librerías y definir constantes se escriben siempre al inicio del fichero y fuera del programa principal. De la misma manera, otros conceptos que se tratan en capítulos posteriores, como la definición de nuevos tipos de datos y funciones propias, también van fuera del `main`.

A continuación, se muestra un ejemplo de un programa que calcula el `20%` de un valor que se pide al usuario. Además, en la tabla 2.7, se indica cómo las variables van cambiando de valor después de la ejecución de cada sentencia.

Ejemplo de un programa:

```
/* Este programa calcula el 20% de un valor leído por pantalla */

/* Incluir librerías */
#include <stdio.h>

/* Definición de constantes */
#define PORCENTAJE 20

/* Programa principal */
main()
{
  int dato;    /* Variable que guarda el valor de la entrada */
  float res;   /* Variable que guarda el 20% del valor
                  de la entrada */

  printf("Introduzca valor: ");
  scanf("%d", &dato);
  res = dato*PORCENTAJE/100.0;
  printf("El %d%% de %d es %.2f\n", PORCENTAJE, dato, res );
}
```

Sentencia	Valor de las variables		Entrada por teclado	Salida en pantalla
	dato	res		
Decl. variables	?	?		
printf(…);	?	?		Introduzca valor:
scanf(…);	28	?	28	
res=…;	28	5.6		
printf (…);	28	5.6		El 20% de 28 es 5.60

Tabla 2.7
Ejecución sentencia a sentencia del programa anterior

En la tabla 2.7, se muestran los valores de las variables `dato` y `res` después de ejecutar la sentencia indicada en la primera columna. El símbolo `?` indica que el valor es indeterminado. Además, en las dos últimas columnas, se muestra cuál es la entrada por teclado y cuáles son los mensajes que se muestran por pantalla.

Antes de finalizar, cabe resaltar que los programas en el lenguaje C se han de escribir de una forma clara para que faciliten la lectura del código. A lo largo de este libro se irán introduciendo las normas que definen el estilo que utilizaremos en este libro. A continuación se indican las cuatro primeras normas y en el anexo 9.B, se recopilan todas ellas:

1. Las llaves (`{}`) del `main` se abren y cierran en una línea exclusiva para ellas, es decir, no se escribe ningún código en la misma línea de la llave. Además, las llaves se han de escribir en la misma columna que la `m` del `main`.
2. Todas las líneas de código (declaración de variables y sentencias) del `main` llevan una indentación o sangrado correspondiente a `2`, `3` o `4` espacios. En este libro, se utiliza una indentación a `2` espacios.
3. El nombre de las variables se escribe en letras minúsculas y el nombre de las constantes, en letras mayúsculas.
4. En los mensajes de textos de la función `printf`, solamente se utilizan los caracteres de la tabla ASCII del anexo 3.A.

En los próximos capítulos, se introducen nuevas normas de estilo, a medida que se tratan nuevas construcciones.

2.7 Ejercicios

1. Escriba la declaración de dos variables enteras, llamadas `a` y `b`, una variable de tipo carácter, llamada `c`, y una variable de tipo real, llamada `d`.

2. Defina la constante `PI` en el lenguaje C.

3. Considerando la declaración y la inicialización de variables siguientes:

```
int i=2, j=5;

float k=3.5;
```

Determine el valor de las expresiones siguientes:

```
a.        (i%2)+(3*j-5)
b.        i*2+j/9*3-1
c.        k<5.0 && i!=3
```

4. Identifique los errores en los programas siguientes y corríjalos:

a.
```
main()
{
  int a;

  5 = a;
}
```

b.
```
main()
{
  char b;

  b = 1.0;
}
```

5. Indique qué muestra por pantalla el programa siguiente:

```
#include <stdio.h>
main()
{
  int a;
  char b,c;

  a = 0;
  b = 'O';
  c = 'S';
  printf("%c%d%c%c%c\n",c,a,c,c,b);
}
```

6.
a. Escriba un programa en el lenguaje C que realice el intercambio del valor de dos variables enteras llamadas `a` y `b`, cuyos valores iniciales se leen por teclado. Muestre por pantalla los valores iniciales de las variables y los valores después de realizar el intercambio.

Ejemplo de ejecución (en negrita, los datos que introduce el usuario):

```
Introduzca los valores de a y b (separados por un espacio):
123 88
Los valores iniciales de a y b son: 123 y 88
Los valores de a y b despues del intercambio son: 88 y 123
```

b. Dibuje una tabla, similar a la tabla 2.7, que indique los valores de las variables después de la ejecución de cada sentencia.

7. Escriba un programa en lenguaje C que lea desde el teclado un valor real correspondiente al radio de una circunferencia y muestre por pantalla el área de la circunferencia.

2.8 Respuesta a los ejercicios propuestos

1.
```
int a, b;
char c;
float d;
```

2.
```
#define PI 3.1416
```

3.

a. `(i%2)+(3*j-5) = 10`

b. `i*2+j/9*3-1 = (i*2)+((j/9)*3)-1 = 4 + 0 - 1 = 3`

c. `k<5.0 && i!=3 = (k<5.0)&&(i!=3) = 1 && 1 = 1 (cierto)`

4.

a.
```
Error en la sentencia de asignación, orden incorrecto.
main()
{
  int a;
  a = 5;
}
```

b.
```
Error en la inicialización de la variable b.
main()
{
  char b;
  b = '1';
}
```

5.
```
S0SSO
```

6.

a.

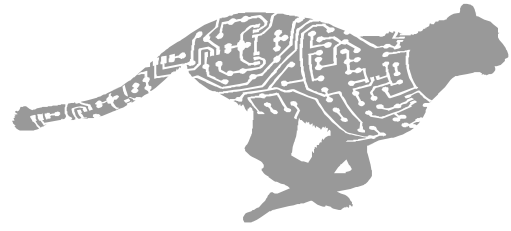
```
#include <stdio.h>
main()
{
  int a, b, c;

  printf("Introduzca los valores de a y b (separados
          por un espacio): ");
  scanf("%d %d%*c", &a, &b);
  printf("Los valores iniciales de a y b son:
          %d %d\n", a, b);
  c = a;
  a = b;
  b = c;
  printf("Los valores de a y b despues del intercambio
        son: %d %d\n", a, b);
}
```

b.

Sentencia	Valor de las variables			Entrada por teclado	Salida en pantalla
	a	b	c		
Decl. Variables	?	?	?		
printf(…);	?	?	?		Introduzca los valores de a y b (separados por un espacio):
scanf(…);	123	88	?	123 88	
printf(…);	123	88	?		Los valores iniciales de a y b son: 123 88
c = a;	123	88	123		
a = b;	88	88	123		
b = c;	88	123	123		
printf(…);					Los valores de a y b despues del intercambio son: 88 y 123

7.

```
#include <stdio.h>
    #define PI 3.1416
       main()
{
  float r, area;

  printf("Introduzca el radio de la circunferencia: ");
  scanf("%f%*c", &r);
  area = PI*r*r;
  printf("El area es: %f\n", area);
}
```



→3

Tipos de datos elementales

En este capítulo, se describen con más detalle los tipos de datos elementales del lenguaje C que se han comentado en el capítulo anterior: los enteros, los caracteres y los reales. Para cada tipo de dato, se muestran todas sus características: su tamaño, su rango de representación, su correspondiente declaración de variables, las operaciones que se pueden realizar y el especificador de formato para la lectura desde el teclado y la escritura por pantalla. Al final del capítulo, se indican las conversiones de tipos de datos elementales que se pueden realizar en el lenguaje C.

3.1 Caracteres

En muchos programas, es necesario manipular caracteres. Un carácter representa un valor alfanumérico: letras, dígitos o caracteres numéricos, signos de puntuación, caracteres de control, etc.

Internamente, cada carácter se representa con un valor numérico. El código que asocia cada carácter con su correspondiente valor numérico es el código ASCII, que son las siglas del American Standard Code for Information Interchange (v. anexo 3.A). Así, por ejemplo, el carácter `'A'` internamente se representa con el valor numérico `65` y el carácter numérico `'0'`, con el valor numérico `48`.

Existe un código ASCII estándar compuesto por 128 caracteres y otro código ASCII extendido, compuesto por 256 caracteres. El código ASCII extendido coincide con el estándar en los primeros 128 caracteres. Los otros 128 caracteres corresponden a caracteres especiales y específicos del idioma, como por ejemplo `ô`, `ó`, `à`, `ñ`, etc.

Además, la tabla ASCII cumple una serie de características que facilitan la implementación de programas para pasar de mayúsculas a minúsculas, o viceversa, ordenar alfabéticamente, pasar de carácter numérico a número entero, o viceversa, etc. Las características de la tabla ASCII son las siguientes:

- Se cumple que `'a' < 'b' < 'c' < … < 'z'`. Esto quiere decir que el valor numérico del carácter `'a'` según la tabla ASCII es menor que el de `'b'` y este menor que el de `'c'`, etc. Además, estos caracteres alfabéticos están almacenados consecutivamente.
- Se cumple que `'A' < 'B' < 'C' < … < 'Z'`, y estos caracteres alfabéticos están almacenados consecutivamente.
- Se cumple que `'0' < '1' < '2' < … < '9'`, y estos caracteres numéricos están almacenados consecutivamente.

Todas las variables de tipo carácter ocupan 1 byte en la memoria. Para declarar una variable de este tipo, se utiliza la palabra reservada `char`, que se puede combinar o no con el calificador `unsigned` (sin signo). El uso del calificador permite definir diferentes rangos de representación. Con el tipo `char`, se representan todos los caracteres de la tabla ASCII estándar y con el tipo `unsigned char`, todos los caracteres de la tabla ASCII extendida.

A continuación, se indican las *características del tipo de dato elemental carácter*.

Tamaño. 1 byte (8 bits)

Rango de representación. Todos los caracteres de la tabla ASCII estándar (valor numérico comprendido entre `0` y `127`)

Si se utiliza el calificador `unsigned`, el rango de representación corresponde con los caracteres de la tabla ASCII extendida (valor numérico comprendido entre `0` y `255`).

Declaración de variables. Se declaran anteponiendo al nombre de la variable la palabra reservada `char` y añadiendo, si es necesario, el calificador `unsigned`.

Ejemplos:

```
char letra;    /* Variable que almacena cualquier carácter
                  de la tabla ASCII estándar (valor numérico
                  entre 0 y 127) */

unsigned char opc;    /* Variable que almacena cualquier
                         carácter de la tabla ASCII extendida
                         (valor numérico entre 0 y 255) */
```

Operaciones. Las operaciones que pueden realizarse con el tipo de dato carácter son las siguientes:

- **Asignación de caracteres**. Por ejemplo, la sentencia `letra = 'c';` asigna el carácter `'c'` a la variable `letra`. Internamente, la variable `letra` almacena el valor numérico `99`.
- **Comparación de caracteres**. Por ejemplo, la expresión `(letra>='a' && letra<='z')` determina si la variable `letra` contiene una `letra` minúscula, comprendida entre la `'a'` y la `'z'`. Si se supone que `letra` contiene el carácter `'c'` del ejemplo anterior, la expresión devolverá `1 (cierto)`.
- **Operaciones aritméticas con caracteres**. A primera vista, parece que no tiene mucho sentido realizar operaciones aritméticas con caracteres. Sin embargo, no es así. Debido a que internamente los caracteres se representan con un valor numérico (el que le corresponde, según el código ASCII), se puede operar con ellos como si fueran números enteros y convertir un carácter en otro. Por ejemplo, para convertir el carácter almacenado en `letra` de minúscula a mayúscula, se hace la operación siguiente:

```
letra = letra - ('a' - 'A');
```

  donde `'a' - 'A'` indica la distancia que hay en la tabla ASCII entre el carácter `'a'` y el carácter `'A'`. Como los caracteres alfabéticos son consecutivos en la tabla, la distancia entre un carácter en minúscula y su correspondiente en mayúscula es constante e igual a `'a' - 'A' = 'b' - 'B' = … = 'z' - 'Z' = 32`. Por tanto, restando esta distancia a cualquier carácter, se convierte un carácter en minúscula a su correspondiente en mayúscula.

  Si se quiere realizar la conversión contraria y convertir el carácter en `letra` de mayúscula a minúscula, se hace la operación siguiente:

```
letra = letra + ('a' - 'A');
```

Lectura desde el teclado y escritura en pantalla. El especificador de formato que se utiliza en las funciones `scanf` y/o `printf` para una variable de tipo carácter es `%c`.

Ejemplo:

```
char letra;
unsigned char opc;

scanf("%c %c",&letra, &opc);  /* Lee dos caracteres de teclado y
                                 los guarda en letra (de tipo char)
                                 y en opc (de tipo unsigned
                                 char) */
```

```
printf("%c %c\n", letra, opc); /* Muestra por pantalla los
                                  caracteres guardados en letra y
                                  opc */
```

La tabla siguiente resume las características del tipo elemental carácter:

Tabla 3.1 Características del tipo elemental carácter

Tipo elemental	Declaración	Rango de representación	Tamaño en bytes	Especificador de formato
Carácter	`char`	Tabla ASCII estándar (0 a 127)	1	`%c`
	`unsigned` `char`	Tabla ASCII extendida (0 a 255)	1	`%c`

Finalmente, cabe comentar que no siempre se pueden mostrar por pantalla los caracteres cuyos valores numéricos van del 128 al 255 de la tabla ASCII extendida. Eso dependerá del entorno de programación que se esté utilizando.

3.2 Enteros

El tipo de dato entero representa los valores enteros positivos y negativos. El lenguaje C permite diferentes variantes del tipo de dato entero (naturales y enteros), dependiendo del calificador de tipo utilizado en la declaración de las variables. Para declarar una variable de este tipo, se utiliza la palabra reservada `int`, que se puede combinar con distintos calificadores para definir diferentes rangos de representación. Los calificadores permitidos para el tipo `int` son: `short` (corto), `long` (largo), `unsigned` (sin signo).

A continuación, se indican las *características del tipo de dato elemental entero*.

Tamaño. El tipo de dato entero ocupa de `2` a `8` bytes en la memoria, dependiendo de la arquitectura, del sistema operativo y del calificador utilizado en la declaración.

Rango de representación. Al igual que el tamaño, el rango de representación del tipo de dato entero depende también de la arquitectura y del calificador utilizado en la declaración. En la tabla 3.2, se muestran con detalle el tamaño y el rango de representación de las diferentes variantes del tipo entero para una arquitectura de `64` bits y el sistema operativo Linux.

Declaración de variables. Las variables de tipo entero se declaran anteponiendo al nombre de la variable la palabra reservada `int`. Al igual que el tipo carácter, se pueden utilizar, si es necesario, uno o varios de los calificadores válidos (`short`, `long` y/o `unsigned`) para definir diferentes rangos de representación. Los calificadores `short` y `long` permiten modificar el tamaño que ocupa la variable en la memoria y, por tanto, su rango de representación. El calificador `unsigned` indica que es una variable sin signo, por lo que únicamente puede almacenar valores naturales (enteros positivos). Las diferentes variantes en la

declaración de variables de tipo entero, suponiendo una arquitectura de 64 bits, son:

- `short int` (o `short`): permite almacenar valores enteros de tamaño 2 bytes.
- `int`: permite almacenar valores enteros de 4 bytes de tamaño.
- `long int` (o `long`): permite almacenar valores enteros de 8 bytes de tamaño.
- `unsigned short int` (o `unsigned short`): permite almacenar valores naturales de 2 bytes de tamaño.
- `unsigned int`: permite almacenar valores naturales de 4 bytes de tamaño.
- `unsigned long int` (o `unsigned long`): permite almacenar valores naturales de 8 bytes de tamaño.

Para cada una de las variantes, se han indicado, entre paréntesis, otras declaraciones equivalentes. Obsérvese que, en algunos casos, cabe prescindir de la palabra reservada `int` en la declaración e indicar únicamente los calificadores.

Ejemplos:

```
int i;                    /* Variable que almacena un valor
                             entero de 4 bytes de tamaño */

unsigned short edad;      /* Variable que almacena un valor
                             natural de 2 bytes de tamaño */
```

Operaciones. Las operaciones que se pueden realizar con el tipo de dato entero son las siguientes:

- **Asignación de enteros**. A continuación, se muestran algunos ejemplos de sentencias para asignar un valor constante a una variable de tipo entero:

```
i = -1;           /* Asigna -1 a la variable i */
edad = 17;        /* Asigna 17 a la variable edad */
```

- **Comparación de enteros**. Por ejemplo, si se consideran las inicializaciones anteriores, las expresiones siguientes que comparan valores enteros son válidas:

```
i != 10          /* Expresión con resultado 1 (cierto) */
edad >= 18       /* Expresión con resultado 0 (falso) */
edad >= 17 && edad < 65   /* Expresión con resultado 1
                             (cierto) */
```

- **Operaciones aritméticas**. El tipo de dato entero permite utilizar los operadores aritméticos `+`, `-`, `*`, `/` y `%`. Por ejemplo, si se consideran las inicializaciones anteriores de las variables `i` y `edad` (v. asignación de enteros), las sentencias siguientes son correctas:

```
i = (i + 3)*2 - i;    /* La variable i vale 5 después de
                         la asignación */

edad = (edad/2) % 3; /* La variable edad vale 2 después
                        de la asignación */
```

Téngase en cuenta que el resultado de la operación `edad/2` es `8` y no `8.5` porque los dos operandos (el numerador y el denominador) son enteros.

Lectura desde el teclado y escritura en pantalla. Los especificadores de formato que se utilizan en las funciones `scanf` y/o `printf` para una variable de tipo entero son: `%d` si la variable es `int`, `%hd` si la variable es `short int`, `%ld` si la variable es `long int`, `%hu` si la variable es `unsigned short`, `%u` si la variable es `unsigned int` y `%lu` si la variable es `unsigned long int`.

Ejemplo:

```
int i;                  /* Variable entera */
unsigned short edad;    /* Variable entera sin signo (natural) */

scanf("%d %hu",&i, &edad);  /* Lee dos enteros de teclado (int y
                               unsigned short) y los guarda en
                               i y edad */
printf("%d %hu\n",i, edad); /* Muestra por pantalla los enteros
                               guardados en i y edad */
```

Finalmente, cabe comentar que, como el tipo de dato `char` internamente está representado mediante un valor numérico de tamaño `1` byte, también puede utilizarse una variable de este tipo (`char` o `unsigned char`) como una variable entera. Para mostrar el valor numérico de una variable de este tipo, se utilizan los especificadores de formato `%hhd` si la variable es de tipo `char` y `%hhu` si es de tipo `unsigned char`.

Tabla 3.2 Características del tipo elemental entero en una arquitectura de 64 bits y sistema operativo Linux

Tipo elemental	Declaración	Rango de Representación	Tamaño en bytes	Especificador de formato
Entero	`char`	-128 a 127	1	`%hhd`
	`short int` (o `short`)	-32768 a 32767	2	`%hd`
	`int`	-2147483648 a 2147483647	4	`%d`
	`long int` (o `long`)	-9223372036854775808 a 9223372036854775807	8	`%ld`
	`unsigned char`	0 a 255	1	`%hhu`
	`unsigned short int` (o `unsigned short`)	0 a 65535	2	`%hu`
	`unsigned int` (o `unsigned`)	0 a 4294967295	4	`%u`
	`unsigned long int` (o `unsigned long`)	0 a 18446744073709551615	8	`%lu`

La tabla 3.2 resume las características del tipo elemental entero en una arquitectura de 64 bits y sistema operativo Linux. En esta tabla, también se incluyen las características de los tipos de datos `char` y `unsigned char` para el caso en que se utilicen como tipo entero.

3.3 Reales

El tipo de dato real representa los valores reales. El lenguaje C permite diferentes variantes del tipo de dato real que se diferencian entre ellas por su rango de representación y su precisión. Para declarar una variable de este tipo, se utilizan las palabras reservadas `float` o `double`. Esta última, además, se puede combinar con el calificador `long` (largo) para obtener mayor rango y precisión.

A continuación, se indican las *características del tipo de dato elemental real*.

Tamaño. El tipo de dato real ocupa de 4 a 16 bytes en la memoria, dependiendo de la arquitectura y del tipo de dato utilizado en la declaración.

Rango de representación. El rango de representación del tipo de dato real depende también de la arquitectura y del tipo de dato utilizado en la declaración. En la tabla 3.3, se muestran con detalle el tamaño y el rango de representación de las diferentes variantes para una arquitectura de 64 bits con el sistema operativo Linux.

Declaración de variables. Las variables de tipo real se declaran anteponiendo al nombre de la variable las palabras `float`, `double` o `long double`, según corresponda.

Ejemplos:

```
float a;            /* Variable real */
double b;           /* Variable real con mayor rango y
                       precisión que la anterior */
```

Operaciones. Las operaciones que se pueden realizar con el tipo de dato real son las siguientes:

– Asignación de reales. A continuación, se muestran algunos ejemplos de sentencias para asignar un valor real constante a una variable de tipo real. Obsérvese que, en el lenguaje C, los números reales se pueden representar con dos notaciones: la decimal (`3.52`) y la científica (`1.7E-300`)

```
a = 3.52;         /* Asigna 3.52 a la variable a */
b = 1.7E-300;     /* Asigna 1.7*10^-300 a la variable b */
```

– **Comparación de reales.** Por ejemplo, si se consideran las inicializaciones anteriores, las expresiones siguientes que comparan valores reales son válidas:

```
a >= 27.0       /* Expresión con resultado 0 (falso) */
b <= 18         /* Expresión con resultado 1 (cierto) */
a*2 >= 17 || b < 0  /* Expresión con resultado 0
                       (falso) */
```

Sin embargo, el lector debe tener en cuenta que los números reales no han de compararse nunca en igualdad (==) o desigualdad (!=) exacta, porque rara vez se encuentran dos reales exactamente iguales. Para comparar reales en igualdad, se utiliza siempre un margen de error. Por ejemplo, para comparar si la variable `a` es igual a `7.04/2`, no se hace `(a == 7.04/2)` porque, incluso cuando la variable `a` almacene el valor `3.52`, el resultado de esta expresión será siempre `falso`. La comparación se realiza de la manera siguiente:

```
((a >= 7.04/2 - ERROR) && (a <= 7.04/2 + ERROR))
```

donde `ERROR` es una constante que define la precisión deseada y que podría estar definida como:

```
#define ERROR 0.1E-02
```

– **Operaciones aritméticas.** El tipo de dato real permite utilizar los operadores aritméticos `+`, `-`, `*`, y `/`. Si se consideran las inicializaciones anteriores de las variables `a` y `b` (v. asignación de reales), las sentencias siguientes son correctas:

```
a = a + 3.0 - 8.0;    /* La variable a vale -2.52
                         después de la asignación */
b = b*1.5/2;          /* La variable b vale 1.28*10^-300
                         después de la asignación */
```

Obsérvese que el resultado de la división `b*1.5/2` es real, ya que el numerador `b*1.5` es real (`double`).

Lectura desde el teclado y escritura en pantalla. Los especificadores de formato que se utilizan en las funciones `scanf` y/o `printf` para una variable de tipo real son, principalmente, `%f` y `%e` si la variable es `float`; `%lf` y `%le` si la variable es `double`, y `%Lf` y `%Le` si la variable es `long double`. La diferencia entre estos formatos es el uso de la notación decimal o científica al mostrar el valor por pantalla. Existen muchos otros especificadores de formato, pero en este libro solo se utilizan los de la tabla 3.3.

Ejemplo:

```
float a;                /* Variable real */
double b;               /* Variable real */
```

```
scanf("%f %lf",&a, &b);    /* Lee dos reales de teclado (float
                              y double) y los guardar en a y b */
printf("%f %lf\n", a, b); /* Muestra por pantalla los reales
                              guardados en a y b */
```

La tabla 3.3 resume las características del tipo elemental real en una arquitectura de 64 bits y con el sistema operativo Linux.

Tipo elemental	Declaración	Rango de representación	Tamaño en bytes	Especificador de formato
Real	`float`	$\pm 1.1754945 * 10^{-38}$ $\pm 3.4028232 * 10^{+38}$	4	`%f` (decimal) `%e` (científica)
	`double`	$\pm 2.225 * 10^{-308}$ $\pm 1.798 * 10^{+308}$	8	`%lf` (decimal) `%le` (científica)
	`long double`	$\pm 8.4 * 10^{-4932}$ $\pm 5.9 * 10^{+4932}$	16	`%Lf` (decimal) `%Le` (científica)

Tabla 3.3
Características del tipo elemental real en una arquitectura de 64 bits y sistema operativo Linux

3.4 Conversión de tipos de datos elementales

El lenguaje C permite asignar una variable o expresión de un tipo de dato determinado a una variable de otro tipo de dato diferente, lo que da lugar a una conversión de tipos. Por regla general, en la conversión, el tipo de dato de la expresión de la derecha de la asignación se convierte en el tipo de dato de la variable a la cual se asigna el valor.

La conversión de tipos puede ser explícita y, en este caso, se realiza utilizando la construcción `cast`, que consiste en indicar explícitamente el tipo de dato al cual se desea convertir una expresión.

La sintaxis general de la construcción `cast` es la siguiente:

```
(nombre del tipo deseado) expresión
```

A continuación, se muestra qué ocurre en las conversiones de tipo más frecuentes.

De carácter a entero. Como ya se ha comentado anteriormente, los caracteres internamente ya se representan con valores enteros. Por tanto, los caracteres se convierten directamente en el entero que corresponde según la tabla ASCII.

Ejemplo:

```
int num;
char car='1';        /* El carácter numérico '1' tiene el valor
```

```
                          49 en la tabla ASCII */
num = car;              /* Conversión implícita que asigna 49 a
                           la variable num */
num = (int) car;        /* Conversión explícita que asigna 49 a
                           la variable num  */
```

De entero a carácter. Un entero ocupa `4` bytes en la memoria, mientras que un carácter solo ocupa `1` byte. Asignar un entero a un carácter implica quedarse únicamente con `1` de los `4` bytes del entero. Por regla general, siempre nos quedamos con el byte de menor peso de los `4` bytes que ocupa en la memoria. Así, si el entero almacena el valor decimal `360`, que en binario se representa por:

```
00000000 00000000 00000001 01101000
```

entonces el byte de menor peso es `01101000`, que en la base decimal es el valor entero `104`. Por tanto, cuando se asigna un entero a un carácter, la variable de tipo carácter almacena el byte de menor peso del entero que se le asigna.

Ejemplo:

```
int num1=65;        /* 65 en binario es:
                       00000000 00000000 00000000 01000001 */
int num2=360;       /* 360 en binario es:
                       00000000 00000000 00000001 01101000 */
char letra;

letra = num1;       /* Conversión implícita que asigna el número
                       correspondiente al byte de menor peso de la
                       variable num1 (01000001, que en la base
                       decimal es 65) a la variable letra, por lo
                       que letra guardará el carácter 'A'  */
letra = (char) num2;  /* Conversión explícita que asigna
                         el número correspondiente al byte de
                         menor peso de la variable num2
                         (01101000, que en la base decimal es
                         104) a la variable letra, por lo que
                         letra guardará el carácter 'h'  */
```

De entero a entero de menor tamaño. Cuando se hace una conversión de un valor entero que se almacena en `N` bytes a otro de tamaño `M` bytes, con `M<N`, el entero de mayor tamaño se convierte en el de menor tamaño, eliminando los bytes de orden superior.

Ejemplo:

```
short num1;
```

```
int num2=65000;        /* 65000 en binario es:
                          00000000 00000000 11111101 11101000 */

num1 = num2;           /* Conversión implícita que asigna -536 a
                          la variable num1 */
num1 = (short) num2;   /* Conversión explícita */
```

En este caso, la variable `num1` contiene los `2` bytes de menor peso de la variable `num2` (`11111101 11101000`) que representan el valor entero `-536` en la base decimal. Para visualizar por pantalla el valor de las variables `num1` y `num2`, hay que utilizar la función `printf` de la manera siguiente:

```
printf("num1 = %hd num2 = %d\n", num1, num2);
/* Muestra por pantalla los valores guardados en num1 y num2 */
```

De entero a real. Cuando asignamos un valor entero a una variable de tipo real, se produce una aproximación, por defecto o por exceso, que deja en el real el valor representable más cercano al valor entero.

Ejemplo:

```
int n1=3;
int n2=2147483645;
float nreal;

nreal = n1;           /* Conversión implícita que asigna 3.0
                         a la variable nreal */

nreal = (float) n1;   /* Conversión explícita */
nreal = n2;           /* Conversión implícita que asigna
                         2147483648.0 a la variable nreal */

nreal = (float) n2;   /* Conversión explícita */
```

Obsérvese que el valor entero `2147483645` no se puede representar exactamente en un `float` y que el valor más cercano representable es `2147483648.0`

De real a entero. Esta conversión produce un truncamiento de la parte decimal del número real.

Ejemplo:

```
int num;
float nreal = 16.81;

num = nreal;          /* Conversión implícita que asigna 16 a
                         la variable num */
```

```
num = (int) nreal;     /* Conversión explícita */
```

De real a real de menor tamaño. Cuando se quiere hacer una conversión de un valor real que se almacena en `N` bytes a otro de tamaño `M` bytes, con `M<N`, se produce una aproximación, que deja en el real de menor tamaño el valor real representable más cercano al valor real de mayor tamaño.

Ejemplo:

```
float n1;
double n2=32433.320945;

n1 = n2;          /* Conversión implícita que asigna 32433.320312
                     a la variable n1 */
n1 = (float) n2; /* Conversión explícita */
```

En este caso, la variable `n1` contiene el valor real representable más cercano al valor de `n2` (en este caso, `32433.320312`). Para visualizar por pantalla el valor de las variables `n1` y `n2`, hay que utilizar la función `printf` de la manera siguiente:

```
printf("n1=%f n2=%lf\n",n1,n2);/* Muestra por pantalla los
                                  valores guardados en n1 y n2 */
```

Hasta aquí se han mostrado los tipos de conversiones más frecuentes que pueden encontrarse al asignar el resultado de una expresión de un tipo determinado a una variable de tipo diferente. Finalmente, cabe aclarar que cuando una expresión combina operandos de distinto tipo internamente para cada operación, los operandos son convertidos al tipo de dato "superior" para realizar la operación. Posteriormente, si la variable a la cual se asigna el resultado es de un tipo de dato diferente al de la expresión, se vuelve a realizar otra conversión siguiendo las reglas explicadas anteriormente. El orden ascendente de los tipos de datos es el siguiente:

```
char < short < int < long int < float < double < long double
```

Así, por ejemplo, si se ejecuta la sentencia siguiente:

```
int res;
int num=3, den=2;
res = num/den + 3.7;     /* Asigna el valor 4 a la variable res */
```

Primero, se realiza la división entera `num/den`, cuyo resultado es `1`. Aquí no se efectúa ninguna conversión de tipos porque ambos operandos son enteros y la división es entera. A continuación, se realiza la suma de `1 + 3.7`; como el segundo operando es real (`3.7`), el entero `1` internamente se convierte en el valor real `1.0` y se realiza la suma. El resultado de la expresión es `4.7`. Final-

mente, en `res` queda almacenado el valor `4`, ya que al hacer la asignación se está pasando de real a entero y, por tanto, hay un truncamiento de la parte decimal.

Si se quiere que la división del ejemplo anterior sea una división real, debe expresarse explícitamente la conversión de tipo de uno de los operandos de la manera siguiente:

```
res = (float) num/den + 3.7;    /* Asigna el valor 5 a la
                                   variable res */
```

En este caso, el resultado de la división es `1.5`, porque explícitamente se ha efectuado una conversión de tipo de entero a real del primer operando (`num`), lo que hace que el segundo operando (`den`) también pase a ser de tipo `float` antes de hacer la operación de división. Después de realizar la operación de división, se realiza la operación de suma, de modo que el resultado de la expresión es `1.5+3.7=5.2`. Finalmente, en la variable `res` queda almacenado el valor entero `5`.

Por tanto, es preciso conocer las reglas en las conversiones de tipo para saber cómo se evalúan las expresiones que combinan diferentes tipos de datos y forzar, de forma explícita, la conversión deseada, si es necesario.

3.5 Ejercicios

1. Ejecute en su ordenador el programa siguiente, escrito en lenguaje C, para conocer el tamaño en bytes de todos los tipos elementales que utiliza su entorno de programación.

```
#include <stdio.h>
main()
{
  char c;
  short s;
  int i;
  long int li;
  float f;
  double d;
  long double ld;
  unsigned char uc;
  unsigned short us;
  unsigned int ui;
  unsigned long int uli;

  printf("Size char=%lu bytes\n", sizeof(c));
  printf("Size short=%lu bytes\n", sizeof(s));
  printf("Size int=%lu bytes\n", sizeof(i));
```

```
    printf("Size long int=%lu bytes\n", sizeof(li));
    printf("Size float=%lu bytes\n", sizeof(f));
    printf("Size double=%lu bytes\n", sizeof(d));
    printf("Size long double=%lu bytes\n", sizeof(ld));
    printf("Size unsigned char=%lu bytes\n", sizeof(uc));
    printf("Size unsigned short=%lu bytes\n",
            sizeof(us));
    printf("Size unsigned int=%lu bytes\n", sizeof(ui));
    printf("Size unsigned long int=%lu bytes\n",
            sizeof(uli));
     }
```

2. Escriba un programa en lenguaje C que lea desde el teclado un carácter en letra mayúscula y muestre por pantalla su carácter correspondiente en letra minúscula.

3. Escriba un programa en lenguaje C que lea desde el teclado un carácter numérico (utilizando el especificador de formato `%c` en la instrucción `scanf`) comprendido entre `0` y `9`, y muestre por pantalla un mensaje que indique su valor entero equivalente y su código en la tabla ASCII correspondiente (utilizando el especificador de formato `%d` en ambos casos).

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca un caracter numerico entre 0 y 9: 4
Su valor numerico correspondiente es: 4
Su codigo en la tabla ASCII es: 52
```

4. Escriba un programa en lenguaje C que lea desde el teclado un número natural (utilizando el especificador de formato `%d` en la instrucción `scanf`) comprendido entre `0` y `127`, y muestre por pantalla el carácter que dicho número representa en la tabla ASCII.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca un valor entero entre 0 y 127: 77
El caracter en la tabla ASCII correspondiente al valor
77 es M
```

5. El programa siguiente en lenguaje C genera un error al compilarlo. Determine qué sentencia está generando el error y escriba correctamente el programa para que se muestre por pantalla el mensaje `b=0`.

```
#include <stdio.h>
main()
{
  int b;
  float a, c;
```

```
  a = 3.8;
  c = 4.5;
  b = (a+c)%4;
  printf("b = %d\n", b);
}
```

Utilizando únicamente la construcción `cast`, ¿cómo rescribiría la sentencia `b = (a+c)%4;` para que el programa muestre por pantalla el mensaje `b=3`?

6. Escriba un programa en lenguaje C que lea desde el teclado dos números enteros positivos y muestre por pantalla el valor del porcentaje que representa el segundo número sobre el primero. Por ejemplo, si los números leídos son, respectivamente, `5` y `2`, entonces por pantalla se mostrará el mensaje: `2 es el 40.0% de 5`.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca dos numeros enteros positivos (separados por espa-
cio): 8 3
3 es el 37.5% de 8
```

7. Dados los programas siguientes, escritos en lenguaje C, indique para cada caso la salida obtenida por pantalla:

a.

```
#include <stdio.h>
main()
{
  int a;
  float b = 3.7189;

  a = b;
  printf("a = %d\n", a);
}
```

b.

```
#include <stdio.h>
main()
{
  int cod;
  char car;

  car = 'B';
  cod = 75;

  printf("El codigo ASCII para la letra %c es %hhd\n", car,
  car);
  printf("El codigo ASCII %d representa el caracter %c\n",
  cod, cod);
}
```

c.

```
#include <stdio.h>
main()
{
  char car;
  int a, b=3, e, f;
  float c, d=10.8;

  a = d;
  printf("La variable a solo almacena valores enteros, enton-
          ces: a = %d. Por tanto, si le asignamos un valor
          real, la parte decimal se trunca\n", a);

  printf("\nEl resultado de dividir %d entre %d es un valor
          entero igual al valor de %d\n", a, b, a/b);

  c = a;
  printf("\nAl asignar a la variable c el valor de la varia-
          ble a = %d, el valor de la variable c es %.1f\n",
          a, c);

  printf("\nAl sumar el valor entero de la variable a con el
          valor real de la variable c, obtenemos un resultado
          real: a + c = %d + %.1f = %.1f\n", a, c, c+a);

  car = 2*a*b;
  printf("\nAl asignar a la variable car el valor     entero
          2*a*b = %d, obtenemos el valor %d que corresponde
          al carácter ASCII %c\n", 2*a*b, car, car);

  d = b/2;
  printf("\nAritmetica entera: b/2 = %d/2 = %.1f\n",
           b, d);

  d = b/2.0;
  printf("\nAritmetica en punto flotante: b/2.0 = %d/2.0 =
           %.1f\n", b, d);

  e = a/b + 3.8;
  printf("\nAritmetica en punto flotante: a/b + 3.8 = %d + 3.8
           = %d\n", a/b, e);

  f = (float) a/b + 3.8;
  printf("\nAritmetica en punto flotante: (float) a/b + 3.8 =
           %.1f + 3.8 = %d\n", (float) a/b, f);
}
```

3.6 Respuesta a los ejercicios propuestos

1.

```
Size char=1 bytes
Size short=2 bytes
Size int=4 bytes
Size long int=8 bytes
Size float=4 bytes
Size double=8 bytes
Size long double=16 bytes
Size unsigned char=1 bytes
Size unsigned short=2 bytes
Size unsigned int=4 bytes
Size unsigned long int=8 bytes
```

2.

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

  printf("Introduzca un caracter en letra mayuscula: ");
  scanf("%c%*c", &c);
  c = c  + ('a' - 'A');
  printf("El caracter en letra minuscula es: %c\n", c);
}
```

3.

```
#include <stdio.h>
main ()
{
  int val;      /* Entero entre 0 y 9 */
  char cn;      /* Carácter numérico */

  printf("\nIntroduzca un caracter numerico
            entre 0 y 9: ");
  scanf("%c%*c", &cn);

  val = cn - '0';
  printf("El valor numerico correspondiente
          es: %d\n", val);
  printf("Su codigo en la tabla ASCII es: %d\n", cn);
}
```

4.

```
#include <stdio.h>
main ()
{
  int num;      /* Natural entre 0 y 127 */
  char car;     /* Carácter */

  printf("\nIntroduzca un valor entero entre 0 y 127: ");
  scanf("%d%*c", &num);
```

```
  car = (char)num;
  printf("El caracter en la tabla ASCII correspondiente al
          valor %d es %c\n", num, car);
}
```

5.

El error del programa es que los operandos del operador módulo (`%`) han de ser de tipo entero. En este caso, el operando `(a+c)` es un operando real, debido a que las variables `a` y `c` son de tipo real. Esto implica que el resultado de la suma entre las variables `a` y `c` también es de tipo real (`float`).

Para solucionar este error, hay que forzar explícitamente que el operando `(a+c)` sea de tipo entero. Esto se hace utilizando la construcción `cast`, es decir, anteponiendo `(int)` al operando `(a+c)`.

```
#include <stdio.h>
main()
{
  int b;
  float a, c;

  a = 3.8;
  c = 4.5;
  b = (int)(a+c)%4;
  printf("b = %d\n", b);
}
```

Para que el programa muestre por pantalla el mensaje `b=3`, la sentencia en negrita ha de escribirse de la manera siguiente:

```
b = ((int)a + (int)c) % 4;
```

6.

```
#include <stdio.h>
main ()
{
  int num1, num2;    /* Enteros positivos */
   float por;         /* Valor del % que representa el   se-
gundo número sobre el primero */

  printf("\n Introduzca dos numeros enteros positivos
           (separados por espacio): ");
  scanf("%d %d%*c", &num1, &num2);

  por = 100*num2/(float) num1; /* por = 100.0*num2/num1 */

  printf("%d es el %.1f%% de %d\n", num2, por, num1);
}
```

7.

a.

```
a = 3
```

b.

El codigo ASCII para la letra B es 66
El codigo ASCII 75 representa el caracter K

c.

La variable a solo almacena valores enteros, entonces: a=10. Por tanto, si le asignamos un valor real, la parte decimal se trunca

El resultado de dividir 10 entre 3 es un valor entero igual al valor de 3

Al asignar a la variable c el valor de la variable a = 10, el valor de la variable c es 10.0

Al sumar el valor entero de la variable a con el valor real de la variable c, obtenemos un resultado real: a + c = 10 + 10.0 = 20.0

Al asignar a la variable car el valor entero 2*a*b = 60, obtenemos el valor 60 que corresponde al carácter ASCII <

Aritmetica entera: b/2 = 3/2 = 1.0

Aritmetica en punto flotante: b/2.0 = 3/2.0 = 1.5

Aritmetica en punto flotante: a/b + 3.8 = 3 + 3.8 = 6

Aritmetica en punto flotante: (float) a/b + 3.8 = 3.3 + 3.8 = 7

3.7 Anexos

Anexo 3.A. Tabla ASCII

Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car
0	00	NUL \0	**32**	20	espacio	**64**	40	@	**96**	60	`
1	01	SOH	**33**	21	!	**65**	41	A	**97**	61	a
2	02	STX	**34**	22	"	**66**	42	B	**98**	62	b
3	03	ETX	**35**	23	#	**67**	43	C	**99**	63	c
4	04	EOT	**36**	24	$	**68**	44	D	**100**	64	d
5	05	ENQ	**37**	25	%	**69**	45	E	**101**	65	e
6	06	ACK	**38**	26	&	**70**	46	F	**102**	66	f
7	07	BEL \a	**39**	27	'	**71**	47	G	**103**	67	g
8	08	BS \b	**40**	28	(	**72**	48	H	**104**	68	H
9	09	TAB \t	**41**	29	)	**73**	49	I	**105**	69	I
10	0A	LF \n	**42**	2A	*	**74**	4A	J	**106**	6A	J
11	0B	VT \v	**43**	2B	+	**75**	4B	K	**107**	6B	K
12	0C	FF \f	**44**	2C	,	**76**	4C	L	**108**	6C	L
13	0D	CR \r	**45**	2D	-	**77**	4D	M	**109**	6D	M
14	0E	SO	**46**	2E	_	**78**	4E	N	**110**	6E	N
15	0F	SI	**47**	2F	/	**79**	4F	O	**111**	6F	O
16	10	DEL	**48**	30	0	**80**	50	P	**112**	70	P
17	11	DC1	**49**	31	1	**81**	51	Q	**113**	71	Q
18	12	DC2	**50**	32	2	**82**	52	R	**114**	72	R
19	13	DC3	**51**	33	3	**83**	53	S	**115**	73	s
20	14	DC4	**52**	34	4	**84**	54	T	**116**	74	t
21	15	NAK	**53**	35	5	**85**	55	U	**117**	75	u
22	16	SYN	**54**	36	6	**86**	56	V	**118**	76	v
23	17	ETB	**55**	37	7	**87**	57	W	**119**	77	w
24	18	CAN	**56**	38	8	**88**	58	X	**120**	78	x
25	19	EM	**57**	39	9	**89**	59	Y	**121**	79	y
26	1A	SUB	**58**	3A	:	**90**	5A	Z	**122**	7A	z
27	1B	ESC	**59**	3B	;	**91**	5B	[	**123**	7B	{
28	1C	FS	**60**	3C	<	**92**	5C	\	**124**	7C	|
29	1D	GS	**61**	3D	=	**93**	5D	]	**125**	7D	}
30	1E	RS	**62**	3E	>	**94**	5E	^	**126**	7E	~
31	1F	US	**63**	3F	?	**95**	5F	_	**127**	7F	DEL

La tabla anterior está organizada en cuatro grupos. Cada grupo está formado por tres columnas: **Dec, Hex y Car**.

La columna **Dec** indica los valores numéricos en base decimal de cada entrada de la tabla (numerados consecutivamente del `0` al `127`); la columna **Hex** contiene los valores numéricos en base hexadecimal, y la columna **Car** indica el carácter correspondiente al valor numérico.

Los `32` primeros caracteres de la tabla son caracteres de control. A continuación, se explican algunos de ellos:

`\0`: Fin de línea
`\a`: Campana (alerta)
`\t`: Tabulador horizontal
`\n`: Nueva línea
`\v`: Tabulador vertical
`\f`: Nueva página
`\r`: Retorno de carro

Sentencias condicionales

Las sentencias de un programa en el lenguaje C se ejecutan secuencialmente, es decir, cada una a continuación de la anterior, empezando por la primera y terminando con la última sentencia. Sin embargo, para resolver problemas más complejos, es necesario ejecutar un conjunto de sentencias u otro en función de una determinada condición, o bien ejecutar el mismo conjunto de sentencias un determinado número de veces. El lenguaje C ofrece sentencias que permiten representar este tipo de comportamiento: son las sentencias de control de flujo.

Hay varios tipos de sentencias de control de flujo. En este libro, se analizan las sentencias condicionales y las sentencias iterativas. Las sentencias condicionales permiten ejecutar un conjunto de sentencias determinadas si se cumple una condición, mientras que las sentencias iterativas permiten ejecutar repetidamente un conjunto de sentencias. En este capítulo, se estudian las sentencias condicionales y, en el siguiente, las sentencias iterativas.

El capítulo empieza con la descripción de la sentencia condicional `if-else`, con sus tres variantes, y a continuación se tratan las anidaciones de las sentencias condicionales. Al final del capítulo, se comentan los errores más comunes que cometen los programadores noveles al utilizar estas sentencias.

4.1 Sentencias condicionales

Las sentencias condicionales, también llamadas *sentencias de selección*, permiten efectuar alguna comprobación lógica en algún punto concreto del programa para seleccionar el grupo de sentencias que se ejecutan entre varios grupos disponibles. Las sentencias condicionales que permite el lenguaje C

son: `if-else` (con todas sus variantes) y `switch`. En este libro, solo se analiza la sentencia `if-else`, con sus tres variantes: `if`, `if-else` y `if-else-if`.

Variante `if`

La sentencia `if` es la sentencia condicional más sencilla que permite ejecutar un grupo de sentencias si una determinada condición o expresión lógica es cierta (tiene un valor igual a `1`).

La sintaxis general de la sentencia `if` es la siguiente:

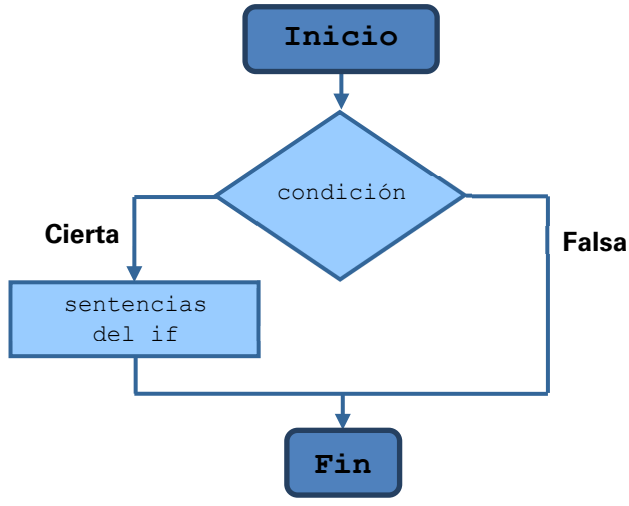
```
if (condición)
{
  sentencias a ejecutar si la condición es cierta;
}
```

Obsérvese que en la sintaxis de esta sentencia aparece un test lógico que depende de una condición. Una condición, tal como se define en el capítulo 2 ("Empezando a programar", sección 2.4), es una expresión lógica que combina operadores relacionales y/o lógicos y cuyo resultado solo puede ser cierto (representado con el valor `1`) o falso (representado con el valor `0`).

De esta forma, al ejecutar esta sentencia `if`, se evalúa primero la condición y, si es cierta, entonces se ejecutan las sentencias del `if`. En caso contrario, no se ejecuta ninguna sentencia. Finalmente, la ejecución de la sentencia `if` termina para cualquiera de los dos resultados de la condición.

La figura 4.1 muestra el diagrama de flujo que describe la estructura algorítmica de esta sentencia.

Fig. 4.1
Diagrama de flujo de la sentencia `if`



Finalmente, cabe tener en cuenta que:

Las sentencias asociadas al `if` pueden ser de cualquier tipo (sentencias de asignación, llamada a una función, sentencias condicionales o iterativas).

Las sentencias han de incluirse dentro del `if` utilizando las llaves pero, si solo hay una sentencia, las llaves no son necesarias y pueden omitirse.

Desde el punto de vista de las normas de estilo (v. anexo 9.B), todas las sentencias del `if` han de estar indentadas entre `2` y `4` espacios.

Ejemplo 1:

```
#include <stdio.h>
main()
{
  int a;

  printf("Introduzca un numero entero positivo: ");
  scanf("%d%*c", &a);
  if (a%2 == 0)
    printf("El numero %d es un numero par\n", a);
}
```

Si, al ejecutar el programa anterior, el usuario introduce el número `6`, el programa muestra en pantalla el mensaje siguiente: `El numero 6 es un numero par`. Pero si, al realizar una segunda ejecución del programa, el usuario introduce el número `5`, el programa no muestra ningún mensaje en pantalla, ya que la condición `a%2 == 0` es falsa.

Obsérvese que, en el programa anterior, no se han utilizado las llaves en la sentencia `if` debido a que solo contiene una sentencia y, en este caso, las llaves pueden omitirse.

Ejemplo 2:

```
#include <stdio.h>
main()
{
  int dia, mes, anyo, fecha;

  printf("Introduzca una fecha en formato aaaammdd: ");
  scanf("%d%*c", &fecha);
  if (fecha > 0)
  {
    dia = fecha%100;
    fecha = fecha/100;
    mes = fecha%100;
```

```
    anyo = fecha/100;
    printf("La fecha es: %02d/%02d/%02d\n", dia, mes, anyo);
  }
}
```

Si, al ejecutar el programa anterior, el usuario introduce `19990308`, el programa muestra en pantalla el mensaje siguiente: `La fecha es: 08/03/1999`. Pero si, al realizar una segunda ejecución del programa, el usuario introduce la fecha siguiente `-11990312`, el programa no muestra ningún mensaje en pantalla, ya que la fecha no es mayor que `0` y la condición `fecha>0` es falsa.

Obsérvese que, a diferencia del programa del ejemplo 1, en este caso son obligatorias las llaves en la sentencia `if` debido a que contiene más de una sentencia.

La tabla 4.1 muestra un ejemplo de ejecución paso a paso del ejemplo 2. Para facilitar la explicación de la ejecución del programa, se numeran las líneas del código como sigue:

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int dia, mes, anyo, fecha;
 5:
 6:   printf("Introduzca una fecha en formato aaaammdd: ");
 7:   scanf("%d%*c", &fecha);
 8:   if (fecha > 0)
 9:   {
10:     dia = fecha%100;
11:     fecha = fecha/100;
12:     mes = fecha%100;
13:     anyo = fecha/100;
14:     printf("La fecha es: %02d/%02d/%02d\n", dia, mes, anyo);
15:   }
16: }
```

Tabla 4.1
Ejecución del programa del ejemplo 2

Línea	Valor de las variables				Entrada por teclado	Salida en pantalla
	dia	mes	anyo	fecha		
4	?	?	?	?		
6	?	?				Introduzca una fecha en formato aaaammdd:
7	?	?	?	19990308	19990308	
8 (c)	?	?	?	19990308		
10	08	?	?	19990308		
11	08	?	?	199903		
12	08	03	?	199903		
13	08	03	1999	199903		
14	08	03	1999	199903		La fecha es: 08/03/1999

La tabla anterior muestra los valores de las variables `dia`, `mes`, `anyo` y `fecha` después de ejecutar la sentencia asociada a la línea del programa indicada en la primera columna. Cuando se evalúa la condición del `if` (línea 8) se indica entre paréntesis si el resultado de la condición es cierto (`c`) o falso (`f`). El símbolo `?` indica que el valor de la variable es indeterminado. Además, en las dos últimas columnas de la tabla se muestra cuál es la entrada por teclado y cuáles son los mensajes que se muestran en pantalla.

Variante `if-else`

La variante `if-else` también es conocida como la *sentencia condicional doble*, ya que, si la condición es cierta, se ejecuta el conjunto de sentencias asociadas al `if` y, si la condición es falsa, se ejecuta el conjunto de sentencias asociadas al `else`. En este caso, *solo se ejecuta un grupo de sentencias (no ambos), dependiendo del resultado lógico de la condición*.

La sintaxis general de la sentencia `if-else` es la siguiente:

```
if (condición)
{
  sentencias a ejecutar si la condición es cierta;
}
else
{
  sentencias a ejecutar si la condición es falsa;
}
```

La figura 4.2 muestra el diagrama de flujo que describe la estructura algorítmica de esta variante condicional.

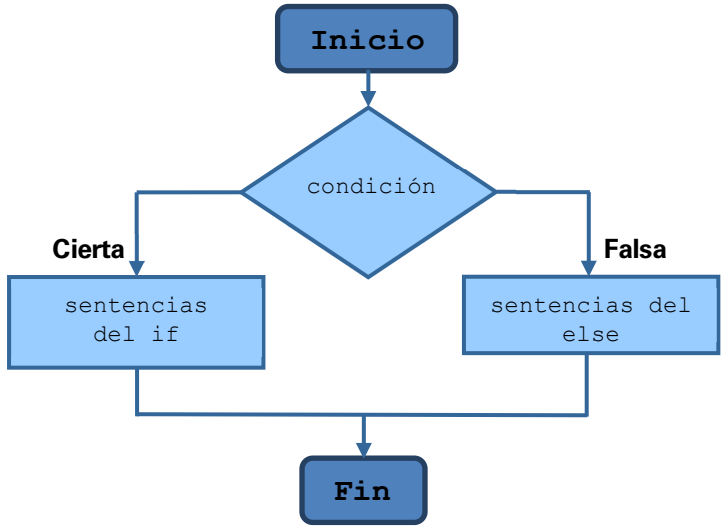


Fig. 4.2
Diagrama de flujo de la sentencia `if-else`

Al ejecutar esta sentencia, se evalúa la condición (test lógico) y, si la condición es cierta, entonces se ejecutan las sentencias del `if`. En caso contrario, se ejecutan las sentencias del `else`. Luego finaliza la ejecución de la sentencia `if-else` para cualquiera de los dos resultados de la condición.

Obsérvese que, en la estructura condicional anterior, se utiliza la parte `else` para ejecutar un grupo de sentencias cuando la condición del `if` es falsa. Si no fuera necesario ejecutar sentencias cuando la condición es falsa, utilizaríamos la variante `if`. Una mala práctica de los programadores noveles es utilizar la sentencia `if-else`, dejando la parte del `if` vacía. Por ejemplo:

```
if (condición)
{
}
else
{
  sentencias a ejecutar si la condición es falsa;
}
```

Este código se puede escribir de forma más elegante utilizando la variante `if`, como se muestra a continuación:

```
if (!condición)           /* !condición es la condición negada */
{
  sentencias a ejecutar si !condición es cierta;
}
```

Al igual que en la variante `if`, si el cuerpo del `if` o del `else` está formado por más de una sentencia, el uso de las llaves es obligatorio. Si, por el contrario, hay una única sentencia, las llaves pueden omitirse donde corresponda (en el `if`, en el `else` o en ambas partes). Además, recuérdese que las sentencias pueden ser de cualquier tipo (sentencia de asignación, llamada a una función, sentencias condicionales o iterativas).

Finalmente, desde el punto de vista de las normas de estilo (anexo 9.B), todas las sentencias están indentadas entre `2` y `4` espacios.

Ejemplo 1:

```
#include <stdio.h>
main()
{
  int a;

  printf("Introduzca un numero entero positivo: ");
  scanf("%d%*c", &a);
  if (a%2 == 0)
    printf("El numero %d es un numero par\n", a);
```

```
  else
    printf("El numero %d es un numero impar\n", a);
}
```

En el programa anterior, si el usuario introduce desde el teclado el número `7`, el programa muestra en pantalla el siguiente mensaje: `El numero 7 es un numero impar` porque la condición (`a%2 == 0`) es falsa cuando la variable `a` vale `7` y, por tanto, se ejecuta la sentencia de la parte `else`. Sin embargo, si el usuario introduce desde el teclado el valor de `10`, el programa muestra en pantalla el mensaje siguiente: `El numero 10 es un numero par` porque la condición (`a%2 == 0`) es cierta y se ejecuta la sentencia de la parte `if`.

Obsérvese, en el ejemplo 1, que no se utilizan las llaves ni en la parte `if`, ni en la parte `else`, debido a que ambas partes solo tienen una única sentencia, y entonces las llaves pueden omitirse.

Ejemplo 2:

```
#include <stdio.h>
main()
{
  int edad;

  float entrada = 6.0;
  printf("Introduzca la edad de la persona: ");
  scanf("%d%*c", &edad);
  if (edad < 18)
  {
    printf("La persona es menor de edad. ");
    entrada = 0.75*entrada;
    printf("El precio de la entrada es de %.2f euros\n",entrada);
  }
  else
    printf("La persona es mayor de edad. No tiene descuento\n");
}
```

En el programa del ejemplo anterior, si el usuario introduce desde el teclado el valor de `34`, el programa muestra en pantalla el siguiente mensaje: `La persona es mayor de edad. No tiene descuento`, ya que la condición (`edad < 18`) es falsa y, por tanto, se ejecuta la sentencia de la parte `else`. Sin embargo, si el usuario introduce desde el teclado el valor de `15`, el programa muestra en pantalla el mensaje siguiente: `La persona es menor de edad. El precio de la entrada es de 4.5 euros`, debido a que la condición (`edad < 18`) es cierta y se ejecutan las sentencias de la parte `if`.

Obsérvese que, a diferencia del ejemplo 1, en este ejemplo las llaves son obligatorias en la parte `if`, ya que hay más de una sentencia.

La tabla 4.2 muestra un ejemplo de ejecución paso a paso del ejemplo 2. Igual que antes, para facilitar la explicación de la ejecución del programa, se numeran las líneas del código como sigue:

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int edad;
 5:   float entrada = 6.0;
 6:
 7:   printf("Introduzca la edad de la persona: ");
 8:   scanf("%d%*c", &edad);
 9:
10:   if (edad < 18)
11:   {
12:     printf("La persona es menor de edad. ");
13:     entrada = 0.75*entrada;
14:     printf("El precio de la entrada es de %.2f
                euros\n",entrada);
15:   }
16:   else
17:     printf("La persona es mayor de edad.
                No tiene descuento\n");
18: }
```

La tabla siguiente muestra los valores de las variables `edad` y `entrada` después de ejecutar la sentencia asociada a la línea del programa indicada en la primera columna. El símbolo `?` indica que el valor de la variable es indeterminado. Cuando se evalúa la condición del `if` (línea 10) se indica entre paréntesis si el resultado de la condición es cierto (`c`) o falso (`f`). Además, las dos últimas columnas de la tabla muestran la entrada por teclado y los mensajes en pantalla.

Tabla 4.2
Ejecución del programa del ejemplo 2

Línea	Valor de las variables		Entrada por teclado	Salida en pantalla
	edad	entrada		
4	?			
5	?	6.0		
7	?	6.0		Introduzca la edad de la persona:
8	15	6.0	15	
10 (c)	15	6.0		
12	15	6.0		La persona es menor de edad.
13	15	4.5		
14	15	4.5		El precio de la entrada es de 4.5 euros

Variante `if-else-if`

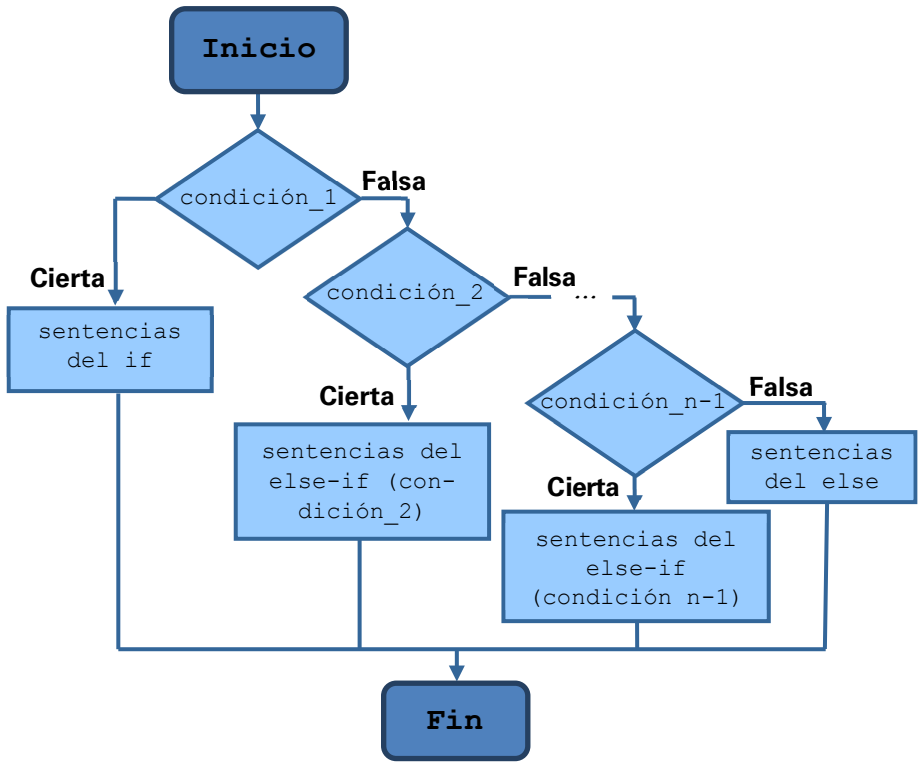
Esta variante permite implementar programas que requieren hacer selecciones múltiples. La sentencia `if-else-if` es equivalente a tener sentencias `if-else` anidadas (es decir, aquellas sentencias `if-else` que tienen otra sentencia `if-else` dentro de la parte `else`).

La sintaxis general de la sentencia `if-else-if` es la siguiente:

```
if (condición_1)
{
  sentencias a ejecutar si la condición_1 es cierta;
}
else if (condición_2)
{
  sentencias a ejecutar si la condición_2 es cierta;
}
...
else if (condición_n-1)
{
  sentencias a ejecutar si la condición_n-1 es cierta;
}
else
{
  sentencias a ejecutar si todas las condiciones anteriores
  son falsas;
}
```

La figura 4.3 muestra el diagrama de flujo correspondiente a la estructura algorítmica de la variante condicional `if-else-if`.

Fig. 4.3
Diagrama de flujo de la sentencia `if-else-if`



El flujo lógico de la sentencia `if-else-if` es de arriba hacia abajo. Por tanto, al ejecutar esta sentencia, se evalúa la primera condición (`condición_1`) y, si la condición es cierta, entonces se ejecutan las sentencias del `if`. En caso contrario, se evalúa la segunda condición (`condición_2`) y, si la condición es cierta, entonces se ejecutan las sentencias del `else-if` asociadas a esa condición. Si la `condición_2` es falsa, entonces se evalúa la tercera condición (`condición_3`), y así sucesivamente, hasta que alguna de las condiciones siguientes sea cierta o hasta que se evalúen todas las condiciones. Si todas las condiciones (desde `condición_1` hasta `condición_n-1`) son falsas, entonces se ejecutan las sentencias asociadas a la parte `else`. Finalmente, termina la ejecución de la sentencia `if-else-if`.

Igual que para las variantes condicionales anteriores, es posible prescindir de la parte `else` si no es necesario ejecutar un conjunto de sentencias cuando no se cumple ninguna de las condiciones anteriores. Además, obsérvese que se utilizan las llaves para determinar el conjunto de sentencias de cada parte de la sentencia `if-else-if`. En caso de que alguna de las partes tenga solo una sentencia, las llaves no son necesarias y pueden omitirse. Recuérdese, además, que las sentencias están indentadas entre `2` y `4` espacios.

Por otra parte, el código asociado a la sentencia `if-else-if` (indicado en el recuadro anterior) también puede reescribirse utilizando únicamente la variante `if-else` de la manera siguiente:

```
if (condición_1)
{
  sentencias a ejecutar si la condición_1 es cierta;
}
else
{
  if (condición_2)
  {
    sentencias a ejecutar si la condición_2 es cierta;
  }
  else
  {
    if (condición_3)
    {
      sentencias a ejecutar si la condición_2 es cierta;
    }
    else
    {
      ...
      if (condición_n-1)
      {
        sentencias a ejecutar si la condición_n-1 es cierta;
      }
      else
      {
        sentencias a ejecutar si todas las condiciones anteriores
        son falsas;
      }
    }
  }
}
```

Sin embargo, esta construcción se utiliza poco porque incrementa el nivel de anidaciones y de indentación en el código, lo que hace que sea menos legible y elegante si se compara con el código de la construcción `if-else-if`.

Ejemplo 1:

```
#include <stdio.h>
main()
{
  float nota;

  printf ("Introduzca su nota de FO (numero entre 0 y 10): ");
  scanf ("%f%*c", &nota);
```

```
  if (nota >= 9.0)
    printf("Tiene un sobresaliente");
  else if(nota >= 7.0)
    printf("Tiene un notable”);
  else if(nota >= 5.0)
    printf("Ha aprobado");
  else
    printf ("Ha suspendido; tiene que matricularse de FO
             de nuevo”);
}
```

En este caso, si el usuario introduce desde el teclado el valor de `5.5` como nota de FO, el programa muestra en pantalla el mensaje siguiente: `Ha aprobado`, porque las condiciones: `nota>=9.0` y `nota>=7.0` son falsas, pero la condición `nota>=5.0` es cierta. De esta forma, se ejecuta la sentencia `printf("Ha aprobado");`. Si el usuario introduce desde el teclado el valor de `8.9`, el programa muestra en pantalla el mensaje: `Tiene un notable`. Si introduce desde el teclado el valor de `9.1`, el programa muestra en pantalla el mensaje: `Tiene un sobresaliente`. Y, por último, si introduce desde el teclado el valor de `4.9`, el programa muestra en pantalla el mensaje siguiente: `Ha suspendido; tiene que matricularse de FO de nuevo`.

Ejemplo 2:

El programa siguiente calcula el volumen de una esfera o de un cilindro dependiendo del carácter introducido desde el teclado por el usuario:

```
 1: #include <stdio.h>
 2: #define PI 3.141516
 3: main()
 4: {
 5:   float vol, h, r;
 6:   char fig;
 7:
 8:   printf("CALCULO DE VOLUMENES:");
 9:   printf("Introduzca el caracter e o E para esfera o ");
10:   printf("c o C para cilindro: ");
11:   scanf("%c%*c", &fig);
12:   if((fig=='e')||(fig=='E'))
13:   {
14:     printf("Introduzca el radio de la esfera: ");
15:     scanf("%f%*c", &r);
16:
17:     vol = (4.0*PI*r*r*r)/3.0;
18:     printf("El volumen de la esfera es: %.2f\n", r, vol);
19:   }
20:   else if((fig=='c')||(fig=='C'))
```

```
21:    {
22:      printf("Introduzca el radio del cilindro: ");
23:      scanf("%f%*c", &r);
24:      printf("Introduzca la altura del cilindro: ");
25:      scanf("%f%*c", &h);
26:      vol = PI*r*r*h;
27:      printf("El volumen del cilindro es: %.2f\n",r, h, vol);
28:    }
29:    else
30:      printf("El caracter introducido no es valido\n");
31: }
```

Obsérvese que, si el usuario introduce desde el teclado `c`, `C`, `e` o `E`, el programa muestra en pantalla el volumen de la figura elegida. Si, por el contrario, el carácter introducido no es `c`, `C`, `e` o `E`, entonces el programa muestra en pantalla el mensaje: `El caracter introducido no es valido`. La tabla 4.3 muestra una ejecución paso a paso de este ejemplo.

En esta tabla, se indica cómo va cambiando el valor de cada variable (`fig`, `h`, `r` y `vol`), después de ejecutar la sentencia asociada a la línea del código indicada en la primera columna. Cuando se evalúa la condición del `if-else-if` (líneas 12 y 20) se indica entre paréntesis si el resultado de la condición es cierto (`c`) o falso (`f`). El símbolo `?` indica que el valor de la variable es indeterminado. Además, en las dos últimas columnas de la tabla, se muestra cuál es la entrada por teclado y cuáles son los mensajes que se muestran en pantalla.

Tabla 4.3
Ejecución paso a paso del programa del ejemplo 2

Línea	Valor de las variables				Entrada por teclado	Salida en pantalla
	fig	h	r	vol		
5		?	?	?		
6	?	?	?	?		
8	?	?	?	?		CALCULO DE VOLUMENES:
9	?	?	?	?		Introduzca el caracter e o E para esfera o
10	?	?	?	?		c o C para cilindro:
11	C	?	?	?	C	
12 (f)	C	?	?	?		
20 (c)	C	?	?	?		
22	C	?	?	?		Introduzca el radio del cilindro:
23	C	?	3.5	?	3.5	
24	C	?	3.5	?		Introduzca la altura del cilindro:
25	C	2.7	3.5	?	2.7	
26	C	2.7	3.5	103.91		
27						El volumen del cilindro es: 103.91

4.2 Anidaciones en las sentencias condicionales

Debido a que no es necesario utilizar las llaves cuando en alguna o en ambas partes de la variante `if-else` hay una sola sentencia, su omisión podría generar ambigüedades en la evaluación de las sentencias condicionales anidadas. Por ejemplo, si una variante `if-else` anida en una variante `if` como se muestra a continuación:

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int n, a, b, z;
 5:
 6:   n = 3;
 7:   a = -3;
 8:   b = 2;
 9:   z = 0;
10:   if (n > 0)
11:     if (a > b)
12:       z = a;
13:     else
14:       z = b;
15:
16:   printf(" z = %d\n", z);
17: }
```

entonces hemos de tener claro si la parte del `else` corresponde al primero o al segundo `if`. Por regla general, un `else` siempre se asocia al `if` más cercano que no tiene `else`. En este caso, la parte `else` se asocia al `if` más interno `(a > b)` y el programa mostraría en pantalla el mensaje `z = 2`. La tabla 4.4 ilustra un ejemplo paso a paso del programa anterior.

Tabla 4.4
Ejecución paso a paso del programa anterior

Línea	Valor de las variables				Entrada por Teclado	Salida en pantalla
	n	a	b	z		
4	?	?	?	?		
6	3	?	?	?		
7	3	-3	?	?		
8	3	-3	2	?		
9	3	-3	2	0		
10 (c)	3	-3	2	0		
11 (f)	3	-3	2	0		
14	3	-3	2	2		
16	3	-3	2	2		z = 2

Si se quiere asociar el `else` al `if` más externo (`if (n > 0)`), entonces es necesario utilizar las llaves y forzar la anidación deseada, como se muestra a continuación:

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int n, a, b, z;
 5:
 6:   n = 3;
 7:   a = -3;
 8:   b = 2;
 9:   z = 0;
10:   if (n > 0)
11:   {
12:     if (a > b)
13:       z = a;
14:   }
15:   else
16:     z = b;
17:
18:   printf("z = %d\n", z);
19: }
```

Ahora, el programa muestra en pantalla el mensaje `z = 0`. La tabla 4.5 muestra una ejecución paso a paso del programa.

Tabla 4.5
Ejecución paso a paso del programa anterior

Línea	Valor de las variables				Entrada por teclado	Salida en pantalla
	n	a	b	z		
4	?	?	?	?		
6	3	?	?	?		
7	3	-3	?	?		
8	3	-3	2	?		
9	3	-3	2	0		
10 (c)	3	-3	2	0		
12 (f)	3	-3	2	0		
18	3	-3	2	0		z = 0

Finalmente, sugerimos a los programadores noveles utilizar las llaves (aunque no sean obligatorias) en las sentencias condicionales anidadas para indicar explícitamente la asociación deseada y evitar errores de ejecución.

4.3 Errores comunes al utilizar las sentencias condicionales

A veces, los programadores principiantes cometen algunos errores al utilizar las sentencias condicionales que no son sencillos de encontrar, porque son

errores semánticos o de ejecución. En estos casos, el programador ha de utilizar el depurador como una herramienta para localizarlos.

A continuación, se indican los errores más comunes que se cometen al utilizar las sentencias condicionales.

Uso del operador de la asignación en lugar del operador relacional de igualdad en la condición del `if`

Comúnmente, los programadores principiantes confunden el operador de la asignación (=) con el operador relacional de igualdad (==) y esto produce errores semánticos de ejecución difíciles de encontrar. Por ejemplo, si consideramos el programa siguiente:

```
main()
{
  int a = 0;

  if (a == 0)
  {
    a = 2*a + 3;
  }
}
```

se ejecuta la instrucción `a = 2*a+3`, ya que la variable `a` tiene el valor `0` y, por tanto, la condición `(a == 0)` es cierta. En este caso, al finalizar la ejecución del programa, la variable `a` tendrá el valor `3`. Pero si, por error, escribimos el código siguiente (donde la expresión asociada a la condición del `if` utiliza el operador de la asignación en lugar del operador relacional de igualdad):

```
main()
{
  int a = 0;


  if (a = 0)
  {
    a = 2*a + 3;
  }
}
```

entonces la instrucción `a = 2*a+3` no se ejecuta nunca, ya que la expresión `a = 0` asigna a la variable `a` el valor `0` y evalúa como condición la expresión `a`. Como la variable `a` tiene el valor `0`, el resultado de la condición es falso. Por tanto, el programa no ejecuta la instrucción asociada a la sentencia `if`. En este caso, al finalizar la ejecución del programa, la variable `a` tendrá el valor `0`.

En este libro, la condición que se utiliza en las diferentes variantes del `if`, es siempre una expresión lógica. Sin embargo, el lenguaje C permite que esa condición sea una expresión cuyo resultado es un entero (por ejemplo, `a+1`, `3`, etc.). En este caso, si el resultado de la expresión es distinto de `0`, se interpreta como cierta. Por tanto, si en el ejemplo anterior, en lugar de utilizar en la sentencia de asignación el valor de `0` se utiliza cualquier otro valor, por ejemplo `a = 2`, la sentencia `a = 2*a+3` sí se ejecuta, ya que la variable `a` tiene un valor distinto de `0`, lo que el lenguaje C evalúa como una condición cierta. En este caso, la variable `a` tendrá el valor `7` al finalizar la ejecución del programa.

A pesar de que el lenguaje C nos permite utilizar en las condiciones expresiones con resultado entero, en este libro solamente se utilizarán expresiones lógicas para facilitar la comprensión de los códigos.

Omisión de llaves en las sentencias condicionales

La omisión de las llaves en las sentencias condicionales es un error frecuente. A continuación, se muestra un ejemplo de un programa erróneo.

Ejemplo:

```
#include <stdio.h>
main()
{
  int edad;
  float entrada = 6.0;

  printf("Introduzca la edad de la persona: ");
  scanf("%d%*c", &edad);
  if (edad < 18)
  {
    printf("La persona es menor de edad. \n");
    entrada = 0.75*entrada;
    printf("El precio de la entrada es de %.2f
            euros\n", entrada);
  }
  else
    printf("La persona es mayor de edad. \n");
    printf("No tiene descuento. Paga entrada completa\n");
}
```

En este código, no se han utilizado las llaves en la parte del `else` y, sin embargo, hay más de una sentencia.

Si el usuario introduce desde el teclado el valor `4` para la edad, este programa mostrará por pantalla los mensajes siguientes:

```
La persona es menor de edad. El precio de la entrada es de 4.5
euros
No tiene descuento. Paga entrada completa
```

Obsérvese que el último mensaje mostrado no forma parte de las sentencias del `if` pero, como no se han puesto las llaves en la parte `else`, entonces esta sentencia se encuentra fuera del `if-else`. Por esta razón, la sentencia `printf("No tiene descuento. Paga entrada completa\n");` se ejecuta siempre, independientemente de si la condición (`edad < 18`) es cierta o no. El programa sería correcto si se pusieran las llaves en la parte del `else`.

Si la omisión de las llaves hubiese ocurrido en la parte del `if`, entonces el programa no compilaría.

Uso de la variante if en lugar de la variante `if-else-if`

En la sección 4.1, se ha comentado que la anidación de sentencias `if-else` puede ser escrita con la variante `if-else-if`. Sin embargo, es muy frecuente que los programadores noveles utilicen, en lugar de la variante `if-else-if`, una secuencia de `if` consecutivos. Esta última variante puede ser correcta en códigos donde las condiciones son excluyentes, pero no es así en códigos con condiciones no excluyentes. Por ejemplo, dado el programa siguiente, que utiliza `if-else-if`:

```
#include <stdio.h>
main()
{
  float nota;

  printf ("Introduzca su nota de FO (numero entre 0 y 10): ");
  scanf ("%f%*c", &nota);
  if (nota >= 9.0)
    printf("Tiene un sobresaliente");
  else if(nota >= 7.0)
    printf("Tiene un notable");
  else if(nota >= 5.0)
    printf("Ha aprobado");
  else
    printf("Ha suspendido; tiene que matricularse de FO de
          nuevo");
}
```

muchos programadores noveles escribirían el programa anterior como sigue:

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

```
{
  float nota;

  printf("Introduzca su nota de FO (numero entre 0 y 10): ");
  scanf("%f%*c", &nota);

  if (nota >= 9.0)
    printf("Tiene un sobresaliente");
  if(nota >= 7.0)
    printf("Tiene un notable");
  if(nota >= 5.0)
    printf("Ha aprobado");
  else
    printf ("Ha suspendido; tiene que matricularse de FO de
           nuevo");
}
```

Obsérvese que, si el usuario introduce desde el teclado el valor de `7.5`, el programa con `if-else-if` muestra por pantalla el mensaje: `Tiene un notable`. Sin embargo, el programa anterior muestra por pantalla los mensajes siguientes:

```
Tiene un notable
Ha aprobado
```

Esto se debe a que las condiciones no son excluyentes, ya que se puede cumplir más de una. En este caso, `nota>=7.0` y `nota>=5.0` son ciertas.

El uso de esta alternativa no genera errores de sintaxis, pero sí introduce errores en la lógica del programa.

Orden lógico de las condiciones de la variante `if-else-if`

Finalmente, también se debe vigilar el orden lógico de las condiciones en la variante `if-else-if` cuando estas condiciones no son excluyentes, ya que puede provocar errores de ejecución.

Ejemplo:

Considérese el código anterior que utiliza la sentencia `if-else-if`, pero variando el orden lógico de las condiciones de la manera siguiente:

```
#include <stdio.h>
main()
{
  float nota;

  printf("Introduzca la nota de FO (numero entre 0 y 10): ");
```

```
  scanf("%f%*c", &nota);

  if (nota >= 5.0)
    printf("Ha aprobado");
  else if (nota >= 7.0)
    printf("Tiene un notable");
  else if (nota >= 9.0)
    printf("Tiene un sobresaliente");
  else
   printf("Ha suspendido; tiene que matricularse de FO
           de nuevo");
}
```

Obsérvese que, si el usuario introduce desde el teclado el valor `9.4`, el programa muestra por pantalla el mensaje: `Ha aprobado`, en lugar de mostrar el mensaje: `Tiene un sobresaliente`. Esto ocurre porque la primera condición del `if-else-if` (`nota >= 5.0`), es cierta para el valor `9.4`, por lo que se ejecuta la sentencia del `if`. Recuérdese que únicamente se ejecutan las sentencias del `if-else-if` asociadas a la primera condición que es cierta; las demás sentencias no se ejecutan. Por tanto, a la hora de escribir los programas, hay que ser cuidadosos y recordar el orden lógico en que se evalúan las condiciones, especialmente cuando estas no son excluyentes.

El programa anterior funcionará correctamente si se cambia el orden de las condiciones, tal como se ha mostrado en la sección anterior, o bien si se modifican las condiciones para que sean excluyentes entre ellas, tal como se muestra en el ejemplo siguiente:

```
#include <stdio.h>
main()
{
  float nota;

  printf("Introduzca la nota de FO (numero entre 0 y 10): ");
  scanf("%f%*c", &nota);

  if ((nota >= 5.0)&&(nota < 7.0))
    printf("Ha aprobado");
  else if ((nota >= 7.0)&& (nota < 9.0))
    printf("Tiene un notable");
  else if (nota >= 9.0)
    printf("Tiene un sobresaliente");
  else
   printf("Ha suspendido; tiene que matricularse de FO
           de nuevo");
}
```

Si ahora el usuario introduce desde el teclado el valor de `9.4`, el programa muestra por pantalla el mensaje: `Tiene un sobresaliente`. En este caso, la salida que aparece por pantalla es correcta. Esto ocurre porque ahora la condición del `if-else-if` que es cierta es `nota >=9.0` y todas las condiciones anteriores son falsas.

4.4 Ejemplo de uso de las sentencias condicionales

A continuación, se proporciona un ejemplo de programa donde es necesario utilizar las sentencias condicionales. Supóngase que se quiere hacer un programa en C que lea por teclado el valor numérico de un día y un mes, y muestre en pantalla un mensaje que indique la estación del año a que corresponde dicha fecha. Si el usuario introduce una fecha incorrecta, el programa muestra por pantalla el mensaje: `Error!!! Fecha incorrecta`.

Para simplificar la implementación, supóngase que todos los meses del año tienen `31` días. Además, recuérdese que la primavera comienza el `21` de marzo, el verano el `21` de junio, el otoño el `21` de septiembre y el invierno el `21` de diciembre.

Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Introduzca dia y mes (separados por un espacio): -2 3
Error!!! Fecha incorrecta
```

Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Introduzca dia y mes (separados por un espacio): 12 13
Error!!! Fecha incorrecta
```

Ejemplo de ejecución 3 (en negrita, los datos que el usuario introduce):

```
Introduzca dia y mes (separados por un espacio): 3 12
El 03/12 estamos en la estacion de otonyo
```

La implementación del programa es:

```
#include <stdio.h>
main()
{
  int dd, mm;

  printf("Introduzca dia y mes (separados por un espacio): ");
  scanf("%d %d%*c", &dd, &mm);
```

```
  if (dd >=1 && dd <= 31 && mm >= 1 && mm <= 12)
  {
    if ((mm == 3 && dd >= 21) || (mm == 6 && dd < 21) ||
        mm == 4 || mm == 5)
      printf("El %02d/%02d estamos en la estacion de
              primavera\n", dd, mm);
    else if ((mm == 6 && dd >= 21) || (mm == 9 && dd < 21) ||
              mm == 7 || mm == 8)
      printf("El %02d/%02d estamos en la estacion de
              verano\n", dd, mm);
    else if ((mm == 9 && dd >= 21) || (mm == 12 && dd < 21) ||
              mm == 10 || mm == 11)
      printf("El %02d/%02d estamos en la estacion de
               otonyo\n", dd, mm);
    else
       printf("El %02d/%02d estamos en la estacion de
               invierno\n", dd, mm);
  }
  else
    printf("Error!!! Fecha incorrecta\n");
}
```

4.5 Ejercicios

1. Indique el mensaje que muestran por pantalla los programas siguientes:

a.

```
#include <stdio.h>
main()
{
  int a = 8;

  if (a = 0)
    printf("Sentencia correcta?\n");
}
```

b.

```
#include <stdio.h>
main()
{
  unsigned int noche = 0;

  if (noche == 1)
     printf("Buenas noches\n");
  else
          printf("Buenos dias\n");
}
```

c.

```
#include <stdio.h>
main()
{
  int num = 1;

  if ((num<=-1)||(num>=1))
     printf("Rango de representacion correcto\n");
  else
     printf("Rango de representacion incorrecto\n");
}
```

2. Dado el siguiente código:

```
#include <stdio.h>
main()
{
int num;

      printf("Introduzca un numero entero: ");
scanf("%d%*c", &num);
if ((num<=-1)||(num>=1))
if (num = 0)
   printf("Rango de representacion correcto\n");
else
   printf("Rango de representacion incorrecto\n");
}
```

a. Indente correctamente el programa anterior.

b. Si el usuario introduce desde el teclado el valor `-1`, indique el mensaje que el programa muestra por pantalla.

c. Si el usuario introduce desde el teclado el valor `0`, ¿cuál es el mensaje que se muestra por pantalla?

d. Si en el programa se sustituye la condición `num = 0` por `num = 2` y el usuario introduce desde el teclado el valor `2`, ¿qué mensaje muestra el programa en pantalla?

e. Indique qué posible error se está cometiendo en la sentencia `if-else`.

3. Escriba un programa en lenguaje C que lea desde el teclado dos números enteros. Si el primer número es menor que el segundo, el programa muestra en pantalla el mensaje ''`Arriba`''. Si el primer número es mayor que el segundo, el programa muestra en pantalla el mensaje ''`Abajo`''. Si los nú-

meros son iguales, el programa muestra en pantalla el mensaje ''Igual''. Supóngase que los números leídos desde el teclado son válidos.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca dos numeros enteros separados por un espacio: 14 -89
Arriba
```

4. Escriba un programa en lenguaje C que determine, mostrando un mensaje por pantalla, si un número entero leído desde el teclado es o no un número múltiplo de 5 y de 7.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca un numero entero: -35
-35 es un numero multiplo de 5 y de 7
```

5. Escriba un programa en lenguaje C que lea del teclado un carácter y muestre un mensaje en pantalla que indique si el carácter es una vocal en letra minúscula, si el carácter es una vocal en letra mayúscula o si el carácter introducido no es una vocal.

   Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Introduzca un caracter: x
El caracter x no es una vocal
```

   Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Introduzca un caracter: A
El caracter A es una vocal en letra mayuscula
```

6.

a. Escriba un programa en lenguaje C que lea del teclado un número entero comprendido entre 0 y 6. Supóngase que cada uno de estos valores corresponde a un día de la semana, considerando que el valor 0 corresponde al día Lunes, y así sucesivamente. El programa ha de mostrar en pantalla un mensaje que indique el nombre del día de la semana correspondiente al valor introducido. Supóngase que el usuario introducirá siempre un valor entero comprendido entre 0 y 6.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca un numero entero entre 0 y 6: 2
El numero corresponde con el dia de la semana: Miercoles
```

b. Modifique el programa del apartado anterior para que, cuando el usuario introduzca desde el teclado un número menor que `0` o mayor que `6`, el programa muestre por pantalla el mensaje: `Error en el dato de entrada`.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca un numero entero entre 0 y 6: 11
Error en el dato de entrada
```

7. Escriba un programa en lenguaje C que muestre por pantalla el máximo de tres números enteros leídos desde el teclado.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca tres numeros enteros (separados por un espacio): -71 11 -2
El maximo es: 11
```

8. Dados `3` números enteros leídos desde el teclado, que representan, respectivamente, el día, el mes y el año de la fecha actual, escriba un programa en lenguaje C que muestre por pantalla la fecha del día anterior. Supóngase que la fecha introducida por el usuario es válida y, para simplificar la implementación, que no hay años bisiestos.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca la fecha actual (dia/mes/anyo): 01/03/2013
La fecha anterior a 01/03/2013 es: 28/02/2013
```

9. El ejemplo de uso de la sección 4.4 determina la estación del año a la que pertenece una fecha leída desde el teclado. Modifique el ejemplo anterior para que ahora no tenga que suponerse que todos los meses del año tienen `31` días. Considérese que el mes de febrero puede tener hasta `29` días.

Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Introduzca dia y mes (separados por un espacio): -2 3
Error!!! Fecha incorrecta
```

Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Introduzca dia y mes (separados por un espacio): 31 4
Error!!! El mes 4 no tiene 31 dias
```

Ejemplo de ejecución 3 (en negrita, los datos que el usuario introduce):

```
Introduzca dia y mes (separados por un espacio): 1 07
El 01/07 estamos en la estacion de verano
```

4.6 Respuesta a los ejercicios propuestos

1.

a. El programa no muestra ningún mensaje en pantalla porque la condición de la sentencia `if (a=0)` no es una expresión lógica sino una expresión con resultado entero igual a `0`, por lo que se interpretará como falso y la sentencia del `if` no se ejecuta.

b.
```
Buenos dias
```

c.
```
Rango de representacion correcto
```

2.

a.
```
#include <stdio.h>
main()
{
  int num;

  printf("Introduzca un numero entero: ");
  scanf("%d%*c", &num);
  if ((num<=-1)||(num>=1))
    if (num = 0)
      printf("Rango de representacion correcto\n");
    else
      printf("Rango de representacion incorrecto\n");
}
```

b.
```
Rango de representacion incorrecto
```

c. No muestra nada por pantalla.

d.
```
Rango de representacion correcto
```

e. El error cometido ha sido utilizar el operador de la asignación en lugar del operador relacional de igualdad en la condición del `if`.

3.

```
#include <stdio.h>
main()
{
  int num1, num2;

  printf("Introduzca dos numeros enteros separados por un
          espacio: ");
  scanf("%d %d%*c", &num1, &num2);

  if (num1 < num2)
```

```
    printf("Arriba\n");
  else if (num1 > num2)
    printf("Abajo\n");
  else
    printf("Igual\n");
}
```

4.

```
#include <stdio.h>
main()
{
  int num;

  printf("Introduzca un numero entero: ");
  scanf("%d%*c", &num);

  if ((num%5 == 0) && (num%7 == 0))
    printf("%d es un numero multiplo de 5 y de 7\n", num);
  else
    printf("%d no es un numero multiplo de 5 y de 7\n",
            num);
}
```

5.

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

  printf("Introduzca un caracter: ");
  scanf("%c%*c", &letra);

  if ((letra == 'a')||(letra == 'e')||(letra == 'i')||
      (letra == 'o')||(letra == 'u'))

    printf("El caracter %c es una vocal en letra
            minuscula\n", letra);

  else if((letra == 'A')||(letra == 'E')||(letra == 'I')||
          (letra == 'O')||(letra == 'U'))

    printf("El caracter %c es una vocal en letra
            mayuscula\n", letra);

  else
    printf("El caracter %c no es una vocal\n", letra);
}
```

6.

a.

```
#include <stdio.h>
main()
{
  int num;

  printf("Introduzca un valor entero entre 0 y 6: ");
  scanf("%d%*c", &num);

  printf("El numero corresponde con el dia de la
          semana: ");
  if (num == 0)
    printf("Lunes\n");
  else if (num == 1)
    printf("Martes\n");
  else if (num == 2)
    printf("Miercoles\n");
  else if (num == 3)
    printf("Jueves\n");
  else if (num == 4)
    printf("Viernes\n");
  else if (num == 5)
    printf("Sabado\n");
  else
   printf("Domingo\n");
}
```

b.

```
#include <stdio.h>
 main()
 {
   int num;

   printf("Introduzca un valor entero entre 0 y 6: ");
   scanf("%d%*c", &num);
   if ((num < 0)||(num > 6))
     printf("Error en el dato de entrada\n");
   else
   {
     printf("El numero corresponde con el dia de la
             semana: ");
     if (num == 0)
       printf("Lunes\n");
     else if (num == 1)
       printf("Martes\n");
     else if (num == 2)
```

```
            printf("Miercoles\n");
          else if (num == 3)
            printf("Jueves\n");
          else if (num == 4)
            printf("Viernes\n");
          else if (num == 5)
            printf("Sabado\n");
          else
            printf("Domingo\n");
        }
    }
```

7.

```
#include <stdio.h>
main()
{
  int num1, num2, num3;

  printf("Introduzca tres numeros enteros (separados
          por un espacio): ");
  scanf("%d %d %d%*c", &num1, &num2, &num3);

  printf("El maximo es: ");
  if ((num1 >= num2) && (num1 >= num3))
    printf("%d\n", num1);
  else if ((num2 >= num1) && (num2 >= num3))
    printf("%d\n", num2);
  else
    printf("%d\n", num3);
}
```

8.

```
#include <stdio.h>
main()
{
  int dd, mm, aa;

  printf("Introduzca la fecha actual (dia/mes/anyo): ");
  scanf("%d/%d/%d%*c", &dd, &mm, &aa);
  printf("La fecha anterior a %02d/%02d/%d es: ",
          dd, mm, aa);

  if (dd != 1)
    printf("%02d/%02d/%d\n", dd-1, mm, aa);
  else
  {
```

```
    if (mm == 1)
      printf("%d/%d/%d\n", 31, 12, aa-1);
    else if((mm == 2)||(mm == 4)||(mm == 6) ||
            (mm == 9) || (mm == 11))
         printf("%d/%02d/%d\n", 31, mm-1, aa);
    else if((mm == 5)||(mm == 7)||(mm == 8) ||
            (mm == 10) || (mm == 12))
      printf("%d/%02d/%d\n", 30, mm-1, aa);
    else
         printf("%d/%02d/%d\n", 28, 02, aa);
  }
}
```

9.

```
main()
{
  int dd, mm;

  printf("Introduzca dia y mes (separados por un
          espacio): ");
  scanf("%d %d%*c", &dd, &mm);

  if (dd >=1 && dd <= 31 && mm >= 1 && mm <= 12)
  {
    if ((mm == 2 && dd >29) || ((mm == 4 || mm == 6 ||
         mm == 9 || mm == 11) && dd > 30))
      printf("Error!!! El mes %02d no tiene %02d dias\n",
              mm, dd);
    else
    {
      if ((mm == 3 && dd >= 21) || (mm == 6 && dd < 21) ||
           mm == 4 || mm == 5)
        printf("El %02d/%02d estamos en la estacion de
                primavera\n", dd, mm);
      else if ((mm == 6 && dd >= 21) ||
               (mm == 9 && dd < 21) ||
                mm == 7 || mm == 8)
        printf("El %02d/%02d estamos en la estacion de
                verano\n", dd, mm);
      else if ((mm == 9 && dd >= 21) ||
               (mm == 12 && dd < 21) ||
                mm == 10 || mm == 11)
        printf("El %02d/%02d estamos en la estacion de
                otonyo\n", dd, mm);
      else
        printf("El %02d/%02d estamos en la estacion de
                invierno\n", dd, mm);
    }
```

```
  }
  else
    printf("Error!!! Fecha incorrecta\n");
}
```

→5

Sentencias iterativas

La necesidad de repetir las mismas sentencias numerosas veces en muchos códigos hace necesario utilizar las sentencias iterativas. Estas sentencias, también llamadas *bucles*, controlan la repetición de un conjunto de instrucciones mediante la evaluación de una condición o mediante un contador. Las sentencias que se repiten en el bucle se denominan *bloque* o *cuerpo del bucle*.

El lenguaje C ofrece tres tipos de sentencias iterativas: `for`, `while` y `do-while`. En este libro, solo se analizan las dos primeras. En este capítulo, primero se muestra la sintaxis de las sentencias iterativas, junto con algunos ejemplos. A continuación se comenta la equivalencia entre la sentencia `for` y la sentencia `while`, así como los errores más comunes al utilizar estas sentencias. Luego, se proporcionan algunos ejemplos de anidaciones de sentencias iterativas y, por último, se muestra un ejemplo de uso de las mismas.

5.1 Sentencias iterativas

Un programa puede requerir que un mismo grupo de sentencias se ejecute repetidamente hasta que se satisfaga una condición. Cada repetición del cuerpo del bucle se denomina *iteración*. En algunos casos, la cantidad de iteraciones se conoce con exactitud. Un ejemplo es determinar la suma de los `5` primeros números naturales, que requiere realizar `5` sumas. Sin embargo, otras veces el número de iteraciones depende de que una determinada condición lógica sea cierta, lo cual impide conocer a priori el número de iteraciones. Un ejemplo de esto sería determinar la suma de una secuencia de números leída desde el teclado y finalizada en `-1`. En este caso, no se sabe la cantidad exacta de números que tendrá la secuencia y, por tanto, no se puede determinar la cantidad exacta de sumas.

El lenguaje C ofrece dos tipos de sentencias iterativas, el `for` y el `while`. Estas sentencias se pueden utilizar indistintamente para implementar cualquier bucle. Sin embargo, en este libro se utiliza el `for` cuando se conoce exactamente el número de iteraciones y el `while` cuando no se conoce. Esto ayudará a adquirir un buen estilo de programación. A continuación, se analizan en detalle estas dos sentencias.

Sentencia `for`

La sentencia iterativa `for` se utiliza cuando se conoce exactamente la cantidad de iteraciones del bucle.

La sintaxis general de la sentencia `for` es la siguiente:

```
for(inicialización; comparación; actualización)
{
  sentencias a ejecutar si la comparación es cierta;
}
```

Obsérvese que la sintaxis de esta sentencia tiene una estructura de control formada por tres partes:

1. La primera de ellas, llamada *inicialización*, especifica el valor inicial de una variable de control que controla la repetición del bucle. Esta parte está formada por una sentencia de asignación, que se ejecuta solo una vez y al inicio del bucle.
2. La segunda parte de la estructura de control, llamada *comparación*, representa una condición (tal como se define en el capítulo 2), que ha de ser satisfecha para ejecutar las instrucciones del cuerpo del bucle. La comparación está formada por una expresión lógica sobre la variable de control del bucle que determina la cantidad de repeticiones. Si esta condición es cierta, se ejecutan las sentencias del cuerpo del bucle y, si es falsa, el bucle finaliza.
3. Por último, la tercera parte, llamada *actualización*, modifica el valor de la variable de control después de cada ejecución del cuerpo del bucle. La actualización generalmente está formada por sentencias que incrementan o decrementan en un valor determinado la variable de control del bucle.

La figura 5.1 muestra el diagrama de flujo correspondiente a la estructura algorítmica de la sentencia `for`.

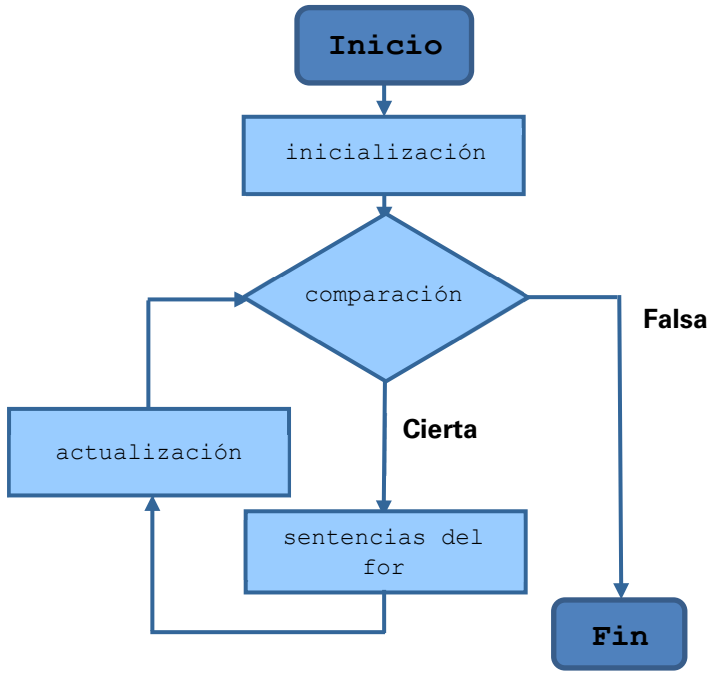


Fig. 5.1
Diagrama de flujo de la sentencia iterativa `for`

En este diagrama, obsérvese que al ejecutarse la sentencia iterativa se realiza (una sola vez) la *inicialización* de la variable de control del `for`. Luego, se evalúa la condición (test lógico) de la *comparación* y, si la condición es falsa, entonces el bucle finaliza. En caso contrario, se ejecutan las sentencias del `for` y se realiza la *actualización* correspondiente de la variable de control para evaluar nuevamente la *comparación* y seguir iterando.

Al igual que las sentencias condicionales, si el cuerpo del `for` tiene más de una sentencia, es obligatorio utilizar las llaves. Si, por el contrario, el cuerpo del bucle tiene solo una sentencia, las llaves pueden omitirse. Además, el tipo de sentencia puede ser de cualquier tipo (asignación, llamada a una función, sentencia condicional o iterativa).

Desde el punto de vista de las normas de estilo (v. anexo 9.B), las sentencias están indentadas entre `2` y `4` espacios.

Ejemplo:

El programa siguiente muestra por pantalla la suma de los `5` primeros números naturales.

```
#include <stdio.h>
main()
{
  int i, suma=0;
```

```
  for(i=1; i<=5; i=i+1)
    suma = suma + i;

  printf("La suma de los 5 primeros naturales es %d\n", suma);
}
```

En la sentencia iterativa `for` de este ejemplo, obsérvese que la variable `i` es la variable de control del bucle, `i=1` es la *inicialización*, `i<=5` es la *comparación* e `i=i+1` es la *actualización*. El bucle realiza `5` iteraciones. El programa muestra en pantalla el mensaje: `La suma de los 5 primeros naturales es 15`.

La tabla 5.1 muestra un ejemplo de ejecución paso a paso del programa anterior. Para facilitar la explicación de la ejecución del programa, se numeran las líneas del código como sigue:

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int i, suma=0;
 5:
 6:   for(i=1; i<=5; i=i+1)
 7:     suma = suma + i;
 8:
 9:   printf("La suma de los 5 primeros naturales es %d\n", suma);
10: }
```

Tabla 5.1
Ejecución del programa del ejemplo

Línea	Valor de las variables		Entrada por teclado	Salida en pantalla
	i	suma		
4	?	0		
6 (c)	1	?		
7	1	1		
6 (c)	2	1		
7	2	3		
6 (c)	3	3		
7	3	6		
6 (c)	4	6		
7	4	10		
6 (c)	5	10		
7	5	15		
6 (f)	6	15		
9	6	15		La suma de los 5 primeros naturales es 15

La tabla anterior muestra los valores de las variables `i` y `suma` después de ejecutar la sentencia indicada en la primera columna. Cuando se evalúa la comparación del for (línea 6) se indica entre paréntesis si esta es cierta (`c`) o falsa (`f`).

El símbolo `?` indica que el valor de la variable es indeterminado. Además, en la última columna, se indican los mensajes que se muestran en pantalla.

Por otra parte, se quiere comentar que el lenguaje C permite omitir alguna o todas las partes de la estructura de control de la sentencia `for`. En este caso, seguirá siendo necesario colocar punto y coma para separar cada una de las partes. Por ejemplo, si se quiere omitir la *inicialización*, entonces la estructura de control de la sentencia `for` es la siguiente:

```
for(; comparación; actualización)
{
  sentencias a ejecutar si la comparación es cierta;
}
```

Además, el lenguaje C también permite incluir más de una sentencia de asignación, tanto en la *inicialización* como en la *actualización.* En este caso, las sentencias de asignación de cada parte se separan con comas. Por ejemplo:

```
for(i=0, j=0; i<5; i=i+1, j=j-1)
{
  sentencias a ejecutar si la comparación es cierta;
}
```

Sin embargo, y a pesar de que el lenguaje C lo permite, en este libro no se omite ninguna de las partes en la estructura de control de la sentencia `for` ni se incluye más de una sentencia de asignación en la *inicialización* y/o la *actualización*, con el objetivo de mantener la claridad en los programas implementados. Aquí se utilizará el `for`, incluyendo siempre las tres partes de la estructura de control (*inicialización*, *comparación* y *actualización*) y cuando se conozca exactamente el número de iteraciones a realizar.

Sentencia `while`

La sentencia iterativa `while` se utiliza cuando no se conoce con exactitud la cantidad de iteraciones del bucle.

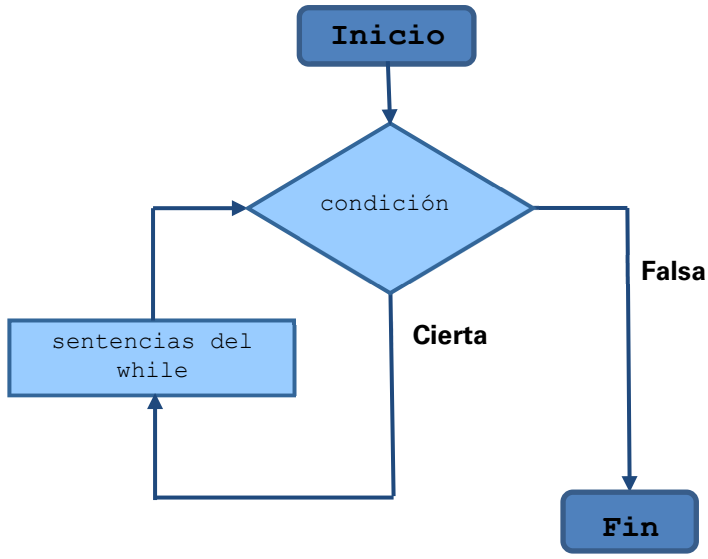
La sintaxis general de la sentencia `while` es la siguiente:

```
while(condición)
{
  sentencias a ejecutar si la condición es cierta;
}
```

La ejecución de una sentencia `while` consiste en evaluar primero la condición. Si es cierta, se ejecuta el conjunto de sentencias del bucle y se vuelve a evaluar la condición. Este proceso se repite hasta que la condición es falsa. En ese momento, finaliza la ejecución del bucle. Debe tenerse en cuenta que alguna de las sentencias del bucle ha de modificar alguna de las variables que apare-

cen en la condición. Si no es así, el bucle `while` no acabará nunca (bucle infinito). La figura 5.2 muestra el diagrama de flujo que describe la estructura algorítmica de la sentencia `while`.

Fig. 5.2
Diagrama de flujo de la sentencia iterativa `while`



Igual que para las sentencias condicionales y el bucle `for`, si el cuerpo del `while` tiene más de una sentencia, es obligatorio utilizar las llaves. Si, por el contrario, el cuerpo del bucle tiene solo una sentencia, las llaves pueden omitirse. Además, el tipo de sentencia puede ser de cualquier tipo (asignación, llamada a una función, sentencia condicional o iterativa). Finalmente, y como norma de estilo (v. anexo 9.B), las sentencias están indentadas entre `2` y `4` espacios.

Ejemplo:

El programa siguiente muestra por pantalla la cantidad total de caracteres de una secuencia leída desde el teclado y finalizada en el carácter punto (`'.'`). La cantidad total de caracteres no incluye el carácter punto.

```
#include <stdio.h>
main()
{
  int carac=0;
  char letra;
  printf("Introduzca secuencia de caracteres finalizada
          en punto : ");
  scanf("%c", &letra);
  while(letra != '.')
  {
    carac = carac + 1;
    scanf("%c", &letra);
  }
```

```
  scanf("%*c");   /* Lee el carácter \n */
  printf("La cantidad de caracteres de la secuencia es:
         %d\n", carac);
}
```

A diferencia del ejemplo que se ha mostrado para la sentencia `for`, en este caso no se sabe *a priori* la cantidad de caracteres de la secuencia que el usuario introducirá por teclado, por lo que no es posible determinar la cantidad de iteraciones del bucle. En este caso, se utiliza el bucle `while` para implementar el programa.

Obsérvese que en el `while` la sentencia `scanf("%c", &letra);` modifica el valor de la variable `letra` en cada iteración y también la condición del `while`. Esto garantiza que el `while` no sea un bucle infinito y, en este caso, finalizará cuando la variable `letra` sea igual al carácter punto.

La tabla 5.2 muestra un ejemplo de ejecución paso a paso del programa anterior. Para facilitar la explicación de la ejecución del programa, las líneas del código se numeran como sigue:

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int carac=0;
 5:   char letra;
 6:
 7:   printf("Introduzca secuencia de caracteres
             finalizada en punto : ");
 8:   scanf("%c", &letra);
 9:   while(letra != '.')
10:   {
11:     carac = carac + 1;
12:     scanf("%c", &letra);
13:   }
14:   scanf("%*c");
15:   printf("La cantidad de caracteres de la secuencia es:
             %d\n", carac);
16: }
```

La tabla muestra los valores de las variables `letra` y `carac` después de ejecutar la sentencia indicada en la primera columna. Cuando se evalúa la condición del `while` (línea 9) se indica entre paréntesis si esta condición es cierta `(c)` o falsa `(f)`. El símbolo `?` indica que el valor de la variable es indeterminado. Además, en las dos últimas columnas, se muestran la entrada por teclado y los mensajes en pantalla.

Tabla 5.2
Ejecución del programa del ejemplo

Línea	Valor de las variables		Entrada por teclado	Salida en pantalla
	letra	carac		
4	?	0		
5	?	0		
7	?	0		Introduzca secuencia de caracteres finalizada en punto :
8	B	0	Bucles.	
9 (c)	B	0		
11	B	1		
12	u	1		
9 (c)	u	1		
11	u	2		
12	c	2		
9 (c)	c	2		
11	c	3		
12	l	3		
9 (c)	l	3		
11	l	4		
12	e	4		
9 (c)	e	4		
11	e	5		
12	s	5		
9 (c)	s	5		
11	s	6		
12	.	6		
9 (f)	.	6		
14	.	6		
15				La cantidad de caracteres de la secuencia es: 6

5.2 Equivalencia entre las sentencias `for` y `while`

Las sentencias iterativas `for` y `while` son equivalentes. Esto implica que todo lo que se implementa utilizando la sentencia `for` puede implementarse también utilizando la sentencia `while`. Sin embargo, a pesar de que las dos sentencias iterativas son equivalentes, aquí se utilizará el `for` cuando se conozca exactamente la cantidad de iteraciones del bucle y el `while` en caso contrario.

La equivalencia entre las sentencias `for` y `while` se muestra en la figura 5.3. A continuación se verán dos ejemplos: El ejemplo 1 explica cómo obtener el código equivalente de un programa implementado con un `for`, utilizando la sentencia `while`. El ejemplo 2 muestra justo lo contrario, cómo obtener el código equivalente de un programa implementado con un `while`, usando el `for`.

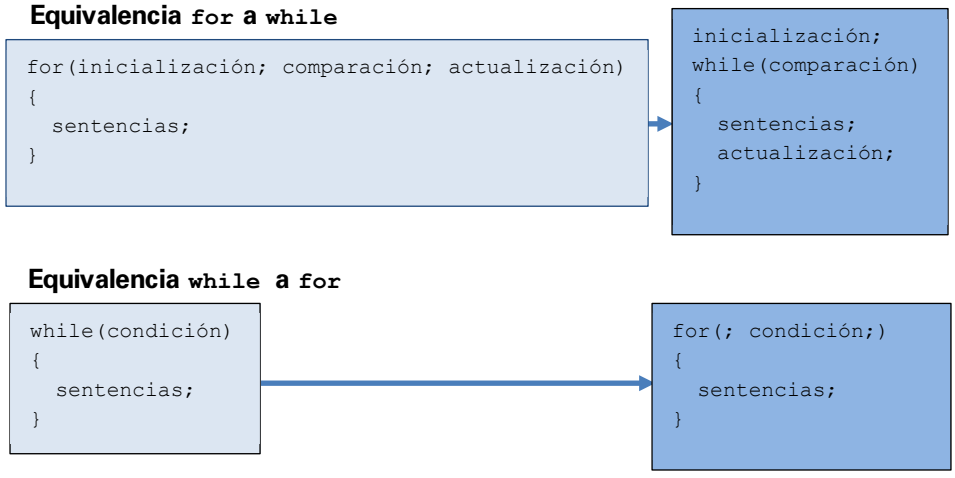


Fig. 5.3 Equivalencia `for`-`while` y `while`-`for`

Ejemplo 1:

El programa siguiente muestra en pantalla los números comprendidos entre `1` y `12` en orden inverso: `12 11 10 9 8 7 6 5 4 3 2 1`

```
#include <stdio.h>
main()
{
  int i;

  for(i=12; i>=1; i=i-1)
    printf("%d ",i);
  printf("\n");
}
```

que es equivalente a:

```
#include <stdio.h>
main()
{
  int i;

  i = 12;
  while (i>=1)
  {
    printf("%d ",i);
    i = i-1;
  }
  printf("\n");
}
```

Obsérvese que la *inicialización* de la sentencia `for` se ha colocado justo antes de la sentencia `while`, la *comparación* es ahora la condición del `while` y la *actualización* es la última sentencia del cuerpo del `while`.

Ejemplo 2:

El programa siguiente muestra en pantalla la suma de los `20` primeros números naturales múltiplos de `3`.

```
#include <stdio.h>
main()
{
  int elem=0, cont, suma=0;

  cont = 0;
  while(cont<20)
  {
    if (elem%3 == 0)
    {
      suma = suma + elem;
      cont = cont + 1;
    }
    elem = elem + 1;
  }
  printf("La suma de los 20 primeros numeros multiplos
          de 3 es %d\n", suma);
}
```

que es también equivalente a:

```
#include <stdio.h>
main()
{
  int elem=0, cont, suma=0, i;

  cont = 0;
  for(; cont<20;)
  {
    if (elem%3 == 0)
    {
      suma = suma + elem;
      cont = cont + 1;
    }
    elem = elem+1;
  }
  printf("La suma de los 20 primeros numeros multiplos
          de 3 es %d\n", suma);
}
```

Obsérvese que la estructura de control de la sentencia `for` no contiene todas las partes. En este caso, se han omitido la *inicialización* y la *actualización*. Solo aparece la *comparación*, que es la condición del `while`. Recuérdese, tal como se ha comentado en la sección 5.1, que en este libro no se omitirá ninguna de las partes de la estructura de control de la sentencia `for` para darle una mayor claridad a los códigos. Por tanto, en este ejemplo se considera más clara y elegante la implementación con la sentencia `while` que la de la sentencia `for`.

Otro código alternativo es:

```
#include <stdio.h>
main()
{
  int elem=0, cont, suma=0, i;

  for(cont=0; cont<20; elem=elem+1)
  {
    if (elem%3 == 0)
    {
      suma = suma + elem;
      cont = cont + 1;
    }
  }
  printf("La suma de los 20 primeros numeros multiplos
          de 3 es %d\n", suma);
}
```

En este caso, se ha utilizado la sentencia anterior al `while` (`cont=0;`) y la última sentencia del `while` (`elem = elem+1;`) como *inicialización* y *actualización* del `for`, respectivamente. De esta manera, no se omite ninguna de las partes de la estructura de control del `for`, pero obsérvese ahora que la *actualización* no depende de la variable de control `cont`, sino de la variable `elem`. Forzar cálculos sin relación en la estructura de control del `for` se considera un mal estilo de programación. Por ello, a pesar de que el bucle `for` obtenido es equivalente al bucle `while`, en este libro se evita tener en la estructura de control del `for` otras variables que no sean la variable de control del bucle.

En resumen, a pesar de que existe una equivalencia entre ambas sentencias, en este libro solo se utiliza la sentencia `for` cuando se conoce a priori la cantidad de iteraciones del bucle, y se utiliza la sentencia `while` en caso contrario.

Finalmente, cabe añadir que, cuando en la actualización del `for` aparecen las sentencias `i=i+1` o `i=i-1`, estas pueden sustituirse por las sentencias: `i++` y `i--`, respectivamente. Los operadores monarios `++` y `--` son los operadores de incremento y decremento. Se dice que estos operadores son monarios porque tienen un solo operando. El orden de precedencia de estos operadores puede consultarse en el anexo 9.A. En este libro, solo se utilizan estos operadores, si así se requiere, en la actualización del `for`.

5.3 Errores comunes al utilizar las sentencias iterativas

A veces, los programadores noveles cometen algunos errores al utilizar las sentencias iterativas. Estos errores no son sencillos de encontrar ya que incorporan errores semánticos en los programas que el compilador no detecta. Una herramienta que permite detectar estos errores es el depurador.

A continuación, se muestran los errores más comunes que se cometen al utilizar las sentencias iterativas.

Cuerpo del bucle vacío

El cuerpo del bucle puede estar vacío (no contener ninguna sentencia). Esto ocurre al colocar el carácter punto y coma al final de la sentencia iterativa, tal como se muestra a continuación:

```
for(inicialización; comparación; actualización);

while(condición);
```

El carácter punto y coma al final de la estructura de control del `for` o de la condición del `while` indica que esa sentencia iterativa no tiene cuerpo, aunque inmediatamente después se escriban un conjunto de sentencias. Este conjunto de sentencias no están dentro del cuerpo del `for` o del `while`, por lo que se ejecutarán una sola vez cuando finalice el bucle.

Ejemplo de bucle vacío utilizando la sentencia `for`:

```
#include <stdio.h>
main()
{
  int i;

  for(i=12; i>=1; i--);
    printf("%d ",i);
}
```

En este caso, el programa realiza `12` veces la actualización del índice `i` hasta que `i` es igual a cero y sin ejecutar ninguna otra sentencia. Cuando `i` vale `0`, la condición del `for` no se cumple y, por tanto, el bucle finaliza. A continuación se realiza (una sola vez) la sentencia `printf`. La sentencia `printf` está fuera del `for` porque se ha puesto el carácter punto y coma al final de la estructura de control. El programa anterior muestra en pantalla el mensaje siguiente: `0`.

Ejemplo de bucle vacío e infinito utilizando la sentencia `while`:

```
#include <stdio.h>
main()
{
  int num = -1;

  while(num!=0);
    scanf("%d ",&num);
}
```

Al igual que en el caso anterior, el cuerpo de la sentencia del `while` está vacío, porque aparece el carácter punto y coma después de la condición. Sin embargo, en este caso, el programa no acaba nunca, ya que la variable `num`, inicializada antes de entrar en el bucle con el valor de `-1`, no se actualiza y siempre

tiene el valor de `-1`, por lo que la condición es siempre cierta y el bucle no finaliza nunca. En este caso, el programa no muestra ningún mensaje por pantalla, ya que nunca termina de ejecutar la sentencia `while`.

El error de bucle vacío o infinito no es detectado por el compilador y supone un error de lógica en la programación.

Bucles que no finalizan

En el ejemplo anterior, ya se ha mostrado un ejemplo de un bucle `while` que no finalizaba por el uso del carácter punto y coma después de escribir la condición del bucle. Este mismo error puede ocurrir en una sentencia `for` cuando la *comparación* o la *actualización* (en su estructura de control) no son correctas.

Ejemplo de bucle que no finaliza utilizando la sentencia for:

```
#include <stdio.h>
main()
{
  int i;

  for(i=12; i>=12; i++)
    printf("%d ",i);
}
```

En este caso, la variable de control del bucle (variable `i`) es inicializada con el valor `12` y en cada iteración se va incrementando en `1`. De esta forma, la condición `i>=12` siempre es cierta y el bucle no finaliza nunca. El programa muestra por pantalla: `12 13 14 15 16 17 18 19 20 ...`

Otro ejemplo de un programa con bucle que no finaliza es:

```
#include <stdio.h>
main()
{
  int i;

  for(i=1; i=12; i++)
    printf("%d ",i);
}
```

Obsérvese que, en este caso, se ha utilizado el operador de la asignación (`=`) en la *comparación* del `for` (en lugar de un operador relacional). Este operador primero asigna el valor `12` a la variable `i` y luego evalúa el valor de `i`. En este caso, como el valor de `i` es distinto de `0`, el lenguaje C evalúa esta expresión como cierta y hace que la comparación del bucle siempre sea cierta, por lo que el bucle no finaliza nunca. El programa muestra por pantalla: `12 12 12 12 12 12 12 12 12` ...

Bucles que no ejecutan el cuerpo del bucle

El cuerpo de un bucle no se ejecuta si la *condición* o la *comparación* en la estructura de control es falsa desde el inicio.

Ejemplo de bucle que no ejecuta el cuerpo del bucle `for`*:*

```
#include <stdio.h>
main()
{
  int i;

  for(i=0; i>=12; i++)
    printf("%d ",i);
}
```

En este caso, la variable de control del bucle (variable `i`) es inicializada con el valor `0` y la condición (`i>=12`) es falsa desde el inicio, por lo que el bucle finaliza sin ejecutar la sentencia `printf`. No se ejecuta ninguna iteración del bucle. El programa no muestra ningún mensaje en pantalla.

Bucles que ejecutan una iteración de más o de menos

En ocasiones, los programas son incorrectos porque las sentencias iterativas ejecutan alguna iteración de más o de menos. Por ello, se debe ser muy cuidadoso a la hora de decidir la *condición* o la *comparación* de las sentencias `while` y `for` porque estas determinan el número de iteraciones.

Ejemplo de bucle con una iteración más:

```
#include <stdio.h>
main()
{
  int base, expo, i;
  float pot;

  printf("Introduzca la base y el exponente (separados por
          un espacio): ");
  scanf("%d %d%*c", &base, &expo);
  pot = 1.0;
  for(i=0; i<=expo; i++)
    pot = pot*base;

  printf ("La potencia de %d elevado a %d es %.2f\n", base,
           expo, pot);
}
```

El programa anterior pretende calcular la potencia base$^{\text{exponente}}$ cuando el exponente es superior o igual a cero. Obsérvese que el programa realiza una multiplicación adicional de la base, ya que la cantidad de iteraciones del `for` es de `expo+1`.

A continuación, se muestran dos alternativas de la estructura de control del `for` que solucionan este problema:

`for(i=1; i<=expo; i++)` o `for(i=0; i<expo; i++)`.

Utilizando cualquiera de estas alternativas, la potencia se calcula correctamente ya que la cantidad de iteraciones del bucle es igual a `expo`.

Una fórmula general (por casos) para calcular la cantidad de iteraciones que realiza un bucle `for` es:

- Si `for(i=limite_inferior; i<=limite_superior; i++)`, la cantidad de iteraciones es: `limite_superior - limite_inferior + 1`
- Si `for(i=limite_inferior; i<limite_superior; i++)`, la cantidad de iteraciones es: `(limite_superior - 1) - limite_inferior + 1`
- Si `for(i=limite_superior; i>=limite_inferior; i--)`, la cantidad de iteraciones es: `limite_superior - limite_inferior + 1`
- Si `for(i=limite_superior; i>limite_inferior; i--)`, la cantidad de iteraciones es: `limite_superior - (limite_inferior + 1) + 1`

5.4 Anidaciones en sentencias iterativas

Los bucles, al igual que las sentencias condicionales, pueden anidar otras sentencias. A continuación, se proporcionan tres ejemplos de programas: el primero muestra una sentencia iterativa que anida una sentencia condicional, mientras que el segundo y el tercer ejemplo muestran sentencias iterativas que anidan otra sentencia iterativa.

Ejemplo 1: Sentencia `for` con sentencia `if-else` anidada

El programa siguiente calcula el valor máximo de la función `f`, definida como $f(x)=x^2-3x+5$, con `x` en el intervalo `[-2,1]`. El programa muestra en pantalla el valor máximo de la función y el valor de `x` en que se alcanza dicho valor.

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int i, max, max_x, aux;
 5:
 6:   max_x = -2;
 7:   max = (max_x)*(max_x)-3*(max_x)+5;
 8:   for(i=-1; i<=1; i++)
 9:   {
10:     aux = i*i-3*i+5;
11:     if(aux>max)
12:     {
13:       max = aux;
14:       max_x = i;
```

```
15:      }
16:   }
17:   printf("El valor maximo de la funcion es %d para x=%d\n",
              max, max_x);
18: }
```

La tabla 5.3 muestra un ejemplo de ejecución paso a paso del programa anterior.

Tabla 5.3
Ejecución del programa del ejemplo 1

Línea	Valor de las variables				Salida en pantalla
	i	max	aux	max_x	
4	?	?	?	?	
6	?	?	?	-2	
7	?	15	?	-2	
8 (c)	-1	15	?	-2	
10	-1	15	9	-2	
11 (f)	-1	15	9	-2	
8 (c)	0	15	9	-2	
10	0	15	5	-2	
11 (f)	0	15	5	-2	
8 (c)	1	15	5	-2	
10	1	15	3	-2	
11 (f)	1	15	3	-2	
8 (f)	2	15	3	-2	
17	2	15	3	-2	El valor maximo de la funcion es 15 para x=-2

Ejemplo 2: Sentencias `for` anidadas

El programa siguiente calcula y muestra en pantalla el valor medio de los elementos de varias listas de números enteros. La cantidad total de listas, así como el número de elementos por lista y los valores de sus elementos (valores enteros), son introducidos desde el teclado. Supóngase que al menos hay una lista que contiene, como mínimo, un elemento.

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int nlis, nele, tele, ele, i, j;
 5:   float prom = 0.0;
 6:
 7:   tele = 0;
 8:   printf("Numero de listas: ");
 9:   scanf("%d%*c",&nlis);
10:
11:   for(i=1; i<=nlis; i++)
12:   {
13:     printf("Cantidad de elementos de la lista %d: ", i);
14:     scanf("%d%*c",&nele);
15:     tele = tele + nele;
16:     printf("Elementos de la lista %d: ", i);
17:
```

```
18:      for(j=1; j<=nele; j++)
19:      {
20:        scanf("%d",&ele);
21:        prom = prom + ele;
22:      }
23:      scanf("%*c");
24:    }
25:    prom = prom/tele;
26:    printf("El valor medio es %.1f:", prom);
27: }
```

La tabla 5.4 muestra un ejemplo de ejecución paso a paso del programa anterior.

Tabla 5.4
Ejecución del programa de sentencias `for` anidadas

Línea	Valor de las variables							Entrada por teclado	Salida en pantalla
	i	j	nlis	nele	tele	ele	prom		
4	?	?	?	?	?	?			
5	?	?	?	?	?	?	0.0		
7	?	?	?	?	0	?	0.0		
8	?	?	?	?	0	?	0.0		Numero de listas:
9	?	?	2	?	0	?	0.0	2	
11 (c)	1	?	2	?	0	?	0.0		
13	1	?	2	?	0	?	0.0		Cantidad de elementos de la lista 1:
14	1	?	2	2	0	?	0.0	2	
15	1	?	2	2	2	?	0.0		
16	1	?	2	2	2	?	0.0		Elementos de la lista 1:
18 (c)	1	1	2	2	2	?	0.0		
20	1	1	2	2	2	1	0.0	1 3	
21	1	1	2	2	2	1	1.0		
18 (c)	1	2	2	2	2	1	1.0		
20	1	2	2	2	2	3	1.0		
21	1	2	2	2	2	3	4.0		
18 (f)	1	3	2	2	2	3	4.0		
23	1	3	2	2	2	3	4.0		
11 (c)	2	3	2	2	2	3	4.0		
13	2	3	2	2	2	3	4.0		Cantidad de elementos de la lista 2:
14	2	3	2	1	2	3	4.0	1	
15	2	3	2	1	3	3	4.0		
16	2	3	2	1	3	3	4.0		Elementos de la lista 2:
18 (c)	2	1	2	1	3	3	4.0		
20	2	1	2	1	3	5	4.0	5	
21	2	1	2	1	3	5	9.0		
18 (f)	2	2	2	1	3	5	9.0		
23	2	2	2	1	3	5	9.0		
11 (f)	3	2	2	1	3	5	9.0		
25	3	2	2	1	3	5	3.0		
26	3	2	2	1	3	5	3.0		El valor medio es: 3.0

Obsérvese en el programa que cada bucle `for` es controlado por diferentes variables de control. El primer bucle tiene la variable de control `i` (v. línea `11`), mientras que el segundo bucle tiene la variable de control `j` (v. línea `18`). Los bucles anidados no tienen la misma variable de control en su estructura. El uso de la misma variable de control en los bucles puede introducir errores lógicos en la ejecución del programa. Sin embargo, cuando los bucles no están anidados, sí se puede utilizar la misma variable de control, ya que los bucles son independientes, de modo que cuando finaliza uno comienza otro.

Ejemplo 3: Sentencias `while` anidadas

El programa siguiente lee desde el teclado una secuencia de números naturales finalizada en `-1` y muestra en pantalla la suma de los dígitos de cada número natural leído.

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int n, sum, aux;
 5:
 6:   printf("Introduzca la secuencia de numeros
             finalizada en -1: \n");
 7:   scanf("%d%*c", &n);
 8:   while(n!=-1)
 9:   {
10:     sum = 0;
11:     aux = n;
12:     while(aux!=0)
13:     {
14:       sum = sum + aux%10;
15:       aux = aux/10;
16:     }
17:     printf("La suma de los digitos de %d es %d\n", n, sum);
18:     scanf("%d%*c", &n);
19:   }
20: }
```

La tabla 5.5 muestra un ejemplo de ejecución paso a paso del programa anterior.

Tabla 5.5
Ejecución del programa de sentencias `while` anidadas

Línea	Valor de las variables			Entrada por teclado	Salida en pantalla
	n	sum	aux		
4	?	?	?		
6	?	?	?	23,4,-1	Introduzca la secuencia de numeros finalizada en -1:
7	23	?	?		
8 (c)	23	?	?		
10	23	0	?		
11	23	0	23		
12 (c)	23	0	23		
14	23	3	23		
15	23	3	2		
12 (c)	23	3	2		
14	23	5	2		
15	23	5	0		
12 (f)	23	5	0		
17	23	5	0		La suma de los digitos de 23 es 5
18	4	5	0		
8 (c)	4	5	0		
10	4	0	0		
11	4	0	4		
12 (c)	4	0	4		
14	4	4	4		
15	4	4	0		
12 (f)	4	4	0		
17	4	4	0		La suma de los digitos de 4 es 4
18	-1	4	0		
8 (f)	-1	4	0		

5.5 Ejemplo de uso de las sentencias iterativas

A continuación, se proporciona un ejemplo de programa en que es necesario utilizar las sentencias iterativas. Supóngase que se quiere hacer un programa en C que muestre en pantalla el menú de opciones siguiente:

```
1. Factorial de un numero entero positivo (>=0)
2. Sumatorio de los n primeros numeros naturales
3. Salir
```

A continuación, el usuario elige una opción del menú anterior. Si la opción elegida es la opción `1`, el programa muestra en pantalla el factorial de un número entero positivo (`>=0`) leído por teclado. El factorial de un número entero positivo `n` se define como sigue:

`n! = n*(n-1)*(n-2)* … *2*1,` siendo `0! = 1`

Si la opción elegida es la opción `2`, el programa muestra en pantalla la suma de los `n` primeros números naturales, donde el valor de `n` es un valor que el usuario introduce desde el teclado.

Al finalizar la ejecución de la opción `1` o la opción `2`, el programa ha de mostrar

de nuevo en pantalla el menú de opciones para que el usuario nuevamente elija una opción. Si la opción elegida es la opción `3`, el programa finaliza su ejecución.

Si el usuario introduce una opción incorrecta, el programa muestra en pantalla el mensaje: `La opcion introducida es incorrecta` y el usuario ha de volver a introducir la opción. Téngase en cuenta que el usuario puede equivocarse varias veces seguidas.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
1. Factorial de un numero entero positivo (>=0)
2. Sumatorio de los n primeros numeros naturales
3. Salir

Introduzca una opcion: 1
Introduzca un numero entero (>=0): 4
El factorial de 4 es: 4! = 24

1. Factorial de un numero entero positivo (>=0)
2. Sumatorio de los n primeros numeros naturales
3. Salir

Introduzca una opcion: 2
Introduzca la cantidad de numeros naturales: 5
La suma de los 5 primeros naturales es: 15

1. Factorial de un numero entero positivo (>=0)
2. Sumatorio de los n primeros numeros naturales
3. Salir

Introduzca una opcion: 8
La opcion introducida es incorrecta
Introduzca de nuevo una opcion: -1

La opcion introducida es incorrecta
Introduzca de nuevo una opcion: 3

El programa ha finalizado
```

La implementación del programa es:

```
#include <stdio.h>
main()
{
  int opc=-1, num, suma, i;
  double facto;   /* esta variable se declara como double para
                     tener un mayor rango de representación */

  while(opc!=3)
  {
    printf("\n");
```

```
    printf("1. Factorial de un numero entero positivo (>=0)\n");
    printf("2. Sumatorio de los n primeros numeros naturales\n");
    printf("3. Salir\n\n");

    printf("Introduzca una opcion: ");
    scanf("%d%*c", &opc);

    while ((opc!=1)&&(opc!=2)&&(opc!=3))
    {
      printf("La opcion introducida es incorrecta\n");
      printf("Introduzca de nuevo una opcion: ");
      scanf("%d%*c", &opc);
      printf("\n");
    }

    if(opc == 1)
    {
      printf("Introduzca un numero entero (>=0): ");
      scanf("%d%*c", &num);

      facto = 1.0;
      for(i=num; i>1; i--)
        facto = facto*i;

      printf("El factorial de %d es: %d! = %.0f\n", num,
              num, facto);
     }
     else if (opc==2)
     {
       suma = 0;

       printf("Introduzca la cantidad de numeros naturales: ");
       scanf("%d%*c", &num);

       for(i=1; i<=num; i++)
         suma = suma+i;

       printf("La suma de los %d primeros naturales es: %d\n",
               num, suma);
     }
     else
      printf("El programa ha finalizado\n");
   }
}
```

5.6 Ejercicios

1. Escriba un programa en lenguaje C que determine, mostrando un mensaje por pantalla, la suma de `10` números enteros leídos desde teclado. Suponga que los números enteros leídos son válidos.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca diez numeros enteros (separados por un espacio): 2
3 1 6 7 1 0 -1 -5 10
La suma de los enteros leidos es: 24
```

2. Escriba un programa en lenguaje C que muestre en pantalla el valor de: $\sum_{i=1}^{10} i^3$.

   Nota. La suma de los `10` primeros términos de este sumatorio es `3025`.

3. Escriba un programa en lenguaje C que determine, mostrando un mensaje en pantalla, la suma de los `20` primeros números enteros positivos múltiplos de `5` y de `7`.

   Nota. La suma de los `20` primeros números enteros positivos múltiplos de `5` y de `7` es `7350`.

4. Escriba un programa en lenguaje C que determine y muestre en pantalla la media aritmética de una lista de números reales positivos introducida desde el teclado y finalizada en el valor `-1.0`. La media aritmética de una lista de números se calcula como la división entre la suma de los valores de todos los números reales y la cantidad de números reales de la lista. Suponga que los números reales introducidos son válidos, que el valor `-1.0` no es un número de la lista y que la lista tiene, como mínimo, un número real distinto del valor de `-1.0`.

   Nota. La media aritmética de `2.0 4.0 0.0 -1.0` es: `(2.0+4.0+0.0)/3`

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca la lista de numeros reales (>0) finalizada en
-1.0: 2.0 4.0 0.0 -1
La media aritmetica de los numeros reales de la lista es:
2.0
```

5. Escriba un programa en lenguaje C que muestre por pantalla una lista de números enteros contando de `3` en `3`, comenzando por el valor entero `5`, hasta el valor `50`. Además, el programa ha de mostrar por pantalla un mensaje que indique el resultado de la suma de los números enteros generados que sean divisibles por `5`.

   Ejemplo de ejecución:

```
La lista de los números enteros es: 5 8 11 14 17 20 23 26 29
32 35 38 41 44 47 50

La suma de los números enteros generados divisibles por 5 es:
110
```

6. Dada una cadena de caracteres leída desde el teclado y finalizada en el carácter punto, escriba un programa en lenguaje C que muestre en pantalla la cadena de caracteres leída cambiando la vocal `a` por la `e`, la vocal `e` por la `i`, la vocal `i` por la `o`, la vocal `o` por la `u` y, finalmente, la vocal `u` por la `a`. Suponga que la cadena solo incluye letras minúsculas.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

   ```
   Introduzca la cadena de caracteres finalizada en punto: la vieja ines.
   La cadena modificada es: le voije onis.
   ```

7. Escriba un programa en lenguaje C que calcule el valor mínimo de la función `f`, definida como $f(x) = 5x^2+3x+8$, para los valores enteros de `x` comprendidos entre $-3<x<10$. El programa mostrará en pantalla el valor mínimo de la función y el valor de `x` en que se alcanza dicho valor.

   Nota. El valor mínimo de la función `f(x)` es `8` y se alcanza en `x = 0`.

8. Para cada uno de los programas siguientes, calcule la cantidad de iteraciones de cada bucle individualmente y luego determine la cantidad total de iteraciones. Además, dibuje para cada programa la tabla de ejecución paso a paso.

a.

```
#include <stdio.h>
main()
{
  int i, j, cont=0;
  for(i=0; i<3; i++)
    for(j=10; j>8; j--)
    {
      printf("Iteracion %d\n", cont+1);
      cont = cont + 1;
    }
}
```

b.

```
#include <stdio.h>
main()
{
  int i, cont=0;
  for(i=0; i<3; i++)
    for(i=10; i>8; i--)
    {
      printf("Iteracion %d\n", cont+1);
      cont = cont + 1;
    }
}
```

c.

```
#include <stdio.h>
main()
{
  int i, j, cont=0;
  for(i=0; i<3; i++)
    for(j=i; j<=2; j++)
    {
      printf("Iteracion %d\n", cont+1);
      cont = cont + 1;
    }
}
```

9. Escriba un programa en lenguaje C que muestre en pantalla la tabla de multiplicar que el usuario desee. El programa ha de comprobar que el valor introducido por el usuario es un valor comprendido entre `1` y `10`. Si no es así, el programa ha de indicar que es un error mostrando un mensaje por pantalla y debe volver a pedir el valor de la tabla. Considere que el usuario puede equivocarse al introducir el valor varias veces seguidas.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca la tabla de multiplicar que desee: 12
Error!!! Introduzca de nuevo la tabla de multiplicar: 2

2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

10. Escriba un programa en lenguaje C que muestre en pantalla todas las tablas de multiplicar como se muestra a continuación:

```
1 x 1 = 1       2 x 1 = 2       3 x 1 = 3
1 x 2 = 2       2 x 2 = 4       3 x 2 = 6
1 x 3 = 3       2 x 3 = 6       3 x 3 = 9
1 x 4 = 4       2 x 4 = 8       3 x 4 = 12
1 x 5 = 5       2 x 5 = 10      3 x 5 = 15
1 x 6 = 6       2 x 6 = 12      3 x 6 = 18
1 x 7 = 7       2 x 7 = 14      3 x 7 = 21
1 x 8 = 8       2 x 8 = 16      3 x 8 = 24
1 x 9 = 9       2 x 9 = 18      3 x 9 = 27
1 x 10 = 10     2 x 10 = 20     3 x 10 = 30

4 x 1 = 4       5 x 1 = 5       6 x 1 = 6
4 x 2 = 8       5 x 2 = 10      6 x 2 = 12
4 x 3 = 12      5 x 3 = 15      6 x 3 = 18
```

```
4 x 4 = 16          5 x 4 = 20          6 x 4 = 24
4 x 5 = 20          5 x 5 = 25          6 x 5 = 30
4 x 6 = 24          5 x 6 = 30          6 x 6 = 36
4 x 7 = 28          5 x 7 = 35          6 x 7 = 42
4 x 8 = 32          5 x 8 = 40          6 x 8 = 48
4 x 9 = 36          5 x 9 = 45          6 x 9 = 54
4 x 10 = 40         5 x 10 = 50         6 x 10 = 60

7 x 1 = 7           8 x 1 = 8           9 x 1 = 9
7 x 2 = 14          8 x 2 = 16          9 x 2 = 18
7 x 3 = 21          8 x 3 = 24          9 x 3 = 27
7 x 4 = 28          8 x 4 = 32          9 x 4 = 36
7 x 5 = 35          8 x 5 = 40          9 x 5 = 45
7 x 6 = 42          8 x 6 = 48          9 x 6 = 54
7 x 7 = 49          8 x 7 = 56          9 x 7 = 63
7 x 8 = 56          8 x 8 = 64          9 x 8 = 72
7 x 9 = 63          8 x 9 = 72          9 x 9 = 81
7 x 10 = 70         8 x 10 = 80         9 x 10 = 90

10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

11. Indique el mensaje que muestran en pantalla cada uno de los programas siguientes:

a.

```
#include <stdio.h>
main()
{
  int i;
  for(i=1; i==1; i--)
    printf("Iteracion %d\n", i);
}
```

b.

```
#include <stdio.h>
main()
{
  int i;
  for(i=1; i>10; i++)
    printf("Iteracion %d\n", i);
}
```

c.

```
#include <stdio.h>
main()
{
  int i;
  for(i=10; i>=10; i++)
    printf("Iteracion %d\n", i);
}
```

d.

```
#include <stdio.h>
main()
{
  int i;
  for(i=-2; i<2; i++)
    printf("Iteracion %d\n", i);
}
```

e.

```
#include <stdio.h>
main()
{
  int i;
  for(i=-2; i=2; i++)
    printf("Iteracion %d\n", i);
}
```

5.7 Respuesta a los ejercicios propuestos

1.

```
#include <stdio.h>
main()
{
  int i, num, suma = 0;

  printf("Introduzca diez numeros enteros (separados por
          por un espacio): ");
  for(i=0; i<10; i++)
  {
    scanf("%d", &num);
    suma = suma + num;
  }
  scanf("%*c");
  printf("La suma de los numeros enteros leidos es:
          %d\n", suma);
}
```

2.

```
#include <stdio.h>
main()
{
  int i, res;

  res = 0;
```

```
  for(i=1; i<=10; i=i+1)
    res = res + i*i*i;

  printf("El resultado de la suma de los 10 primeros
          terminos del sumatorio es: %d\n", res);
}
```

3.

```
#include <stdio.h>
main()
{
  int cont, num, suma =0;

  num = 1;
  cont = 0;
  while (cont < 20)
  {
    if ((num%5 == 0) && (num%7 == 0))
    {
      suma = suma + num;
      cont = cont + 1;
    }
    num = num + 1;
  }
  printf("La suma de los veinte primeros numeros enteros
          positivos multiplos de 5 y de 7 es:
          %d\n", suma);
}
```

4.

```
#include <stdio.h>
main()
{
  int n;
  float num, suma = 0.0;

  printf("Introduzca la lista de numeros reales (>0)
          finalizada en -1.0: ");
  scanf("%f", &num);
  n = 0;
  while (num >= 0)
  {
    suma = suma + num;
    n = n + 1;
    scanf("%f", &num);
  }
  printf("La media aritmetica de los numeros reales de la
          lista es: %.1f\n", suma/n);
}
```

5.

```
#include <stdio.h>
main()
{
  int num, suma = 0;
```

```
  printf("La lista de los numeros enteros es: ");
  num = 5;
  while (num <= 50)
  {
    printf("%d ", num);
    if (num%5 == 0)
      suma = suma + num;

    num = num + 3;
  }
  printf("\nLa suma de los numeros enteros generados
            divisibles por 5 es: %d\n", suma);
}
```

6.

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

  printf("Introduzca la cadena de caracteres finalizada en
          punto: ");
  scanf("%c", &car);
  printf("La cadena modificada es: ");

  while (car != '.')
  {
    if (car == 'a')
      printf("e");
    else if (car == 'e')
      printf("i");
    else if (car == 'i')
      printf("o");
    else if (car == 'o')
      printf("u");
    else if (car == 'u')
      printf("a");
    else
      printf("%c", car);

    scanf("%c", &car);
  }
  scanf("%*c");
  printf(".\n");
}
```

7.

```
#include <stdio.h>
main()
{
  int i, minimo, min_x, aux;

  min_x = -2;
  minimo = 5*min_x*min_x+3*min_x +8;
```

```
  for(i=-1;i<10; i++)
  {
    aux = 5*i*i+3*i+8;
    if(aux<minimo)
    {
      minimo = aux;
      min_x = i;
    }
  }
  printf("El valor minimo de la funcion es %d
          para x = %d\n", minimo, min_x);
}
```

8.

a.

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int i, j, cont=0;
 5:
 6:   for(i=0; i<3; i++)
 7:     for(j=10; j>8; j--)
 8:     {
 9:       printf("Iteracion %d\n", cont+1);
10:       cont = cont + 1;
11:     }
12: }
```

La cantidad de iteraciones del bucle `for(i=0; i<3; i++)` (línea 6) es: `(3-1) + 0 + 1 = 3` iteraciones.

La cantidad de iteraciones del bucle `for(j=10; j>8; j--)` (línea 7) es: `10 - (8+1) + 1 = 2` iteraciones.

La cantidad total de iteraciones es: `3*2 = 6` iteraciones.

Línea	Valor de las variables			Entrada por teclado	Salida en pantalla
	i	j	cont		
4	?	?	0		
6 (c)	0	?	0		
7 (c)	0	10	0		
9	0	10	0		Iteracion 1
10	0	10	1		
7 (c)	0	9	1		
9	0	9	1		Iteracion 2
10	0	9	2		
7 (f)	0	8	2		
6 (c)	1	8	2		
7 (c)	1	10	2		
9	1	10	2		Iteracion 3
10	1	10	3		
7 (c)	1	9	3		
9	1	9	3		Iteracion 4
10	1	9	4		
7 (f)	1	8	4		
6 (c)	2	8	4		
7 (c)	2	10	4		
9	2	10	4		Iteracion 5
10	2	10	5		
7 (c)	2	9	5		
9	2	9	5		Iteracion 6
10	2	9	6		
7 (f)	2	8	6		
6 (f)	3	8	6		

b.

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int i, cont=0;
 5:
 6:   for(i=0; i<3; i++)
 7:     for(i=10; i>8; i--)
 8:     {
 9:       printf("Iteracion %d\n", cont+1);
10:       cont = cont + 1;
11:     }
12: }
```

La cantidad de iteraciones del bucle `for(i=0; i<3; i++)` (línea 6), si ignoramos el código del cuerpo del bucle, es: `(3-1) + 0 + 1 = 3` iteraciones.

La cantidad de iteraciones del bucle `for(i=10; i>8; i--)` (línea 7) es: `10 - (8+1) + 1 = 2` iteraciones.

Sin embargo, en este caso, la cantidad total de iteraciones es: `1*2 = 2` iteraciones, puesto que el bucle más interno cambia el valor de la variable `i` (variable de control de ambos bucles) y el bucle más externo solo realiza una iteración, ya que el valor de la variable `i` después de ejecutar el bucle

más interno y de hacer la actualización (`i++`) del for externo es `9` y, por tanto, la comparación `i<3` es falsa.

Línea	Valor de las variables		Entrada por teclado	Salida en pantalla
	`i`	`cont`		
4	?	0		
6 (c)	0	0		
7 (c)	10	0		
9	10	0		Iteracion 1
10	10	1		
7 (c)	9	1		
9	9	1		Iteracion 2
10	9	2		
7 (f)	8	2		
6 (f)	9	2		

c.

```
 1: #include <stdio.h>
 2: main()
 3: {
 4:   int i, j, cont=0;
 5:
 6:   for(i=0; i<3; i++)
 7:     for(j=i; j<=2; j++)
 8:     {
 9:       printf("Iteracion %d\n", cont+1);
10:       cont = cont + 1;
11:     }
12: }
```

La cantidad de iteraciones del bucle `for(i=0; i<3; i++)` (línea 6) es: `(3-1) + 0 + 1 = 3` iteraciones.

La cantidad de iteraciones del bucle `for(j=i; j<=2; j++)` (línea 7) aplicando la fórmula es `2 - i + 1` para `i = {0, 1, 2}`. Observe que la cantidad de iteraciones del bucle más interno varía por cada iteración del bucle más externo. Entonces la cantidad total de iteraciones es $\sum_{i=0}^{2}(2-i+1)$ = `3 + 2 + 1 = 6` iteraciones.

Línea	Valor de las variables			Entrada por teclado	Salida en Pantalla
	i	j	cont		
4	?	?	0		
6 (c)	0	?	0		
7 (c)	0	0	0		
9	0	0	0		Iteracion 1
10	0	0	1		
7 (c)	0	1	1		
9	0	1	1		Iteracion 2
10	0	1	2		
7 (c)	0	2	2		
9	0	2	2		Iteracion 3
10	0	2	3		
7 (f)	0	3	3		
6 (c)	1	3	3		
7 (c)	1	1	3		
9	1	1	3		Iteracion 4
10	1	1	4		
7 (c)	1	2	4		
9	1	2	4		Iteracion 5
10	1	2	5		
7 (f)	1	3	5		
6 (c)	2	3	5		
7 (c)	2	2	5		
9	2	2	5		Iteracion 6
10	2	2	6		
7 (f)	2	3	6		
6 (f)	3	3	6		

9.

```
#include <stdio.h>
main()
{
  int tabla, i;

  printf("Introduzca la tabla de multiplicar: ");
  scanf("%d%*c", &tabla);

  while((tabla<1) || (tabla>10))
  {
    printf("Error!!! Introduzca de nuevo la tabla de
            multiplicar: ");
    scanf("%d%*c", &tabla);
  }
  for(i=1; i<=10; i++)
    printf("%d x %d = %d\n", tabla, i, tabla*i);
}
```

10.

```
#include <stdio.h>
main()
{
  int t, i, j;

  t = 1;
  for(i=1; i<=3; i++)
```

```
    {
      for(j=1; j<=10; j++)
        printf("%d x %d = %d\t\t %d x %d = %d\t\t
                %d x %d = %d\n", t, j, t*j, t+1, j, (t+1)*j,
                t+2, j, (t+2)*j);
      t = t+3;
      printf("\n");
    }
    for(i=1; i<=10; i++)
      printf("10 x %d = %d\n", 10, i, 10*i);
  }
```

11.

a.

```
Iteracion 1
```

b.

El programa no muestra ningún mensaje por pantalla

c.

```
Iteracion 10
Iteracion 11
Iteracion 12
…
```

El programa no finaliza porque la comparación de la estructura de control del bucle es siempre cierta.

d.

```
Iteracion -2
Iteracion -1
Iteracion 0
Iteracion 1
```

e.

```
Iteracion 2
Iteracion 2
Iteracion 2
…
```

El programa no finaliza porque en la comparación de la estructura de control del bucle se está utilizando el operador de asignación en lugar de un operador relacional. Por tanto, se está asignando `2` a la variable `i` y, a continuación, se evalúa la expresión `i`. Como la variable `i` almacena un `2` (valor distinto de `0`), la expresión se evalúa siempre como cierta.

Estructuras

En los capítulos anteriores, únicamente se han manipulado los tipos de datos elementales (enteros, caracteres y reales) que el propio lenguaje ofrece. En este capítulo y en el siguiente, se muestra cómo el programador puede crear nuevos tipos de datos, más complejos, que permitirán no solo almacenar la información de una manera más estructurada, sino también poder implementar programas de mayor dificultad. El capítulo se inicia mostrando al lector cómo declarar y manipular estructuras, y en el próximo capítulo se explican los vectores.

6.1 Estructuras

Hasta ahora se han visto tipos de datos elementales (enteros, caracteres, reales) que permiten almacenar un único dato de un tipo determinado. Así, se puede declarar una variable de tipo real para almacenar el área de un cuadrado, o una variable de tipo entera para almacenar la suma de los dígitos de un número, o una variable de tipo carácter para almacenar la primera letra de un nombre, etc.

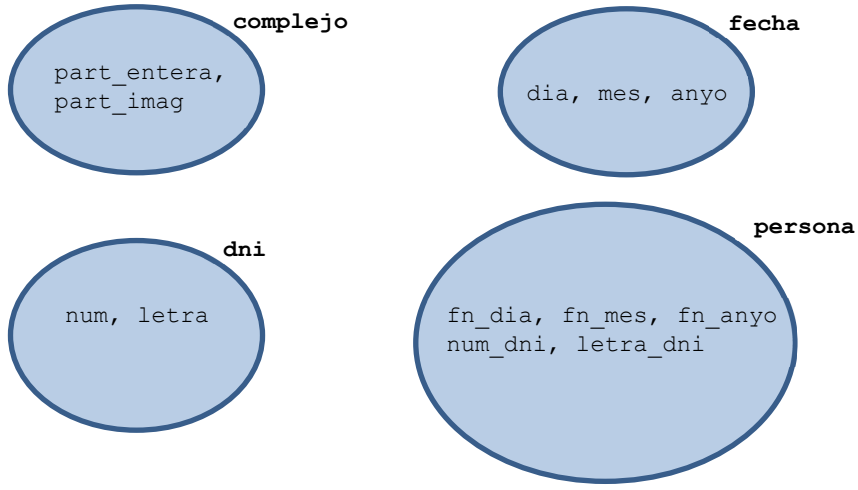
Ahora bien, en muchos programas la información que se quiere manipular es un poco más compleja y está compuesta por varios elementos de igual o de distinto tipo. Por ejemplo: un número complejo que está compuesto por dos valores reales (una parte real y una parte imaginaria), una fecha que consta de tres enteros (día, mes y año), un DNI que está formado por un entero y un carácter (número y letra del DNI), o incluso los datos de una persona, que podrían ser su DNI y su fecha de nacimiento.

Con lo que se ha visto hasta ahora, este tipo de información se podría manipular declarando variables individuales que almacenen el valor de cada elemento, tal como se muestra a continuación:

```
/* Variables para almacenar la información de un número complejo */
float part_entera, part_imag;

/* Variables para almacenar la información de una fecha */
int dia, mes, anyo;

/* Variables para almacenar la información de un DNI */
int num;
char letra;

/* Variables para almacenar la información de una persona (DNI
   y fecha de nacimiento) */
int fn_dia, fn_mes, fn_anyo;
int num_dni;
char letra_dni;
```

Sin embargo, nos gustaría poder declarar una única variable que contenga toda la información que está relacionada, es decir, una única variable para almacenar la información de un complejo, otra para almacenar una fecha, otra para almacenar un DNI y, finalmente, otra para los datos de una persona (v. Fig. 6.1). En total, nos gustaría declarar únicamente cuatro variables, en lugar de las doce anteriores. Esto se consigue con las *estructuras*.

Fig. 6.1
Ejemplos de estructuras



Una *estructura* es un tipo de dato estructurado heterogéneo, compuesto por varios elementos que pueden ser de tipos de datos diferentes, tanto elementales como estructurados.

6.2 Declaración de estructuras

En el lenguaje de programación C, las variables de tipo *estructura* se pueden declarar de varias maneras. En este libro, solamente se estudia una de ellas, que consiste en una declaración en dos pasos: en primer lugar, se define el tipo de dato *estructura* para indicar los diferentes elementos que componen la estructura y, en segundo lugar, se declaran las variables de ese nuevo tipo de dato. A continuación, se muestra la sintaxis general y algunos ejemplos para cada uno de estos pasos.

Definición del tipo de dato

El lenguaje C permite definir nuevos tipos de datos utilizando el especificador de identificador `typedef`. Con este especificador, se puede dar un nombre o identificador a un tipo de dato. El especificador `typedef` se puede utilizar con cualquier tipo de datos, pero ahora se estudia asociado a las estructuras.

La sintaxis general para definir un tipo de dato estructura y asignarle un identificador es la siguiente:

```
typedef struct
{
  tipo_camp1  nom_camp1;       /* Definición del campo 1 */
  tipo_camp2  nom_camp2;       /* Definición del campo 2 */
   ...
  tipo_campN  nom_campN;       /* Definición del campo N */
} nombre_tipo_estructura;
```

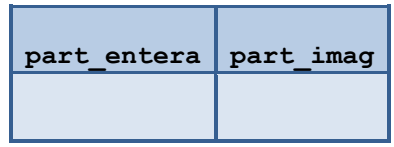
Con estas líneas de código, se define un tipo de dato *estructura* formado por `N` elementos, que se referencia con el identificador `nombre_tipo_estructura`. Los elementos de una estructura se denominan *campos* y cada uno de los campos puede ser de cualquier tipo de dato (elemental, estructura, vectores, punteros, etc.). Esta definición de tipo de dato se hace fuera del programa principal, justo después de incluir las librerías y definir constantes (v. capítulo 2, sección 2.6).

Como norma de estilo (v. anexo 9.B), se utiliza siempre como `nombre_tipo_estructura` un identificador que comience con el carácter `t` o `T`, para así identificar fácilmente que se refiere al nombre de un tipo de dato y no al de una variable.

A continuación, se muestran algunos ejemplos de definición de estructuras y su representación gráfica.

Ejemplo 1:

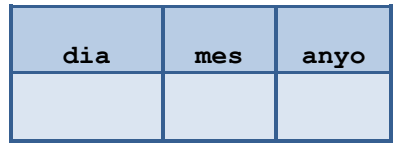
```
typedef struct
{
  float part_entera;
  float part_imag;
} tcomplejo;
```



Se ha definido el tipo de dato `tcomplejo` como una estructura con dos campos de tipo `float`: la parte real (`part_entera`) y la parte imaginaria (`part_imag`).

Ejemplo 2:

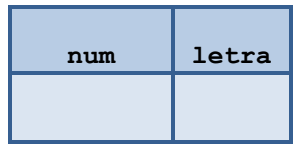
```
typedef struct
{
  int dia, mes, anyo;
} tfecha;
```



Aquí se ha definido el tipo de dato `tfecha` como una estructura compuesta por tres campos de tipo `int`: `dia`, `mes` y `anyo`.

Ejemplo 3:

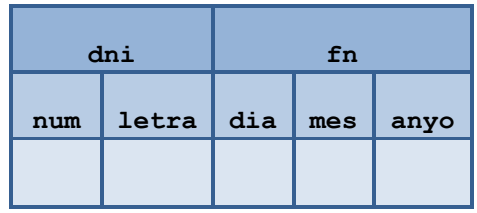
```
typedef struct
{
  int num;
  char letra;
} tdni;
```



Este ejemplo define el tipo de dato `tdni` como una estructura formada por dos campos de diferente tipo: uno de tipo `int` para almacenar el número de dni (`num`) y otro de tipo `char` para almacenar la letra (`letra`).

Ejemplo 4:

```
typedef struct
{
  tdni dni;
  tfecha fn;
} tpersona;
```



Finalmente, se muestra cómo definir estructuras cuyos campos son también de tipo estructura (estructuras anidadas). Aquí se supone que previamente se han definido los tipos `tdni` y `tfecha` (ejemplos 2 y 3). Así, se define el tipo `tpersona` como una estructura formada por dos campos: el campo `dni` para

almacenar el dni de la persona y el campo `fn` para almacenar su fecha de nacimiento.

Con respecto a las normas de estilo, obsérvese que, en todos los ejemplos anteriores, el nombre del tipo de dato empieza por el carácter `t` (como se ha comentado anteriormente) y que hay un espacio de separación entre la llave y este carácter. Obsérvese también que todos los campos de la estructura están indentados a `2` espacios.

Declaración de la variable de tipo estructura

Una vez definido el nuevo tipo de dato, se pueden declarar variables de este tipo. Es ahora, en la declaración, cuando se reserva espacio en la memoria para almacenar los valores que tomarán las variables durante la ejecución del programa. Para las variables de tipo estructura, se reservan en la memoria el número de bytes equivalente a la suma de los bytes que ocupa cada uno de sus campos. Por ejemplo, para una variable de tipo `tfecha`, se reservan `12` bytes, que corresponden a los `4` bytes que ocupa cada uno de sus campos de tipo `int`.

La sintaxis general para declarar una variable de tipo estructura es exactamente igual que la que se utiliza para declarar variables de tipo elemental:

```
/* Declaración de una variable de tipo estructura */
nombre_tipo_estructura nom_var;

/* Declaración de dos variables de tipo estructura */
nombre_tipo_estructura nom_var1, nom_var2;

/* Declaración e inicialización de una variable de tipo
   estructura */
nombre_tipo_estructura nom_var={val_inic_camp1, val_inic_camp2,
                                ... , val inic campN};
```

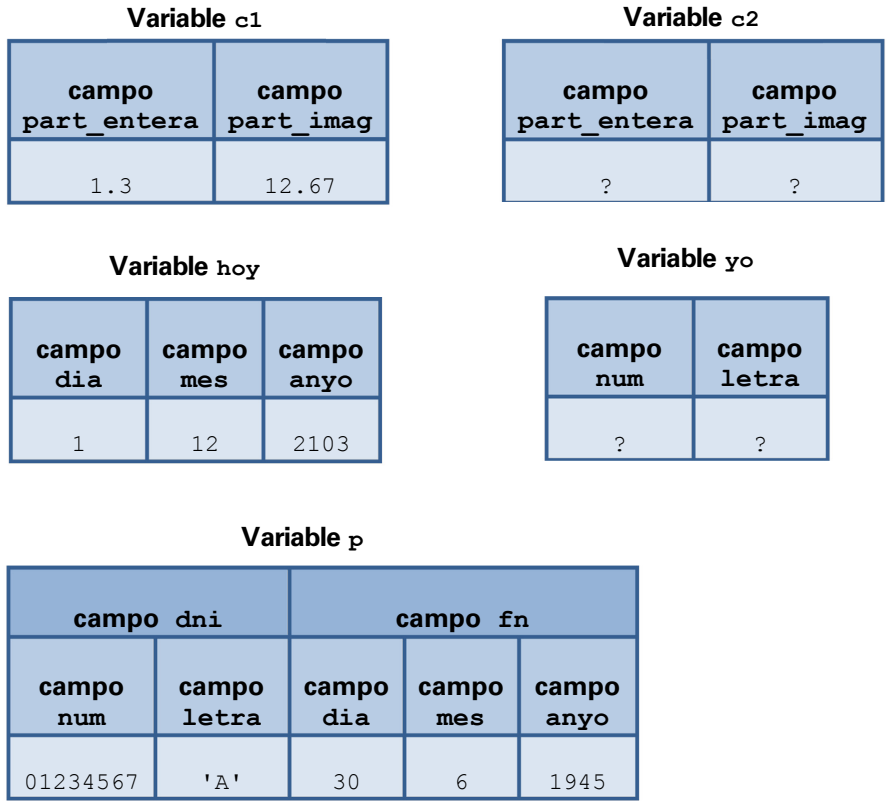
Algunos ejemplos de declaración de variables de tipo estructura, asumiendo las definiciones de los tipos de datos anteriores, son los siguientes:

```
/* Declara una variable de tipo estructura */
tdni yo;

/* Declara e inicializa una variable de tipo estructura */
tfecha hoy={1, 12, 2013};

/* Declara dos variables de tipo estructura (c1 y c2) e
   inicializa una de ellas (c1) */
tcomplejo c1={1.3, 12.67}, c2;

/* Declara e inicializa una variable de tipo estructura */
tpersona p={{01234567, 'A'}, {30, 6, 1945}};
```

Obsérvese en el último ejemplo que, cuando el campo que se inicializa es, a su vez, una estructura (campo `dni` o `fn` de la variable `p`), los valores del campo se agrupan entre llaves.

La figura 6.2 muestra gráficamente cómo se almacena la información en cada una de estas variables. El carácter `'?'` indica que el valor es indeterminado.

Fig. 6.2
Ejemplos de declaración e inicialización de variables de tipo estructura



6.3 Operaciones con estructuras

A continuación, se describen los tres tipos de operaciones que se pueden realizar con las estructuras: acceso a un campo concreto de la estructura, operaciones sobre un campo y, finalmente, operaciones con toda la estructura completa.

Acceso a un campo de la estructura

La operación básica relacionada con las estructuras es el acceso a un campo concreto de la estructura.

La sintaxis general para acceder a un campo de una variable de tipo estructura es la siguiente:

```
nom_var.nom_camp      /* Acceso a un campo de una estructura */
```

donde `nom_var` es el nombre de la variable de tipo estructura y `nom_camp` es el nombre del campo al cual se quiere acceder.

Como ejemplo, suponiendo la declaración de variables de tipo estructura del apartado anterior (v. sección 6.2), las expresiones siguientes para acceder a los campos de las variables son correctas:

```
hoy.dia       /* Acceso al campo dia de la variable hoy */
c1.part_imag /* Acceso al campo part_imag de la variable c1 */
yo.letra     /* Acceso al campo letra de la variable yo */
p.fn.anyo    /* Acceso al campo anyo del campo fn de
                la variable p */
```

Obsérvese en el último ejemplo que, en las estructuras anidadas, si el campo al cual se accede es, a su vez, de tipo estructura, se vuelve a utilizar el operador "`.`" para acceder a un campo concreto de dicha estructura.

Operaciones con un campo

Las operaciones que se pueden realizar con un campo concreto de una estructura son las correspondientes a su tipo. Por tanto, si el campo es de tipo entero, se pueden realizar las mismas operaciones que con una variable de tipo entero (suma, resta, módulo, comparaciones, etc.); si es de tipo carácter, se pueden realizar todas las operaciones válidas para caracteres, etc.

Considerando los ejemplos de la sección 6.2, algunos códigos que operan con campos de estructuras son los siguientes:

Ejemplo 1:

```
c1.part_entera = 2.1;
c1.part_imag = 3.4;

/* Inicializa los dos campos de la variable c1 a un valor cons-
    tante */
```

Ejemplo 2:

```
c1.part_entera = c1.part_entera + 1;

/* Incrementa en 1 el campo part_entera de la variable c1 */
```

Ejemplo 3:

```
if (hoy.dia==20 && hoy.mes==11)
  printf ("Felicidades\n");

/* Consulta si los campos dia y mes de la variable hoy son
   20 y 11, respectivamente */
```

Ejemplo 4:

```
if (yo.letra=='A')
  printf("OK\n");
else
  printf("ERROR\n");

/* Consulta si el campo letra de la variable yo es el
   carácter 'A' */
```

Ejemplo 5:

```
if (p.fn.anyo<hoy.anyo-18)
  printf ("Mayor de edad\n");

/* Consulta si el campo anyo del campo fn de la variable p es
   menor que el campo anyo de la variable hoy menos 18, es
   decir, está determinando si la persona p es mayor de edad
   a día de hoy */
```

Ejemplo 6:

```
printf("Introduzca parte entera y parte imaginaria: ");
scanf("%f %f%*c", &c1.part_entera, &c1.part_imag);

/* Lee del teclado los dos campos de la variable c1 */
```

Operaciones con estructuras completas

La única operación que se puede hacer con estructuras completas es asignar una estructura a otra y solamente si estas son del mismo tipo.

Así, suponiendo la declaración de variables del ejemplo de la sección anterior (v. sección 6.2), las sentencias siguientes son correctas y están copiando la información almacenada en una estructura sobre otra del mismo tipo:

```
c1 = c2;     /* Copia el contenido del complejo c2 en c1 */

yo = p.dni; /* Copia el contenido del campo dni de la variable p
               en la variable yo */

p.fn = hoy; /* Copia el contenido de la variable hoy en el campo
               fn de la variable p */
```

Como se acaba de ver, el lenguaje C permite copiar estructuras completas con una única sentencia de asignación, pero no permite comparar estructuras aun-

que sean del mismo tipo. Así, el código siguiente es incorrecto y el compilador da un mensaje de error que indica que los operandos del operador == no son válidos:

```
if (c1==c2)           /* CÓDIGO INCORRECTO */
{
  ...
}
```

La manera correcta de comparar estructuras es hacerlo campo a campo. El código correcto del ejemplo anterior es:

```
/* CÓDIGO CORRECTO */
if (c1.part_entera==c2.part_entera && c1.part_imag==c2.part_imag)
{
  ...
}
```

6.4 Ejemplo de uso de estructuras

Finaliza el capítulo con un ejemplo de programa en que es necesario utilizar estructuras. Supóngase que debe almacenarse la información siguiente de una persona:

- Número y letra del DNI
- Fecha de nacimiento (día, mes y año)
- Sexo (masculino o femenino)

El tipo de dato adecuado para almacenar esta información es una estructura que se puede representar gráficamente tal como muestra la figura 6.3.

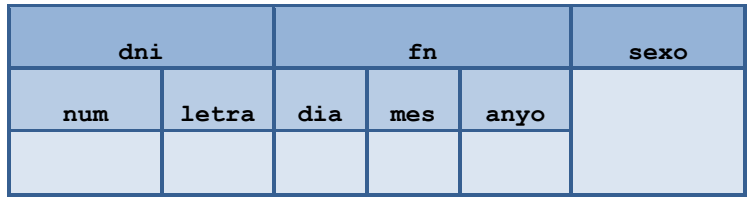


Fig. 6.3
Representación gráfica del tipo de dato `tpersona`

Utilizando esta estructura, se escribe un programa que lea del teclado la información de `5` personas y muestre por pantalla el DNI de la persona más joven del sexo femenino y el de la más adulta del sexo masculino. A continuación, se muestra un ejemplo de ejecución.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
DNI de la persona 1: 88888888A
F. nacimiento persona 1 (d/m/a): 05/03/76
```

```
Sexo de la persona 1 (m o f): f

DNI de la persona 2: 11111111B
F. nacimiento persona 2 (d/m/a): 03/05/81
Sexo de la persona 2 (m o f): m

DNI de la persona 3: 12345678T
F. nacimiento persona 3 (d/m/a): 11/12/75
Sexo de la persona 3 (m o f): m

DNI de la persona 4: 22222222R
F. nacimiento persona 4 (d/m/a): 05/03/71
Sexo de la persona 4 (m o f): f

DNI de la persona 5: 33333333Z
F. nacimiento persona 5 (d/m/a): 01/11/78
Sexo de la persona 5 (m o f): f

El DNI de la persona femenina mas joven 33333333Z
El DNI de la persona masculina mas adulta es 12345678T
```

El programa es el siguiente:

```
#include <stdio.h>
#define NUMPERSONAS 5

typedef struct
{
  int dia,mes,anyo;
} tfecha;

typedef struct
{
  int num;
  char letra;
} tdni;

typedef struct
{
  tdni dni;     /* DNI (número y letra) */
  tfecha fn;    /* Fecha de nacimiento */
  char sexo;    /* 'm'-masculino, 'f'-femenino */
} tpersona;

main()
{
  tpersona p, joven, adulta;
  int i;

  joven.dni.num = -1;   /* Indica que no se ha encontrado
                           ninguna persona femenina */
  adulta.dni.num = -1;  /* Indica que no se ha encontrado
                           ninguna persona masculina */

  for (i=1; i<=NUMPERSONAS; i++)
  {
    /* Se leen los datos de una persona */
```

```
    printf("DNI de la persona %d: ", i);
    scanf("%d%c%*c", &p.dni.num, &p.dni.letra);
    printf("F. nacimiento persona %d (d/m/a): ", i);
    scanf("%d%*c%d%*c%d%*c", &p.fn.dia, &p.fn.mes, &p.fn.anyo);
    printf("Sexo de la persona %d (m o f): ", i);
    scanf("%c%*c", &p.sexo);

    /* Si es la primera persona femenina, inicializar joven;
       Si es la primera persona masculina, inicializar adulta;
       Si es persona femenina (pero no la primera) y más joven
       que la actual, actualizar joven;
       Si es persona masculina (pero no la primera) y más
       adulta que la actual, actualizar adulta */
    if (p.sexo=='f' && joven.dni.num==-1)
      joven = p;
    else if (p.sexo=='m' && adulta.dni.num==-1)
      adulta = p;
    else if (p.sexo=='f' && (p.fn.anyo>joven.fn.anyo ||
            (p.fn.anyo==joven.fn.anyo &&
             p.fn.mes>joven.fn.mes) ||
            (p.fn.anyo==joven.fn.anyo &&
             p.fn.mes==joven.fn.mes &&
             p.fn.dia>joven.fn.dia)))
      joven = p;
    else if (p.sexo=='m' && (p.fn.anyo<adulta.fn.anyo ||
            (p.fn.anyo==adulta.fn.anyo &&
             p.fn.mes<adulta.fn.mes)||
            (p.fn.anyo==adulta.fn.anyo &&
             p.fn.mes==adulta.fn.mes &&
             p.fn.dia<adulta.fn.dia)))
      adulta = p;
  }

  /* Mostrar el DNI de la mujer más joven y el hombre más
       adulto */
  if (joven.dni.num==-1)
    printf ("No se han introducido personas del sexo
             femenino\n");
  else
    printf ("El DNI de la persona femenina mas joven es
             %d%c\n", joven.dni.num, joven.dni.letra);

  if (adulta.dni.num==-1)
    printf ("No se han introducido personas del sexo
             masculino\n");
  else
     printf ("El DNI de la persona masculina mas adulta es
              %d%c\n", adulta.dni.num, adulta.dni.letra);
}
```

6.5 Ejercicios

1. Defina el tipo de dato estructura `tficha` que permita almacenar la información de una ficha del dominó.
2. Utilizando el tipo de dato `tficha` del problema 1, escriba un programa en C que inicialice una variable de tipo `tficha` con la información de la ficha "doble 6" y la muestre en pantalla.
3. Dada la definición siguiente de tipo de dato y declaración de variables:

```
typedef struct
{
  int mes, anyo;
} tfecha;

typedef struct
{
  int isbn;     /* Numero de ISBN (9 dígitos) */
  tfecha fp;    /* Fecha de publicación */
  int nump;     /* Número de páginas */
} tlibro;

tlibro libro1, libro2;

int n, m;

tfecha fecha1, fecha2;
```

Indique cuáles de las sentencias siguientes son correctas sintácticamente y cuáles darán un error de compilación.

a.
```
libro1.mes = 12;
```
b.
```
libro1.fp = fecha1;
```
c.
```
tfecha.anyo = 2013;
```
d.
```
fecha2.mes = m;
```
e.
```
if (2012 == fecha2.mes)
      m = 1;
```
f.
```
libro2.fp.mes = m + 3;
```
g.
```
n = libro1.nump;
```
h.
```
if (fecha2 == libro1.fp)
   m = 1;
```
i.
```
libro2 = libro1;
```
j.
```
fecha1 = libro2.fp;
```
k.
```
n = libro2.isbn + libro1.isbn;
```
l.
```
if (fecha1.mes == fecha2.mes)
  m = 1;
```
m.
```
scanf("Publicacion: %d\n", &libro2.fp);
```
n.
```
libro1.tfecha = fecha2;
```

4. Dado el siguiente tipo de dato `tfraccion`, que almacena información sobre una fracción:

```
typedef struct
{
  unsigned int num;    /* Numerador */
  unsigned int den;    /* Denominador */
} tfraccion;
```

Escriba un programa en lenguaje C que lea del teclado dos fracciones (con numerador y denominador mayor estricto que cero) y muestre por pantalla el resultado de sumarlas, restarlas, multiplicarlas y dividirlas. No es necesario simplificar las fracciones.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Fraccion 1 (formato x/y): 2/3
Fraccion 2 (formato x/y): 5/8

RESULTADO DE LAS OPERACIONES:
Suma: 2/3 + 5/8 = 31/24

Resta: 2/3 - 5/8 = 1/24

Multiplicacion: 2/3 * 5/8 = 10/24

Division: 2/3 / 5/8 = 16/15
```

5. Dado el siguiente tipo de dato `trecta`, que almacena información sobre la ecuación de una recta (pendiente y ordenada de origen):

```
typedef struct
{
  float m;         /* Pendiente de la recta */
  float b;         /* Ordenada de origen */
} trecta;
```

Escriba un programa en lenguaje C que lea del teclado la ecuación de dos rectas y muestre por pantalla si las rectas son iguales, son paralelas o intersecan en un punto. En este último caso, determine el punto de corte. Utilice un error absoluto de `0.00001` para comparar dos números reales.

Algunos ejemplos de ejecución son:

Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Recta 1 (formato y=m*x+b): y=-2.3*x+-2
Recta 2 (formato y=m*x+b): y=-2.30000001*x+2

Las dos rectas son PARALELAS
```

Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Recta 1 (formato y=m*x+b): y=-2.3*x+-2
Recta 2 (formato y=m*x+b): y=-2.30000001*x+-2

Las dos rectas son IGUALES
```

Ejemplo de ejecución 3 (en negrita, los datos que el usuario introduce):

```
Recta 1 (formato y=m*x+b): y=-2.3*x+-2
Recta 2 (formato y=m*x+b): y=-2.301*x+-5

Las dos rectas INTERSECAN en el punto
(-2999.50, 6896.85)
```

6. Sea el siguiente tipo de dato `tnum` que almacena la información de un número y la suma de sus dígitos:

```
typedef struct
{
  int num;         /* Número */
  int sum_dig;     /* Suma de los dígitos del número */
} tnum;
```

Escriba un programa en lenguaje C que lea del teclado una secuencia de números acabada en `-1` y muestre por pantalla el número de la secuencia que tiene el mayor valor de la suma de sus dígitos. En caso de empate, es decir, si hay varios números con el máximo valor de la suma de sus dígitos, ha de mostrar el primer número que aparezca en la secuencia.

A continuación, se proporcionan algunos ejemplos de ejecución:

Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Introduzca una secuencia de numeros acabada en -1: -1

Secuencia de numeros vacia
```

Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Introduzca una secuencia de numeros acabada en -1: 2234, 345,
66, 24, -1

El numero de la secuencia con una suma de digitos mayor es:
345 (suma de digitos = 12)
```

7. Dado el tipo de dato `tlibro` del problema 3, escriba un programa en lenguaje C que lea del teclado la información de `10` libros y muestre en pantalla toda la información del libro que tiene más de `150` páginas y fecha de publicación más antigua. En caso de empate, ha de mostrar el último libro introducido.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca la informacion de 10 libros:
(Formato isbn:fp_mes/fp_any:num_paginas)
929222222:12/2000:130
872372882:12/2000:151
323782833:11/1966:200
863826346:6/1988:256
553457882:6/1983:111
934545453:12/2000:183
844455531:5/2003:351
354531233:1/2005:325
```

```
666445343:11/1966:193
544334445:3/1998:89
El libro mas antiguo con mas de 150 paginas es:
ISBN: 666445343
Fecha de publicacion: 11/1966
Num. paginas: 193
```

6.6 Respuesta a los ejercicios propuestos

1.

```
typedef struct
{
  int ex1;    /* Extremo 1 */
  int ex2;    /* Extremo 2 */
} tficha;
```

2.

```
main()
{
  tficha f={6,6};

  printf("La ficha es: %d-%d\n", f.ex1, f.ex2);
}
```

Otra solución:

```
main()
{
  tficha f;

  f.ex1 = 6;
  f.ex2 = 6;
  printf("La ficha es: %d-%d\n", f.ex1, f.ex2);
}
```

3.

```
a. libro1.mes = 12;                  /* ERROR COMPILACIÓN */

b. libro1.fp = fecha1;               /* CORRECTA */

c. tfecha.anyo = 2013;               /* ERROR COMPILACIÓN */

d. fecha2.mes = m;                   /* CORRECTA */

e. if (2012 == fecha2.mes)
             m = 1;                 /* CORRECTA */

f. libro2.fp.mes = m + 3;            /* CORRECTA */

g. n = libro1.nump;                  /* CORRECTA */

h. if (fecha2 == libro1.fp)
             m = 1;                 /* ERROR COMPILACIÓN */
```

```
i. libro2 = libro1;                /* CORRECTA */

j. fecha1 = libro2.fp;             /* CORRECTA */

k. n = libro2.isbn + libro1.isbn; /* CORRECTA */

l. if (fecha1.mes == fecha2.mes)
             m = 1;                /* CORRECTA */

m. scanf("Publicacion: %d\n", &libro2.fp);
                                   /* ERROR COMPILACIÓN */

n.  libro1.tfecha = fecha2;        /* ERROR COMPILACIÓN */
```

4.

```
#include <stdio.h>

typedef struct
{
  unsigned int num;    /* Numerador */
  unsigned int den;    /* Denominador */
} tfraccion;

main()
{
  tfraccion f1, f2, res;
  unsigned int mcm, mcd;
  unsigned int a, b, r;

  printf("Fraccion 1 (formato x/y): ");
  scanf("%u/%u%*c", &f1.num, &f1.den);
  printf("Fraccion 2 (formato x/y): ");
  scanf("%u/%u%*c", &f2.num, &f2.den);

/* Calcula el mcm y el mcd de los denominadores de f1 y
   f2: mcm (a,b) = mcm (b, a%b) */
  a = f1.den;
  b = f2.den;
  r = a%b;
  while (r!=0)
  {
    a = b;
    b = r;
    r = a%b;
  }

  mcd = b;
  mcm = f1.den*f2.den/mcd;  /* mcm(a,b)=a*b/mcd(a,b); */

  printf("\nRESULTADO DE LAS OPERACIONES:\n");

  /* Suma */
  res.den = mcm;
  res.num=(res.den/f1.den)*f1.num+(res.den/f2.den)*f2.num;
  printf("\nSuma: %u/%u + %u/%u = %u/%u\n", f1.num,
          f1.den, f2.num, f2.den, res.num, res.den);
```

```
  /* Resta */
  res.num=(res.den/f1.den)*f1.num-(res.den/f2.den)*f2.num;
  printf("\nResta: %u/%u - %u/%u = %u/%u\n", f1.num,
          f1.den, f2.num, f2.den, res.num, res.den);

  /* Multiplicación */
  res.num = f1.num*f2.num;
  res.den = f1.den*f2.den;
  printf("\nMultiplicacion: %u/%u * %u/%u = %u/%u\n",
         f1.num, f1.den, f2.num, f2.den, res.num, res.den);

  /* División */
  res.num = f1.num*f2.den;
  res.den = f1.den*f2.num;
  printf("\nDivision: %u/%u / %u/%u = %u/%u\n", f1.num,
         f1.den, f2.num, f2.den, res.num, res.den);
}
```

5.

```
#include <stdio.h>
#define ERROR 0.00001

typedef struct
{
  float m;         /* Pendiente de la recta */
  float b;         /* Ordenada de origen */
} trecta;          /* Ecuación de la recta: y = m*x + b */

main()
{
  trecta r1, r2;
  float x, y;

  printf("Recta 1 (formato y=m*x+b): ");
  scanf("y=%f*x+%f%*c", &r1.m, &r1.b);
  printf("Recta 2 (formato y=m*x+b): ");
  scanf("y=%f*x+%f%*c", &r2.m, &r2.b);

  if (r1.m-ERROR<=r2.m && r2.m<=r1.m+ERROR &&
      r1.b-ERROR<=r2.b && r2.b<=r1.b+ERROR)
    printf ("\nLas dos rectas son IGUALES\n");
  else if (r1.m-ERROR<=r2.m && r2.m<= r1.m+ERROR)
    printf ("\nLas dos rectas son PARALELAS\n");
  else
  {
    x = (r2.b-r1.b)/(r1.m-r2.m);
    y = r1.m*x + r1.b;
    printf ("\nLas dos rectas INTERSECAN en el punto
             (%.2f, %.2f)\n", x, y);
  }
}
```

6.

```
#include <stdio.h>

typedef struct
{
  int num;        /* Número */
  int sum_dig;          /* Suma de los dígitos del número */
} tnum;

main()
{
  tnum el, mayor;
  int cociente;

  mayor.sum_dig = -1;

  printf("Introduzca una secuencia de números acabada
          en -1: ");
  scanf("%d%*c", &el.num);

  while (el.num!=-1)
  {
    el.sum_dig = 0;
    cociente = el.num;
    while (cociente!= 0)
    {
      el.sum_dig = el.sum_dig + cociente%10;
           cociente = cociente/10;
    }

    if (el.sum_dig>mayor.sum_dig)
      mayor = el;
    scanf("%d%*c", &el.num);
  }

  if (mayor.sum_dig==-1)
    printf ("\nSecuencia de numeros vacia\n\n");
  else
    printf ("\nEl numero de la secuencia con la suma de
            digitos mayor es: %d (suma de digitos = %d)\n",
            mayor.num, mayor.sum_dig);
}
```

7.

```
#include <stdio.h>
typedef struct
{
  int mes, anyo;
} tfecha;

typedef struct
{
  int isbn;     /* Número de ISBN (9 dígitos) */
  tfecha fp;    /* Fecha de publicación */
  int nump;     /* Número de páginas */
} tlibro;
```

```
main()
{
  tlibro el, antic;
  int i;

  antic.nump = -1; /* Identifica si se encuentra al
                      menos un libro con más de 150 pág. */
  printf("\nIntroduzca la informacion de 10 libros:\n");
  printf ("(Formato isbn:fp_mes/fp_any:num_paginas)\n");

  for (i=0; i<10; i++)
  {
    scanf("%d:%d/%d:%d%*c", &el.isbn, &el.fp.mes,
           &el.fp.anyo, &el.nump);
    if (el.nump>150)
    {
      if (antic.nump==-1)
        antic = el;  /* Primer libro con más de 150 pág. */
      else if (el.fp.anyo<antic.fp.anyo ||
              (el.fp.anyo==antic.fp.anyo &&
               el.fp.mes<=antic.fp.mes))
        antic = el;  /* El libro más antiguo hasta ahora */
    }
  }

  if (antic.nump == -1)
        printf("\nNo hay libros con mas de 150 paginas\n ");
  else
  {
    printf("\nEl libro mas antiguo con mas de 150 paginas
            es:\n");
    printf("ISBN: %d\nFecha de publicacion: %d/%d\n
            Num. paginas: %d\n", antic.isbn,
            antic.fp.mes, antic.fp.anyo, antic.nump);
  }
}
```

→7

Vectores

En los capítulos anteriores, se ha mostrado cómo utilizar y manipular variables de tipos de datos elementales (enteros, caracteres y reales) y de tipo estructura. Recuérdese que una estructura es un tipo de dato compuesto por varios elementos o campos, que pueden ser de tipos de datos diferentes (tipo de dato heterogéneo). En este capítulo, se explica un nuevo tipo de dato estructurado, el vector, donde todos los elementos son del mismo tipo (tipo de dato homogéneo). Se muestra cómo el programador puede crear el tipo de dato vector, y cómo puede declarar y manipular variables de este tipo. El capítulo finaliza mostrando algunos algoritmos básicos relacionados con los vectores.

7.1 Vectores

Como ya se ha comentado en el capítulo anterior, muchos programas requieren tipos de datos más complejos que los tipos elementales para su implementación. Estos tipos de datos son los tipos estructurados y están compuestos por varios elementos de igual o de distinto tipo. Como ya se ha visto, las estructuras permiten definir tipos de datos heterogéneos cuyos elementos o campos pueden ser de igual o de distinto tipo. Es el caso, por ejemplo, del DNI, formado por un entero y un carácter (número y letra del DNI).

En este capítulo, se muestra otro tipo de dato estructurado, pero que, a diferencia de las estructuras, es un tipo de dato homogéneo, es decir, donde todos los elementos son del mismo tipo. Este tipo de dato se denomina *vector* y es muy útil y necesario en programas en que un conjunto de variables del mismo tipo han de ser todas manipuladas siempre de la misma manera. A continuación se ilustra la utilidad de los vectores con un ejemplo.

Supóngase que se quiere escribir un programa que muestre por pantalla el número de veces que el usuario ha pulsado el número `0`, el `1`, el `2` y el `3` en una secuencia de números que finaliza con el valor `-1`. Por ejemplo, si la secuencia

es: `0, 2, 4, 12, 45, 3, 3, 67, 2, 23, 2, -1`, en pantalla saldrá: `contador0 = 1`, `contador1 = 0`, `contador2 = 3`, `contador3 = 2`.

Con lo que se ha visto hasta ahora, y sin conocer cómo se utilizan los vectores, el programa sería el siguiente:

```
#include <stdio.h>
main()
{
  int cont0, cont1, cont2, cont3, num;

  cont0 = 0;
  cont1 = 0;
  cont2 = 0;
  cont3 = 0;
  printf ("Introduzca una secuencia de numeros acabada en -1: ");
  scanf ("%d%*c", &num);

  while (num!=-1)
  {
    if (num==0)
      cont0 = cont0+1;
    else if (num==1)
      cont1 = cont1+1;
    else if (num==2)
      cont2 = cont2+1;
    else if (num==3)
      cont3 = cont3+1;
    scanf("%d%*c", &num);
  }

  printf("\ncontador0 = %d, contador1 = %d, contador2 = %d,
         contador3 = %d\n", cont0, cont1, cont2, cont3);
}
```

¿Cómo se debería modificar el programa anterior para contar ahora las ocurrencias de todos los números desde el `0` hasta el `100`? Es obvio que, con una implementación que incluya una variable para contar las ocurrencias de cada número, aunque factible, es inviable por la cantidad de líneas de código y de variables.

Obsérvese en el código anterior que las `4` variables `cont0`, `cont1`, `cont2` y `cont3` se utilizan exactamente de la misma manera:

- Las `4` variables son de tipo entero y se inicializan a cero.
- Las `4` variables se incrementan en función de una condición que es muy similar en todas ellas.
- Las `4` variables se muestran en pantalla.

Los vectores permiten agrupar estas cuatro variables del mismo tipo y con el mismo comportamiento en una única variable. De esta forma, en el programa

anterior puede utilizarse una variable de tipo vector (llamada `cont`) capaz de almacenar `4` valores de tipo entero, que serán referenciados individualmente utilizando un índice. La figura 7.1 muestra gráficamente cómo sería esa única variable `cont` que almacena el contador de 0, el contador de 1, el contador de 2 y el contador de 3.

Fig. 7.1
Representación gráfica de una variable de tipo vector

Con variables de tipo vector, se pueden implementar códigos como el anterior de una forma clara y sencilla. A lo largo de este capítulo, se verá cómo declarar y utilizar este tipo de variables.

7.2 Declaración de vectores

En el lenguaje de programación C, las variables de tipo *vector* se pueden declarar de dos maneras:

a) La primera consiste en declarar la variable directamente de tipo vector en el programa principal. Esta forma de declaración se asemeja a la declaración de variables de tipo elemental.

b) La segunda consiste en definir primero, fuera del programa principal y utilizando la construcción `typedef`, un tipo de dato vector. A continuación, en el programa principal, se declara la variable de tipo vector utilizando el tipo de dato definido previamente. Esta forma de declaración es similar a la que se ha utilizado para declarar estructuras.

A continuación, se analizan con detalle las dos alternativas anteriores.

Declaración directa

Para declarar una variable de tipo vector directamente en el programa principal, se debe especificar:

- El nombre de la variable.
- El tipo de dato de sus elementos: todos los elementos de un vector son del mismo tipo.
- La dimensión del vector: ha de ser un valor constante, conocido en el tiempo de compilación (antes de la ejecución del programa).

La sintaxis general para declarar una variable de tipo vector es la siguiente:

```
/* Declaración de una variable de tipo vector */
tipo_elemento nom_variable[dimension];

/* Declaración e inicialización de una variable de
   tipo vector */
tipo_elementos nom_variable[dimension]={val_primer_elemento,
                                        val_segundo_elemento,
                                        ...};
```

A continuación, se muestran algunos ejemplos de declaración de variables de tipo vector:

```
int cont[4];          /* Declara la variable cont como un
                         vector formado por 4 enteros */

#define DIM 80
char frase[DIM];      /* Declara la variable frase como un
                         vector de caracteres de dimensión DIM,
                         donde DIM está definido como una
                         constante con valor 80 */

tfecha vf[100];       /* Declara la variable vf como un
                         vector de fechas de dimensión 100.
                         Se asume que el tipo tfecha está
                         definido como se ha visto en el
                         capítulo 6, sección 6.2 */

int v[4]={0,0,0,0};   /* Declara e inicializa la variable
                         v como un vector formado por 4
                         enteros, inicializados todos a 0 */

tfecha lf[100]={{12,3,1969},{31,1,2000}};
                      /* Declara e inicializa la variable
                         lf como un vector formado por
                         100 fechas. En este caso, solo se
                         Inicializan las dos primeras fechas */
```

Obsérvese en el último ejemplo que no es necesario inicializar siempre el vector completo. Se puede inicializar parcialmente el vector, pero siempre serán las primeras posiciones. Así, en este último ejemplo, se ha declarado un vector `lf` de dimensión `100` y únicamente se han inicializado las posiciones `0` y `1` del vector.

Declaración a través de la construcción `typedef`

Otra alternativa para declarar variables de tipo vector consiste en definir primero un nuevo tipo de dato utilizando la misma construcción `typedef` que se utiliza para las estructuras (v. capítulo 6, sección 6.2).

La sintaxis general para definir un tipo de dato vector y asignarle un identificador es la siguiente:

```
typedef tipo_elemento nombre_tipo_vector[dimensión];
```

De esta manera, se define un nuevo tipo de dato que se referencia como `nombre_tipo_vector` y consiste en un vector de tamaño `dimensión` cuyos elementos son de tipo `tipo_elemento`.

Con respecto a las normas de estilo (v. anexo 9.B), y tal como se ha comentado en el capítulo anterior, el `nombre_tipo_vector` es un identificador que comienza con el carácter `t` o `T`, para identificar fácilmente que se trata del nombre del tipo de dato y no el de una variable.

Algunos ejemplos de definiciones de tipo de dato vector son los siguientes:

```
typedef int tcontador[4];
/* Define el tipo de dato tcontador como un vector de 4 enteros.
   tcontador es un nuevo tipo de dato, NO es una variable */

#define DIM 80
typedef char tfrase[DIM];
/* Define el tipo de dato tfrase como un vector de 80 caracteres.
   tfrase es un nuevo tipo de dato, NO es una variable */

typedef tfecha tvector_fechas[100];
/* Define el tipo de dato tvector_fechas como un vector de 100
   fechas. tvector_fechas es un nuevo tipo de dato, NO es una
   variable. Se asume que el tipo tfecha está definido como en la
   sección 6.2 */
```

Una vez definido el nuevo tipo de dato, se pueden declarar variables de este tipo.

La sintaxis general para declarar una variable de tipo vector es la siguiente:

```
/* Declaración de una variable de tipo vector */
nombre_tipo_vector  nom_variable;

/* Declaración e inicialización de una variable de tipo
   vector */
nombre_tipo_vector  nom_variable ={val_primer_elemento,
                                   val_segundo_elemento,
                                   ...};
```

A continuación, se muestran algunos ejemplos de declaración de variables asumiendo las definiciones de tipos anteriores:

```
tcontador cont;          /* Declara la variable cont como
                            un vector formado por 4 enteros */
```

```
tfrase frase;           /* Declara la variable frase como un
                           vector de caracteres de dimensión 80 */

tvector_fechas vf;      /* Declara la variable vf como un
                           vector de fechas de dimensión 100 */

tcontador v={0,0,0,0};  /* Declara e inicializa la variable
                           v como un vector formado por
                           4 enteros, inicializados todos a 0 */

tvector_fechas lf={{12,3,1969},{31,1,2000}};
                        /* Declara e inicializa la variable
                           lf como un vector formado por
                           100 fechas. En este caso, solo se
                           Inicializan las dos primeras fechas */
```

Finalmente, recuérdese que es en la declaración de variables cuando se reserva espacio en la memoria para almacenar los valores que tomarán las variables durante la ejecución del programa. Para las variables de tipo vector, se reservan en la memoria tantos bytes como los que ocupa el tipo de dato de un elemento multiplicado por la dimensión del vector. Así, por ejemplo, la variable `frase` del ejemplo anterior ocupa `100` bytes en la memoria (`100` elementos * `1` byte por elemento) y la variable `vf` ocupa `1200` bytes (`100` elementos * `12` bytes por elemento).

7.3 Operaciones con vectores

A continuación, se describen las operaciones que se pueden realizar con los vectores: acceso a un elemento concreto del vector, operaciones sobre un elemento y operaciones con vectores completos.

Acceso a un elemento del vector

Para acceder a un elemento concreto de un vector, se utiliza un índice que indica la posición del vector a la cual se quiere acceder. En el lenguaje de programación C, el índice de un vector empieza siempre en `0` (primera posición) y finaliza en la dimensión del vector menos `1` (última posición). Obsérvese que, en total, hay tantos elementos en el vector como su dimensión indica.

La sintaxis general para acceder un elemento concreto de un vector es:

```
nom_variable[índice]    /* Acceso a un elemento de un vector */
```

donde `nom_variable` es el nombre de una variable de tipo vector e `índice` es una expresión cuyo resultado es un valor entero.

Como ejemplo, suponiendo la declaración de variables de tipo vector del apartado anterior (v. sección 7.2), las expresiones siguientes para acceder a un elemento del vector son correctas:

```
cont[0]          /* Acceso al elemento de la posición 0 (primer
                    elemento) del vector cont */

frase[i+1]       /* Acceso al carácter de la posición i+1 del
                    vector frase. Se asume que i es una variable de
                    tipo entera y que el resultado de la
                    expresión i+1 es un valor entre 0 y 79 */

vf[34].dia       /* Acceso al campo dia de la fecha almacenada en
                    la posición 34 del vector vf */

v[3]             /* Acceso al elemento de la posición 3 del
                    vector v */

lf[v[3]].anyo    /* Acceso al campo anyo del elemento del vector lf
                    que está en la posición que indica v[3] */
```

Obsérvese en el último ejemplo que el índice del vector puede ser incluso el elemento de otro vector, siempre y cuando este sea de tipo entero.

La figura 7.2 muestra gráficamente las variables declaradas en los ejemplos de la sección 7.2, señalizando con una X el elemento al cual se accede con las expresiones anteriores. En el último ejemplo, obsérvese que estamos accediendo a la posición `0` del vector `lf` porque el índice `v[3]` vale `0`.

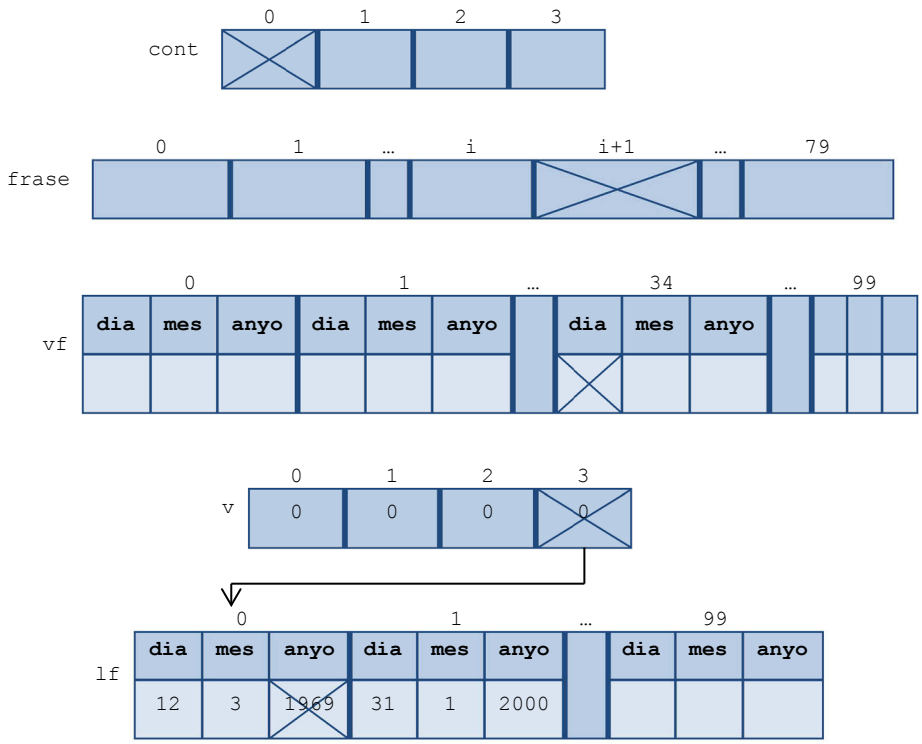


Fig. 7.2 Representación gráfica de variables de tipo vector y acceso a los elementos

Finalmente, cabe destacar que un error de ejecución bastante frecuente al manipular vectores es acceder fuera del rango del vector. Así, considerando las declaraciones anteriores, si se intenta acceder al elemento `cont[4]` o `frase[80]` o `vf[100]`, se está accediendo a posiciones inválidas del vector y habrá un error de ejecución.

Operaciones con un elemento del vector

Las operaciones que se pueden realizar con un elemento concreto de un vector son las correspondientes a su tipo. Por tanto, si el elemento es de tipo entero, se podrán realizar las mismas operaciones que se realizan con una variable de tipo entero (suma, resta, módulo, comparaciones, etc.); si es de tipo carácter, se podrán realizar todas las operaciones válidas para caracteres, etc.

Considerando los ejemplos de declaraciones de la sección 7.2, algunos ejemplos de códigos que operan con elementos de un vector son los siguientes:

Ejemplo 1:

```
cont[0] = 0;          /* Inicializa a 0 el elemento de la posición
                         0 (primer elemento) del vector cont */
```

Ejemplo 2:

```
if (frase[i]>= 'a' && frase[i]<= 'z')
  printf ("Es una letra minúscula\n");

/* Consulta si el carácter en la posición i del vector frase
   es un carácter en minúscula, es decir, si está entre la a-z.
   Supóngase que i es una variable entera que contiene un valor
   válido para acceder al vector frase */
```

Ejemplo 3:

```
if (vf[0].dia==12 && vf[0].mes==2 && vf[0].anyo==1900)
  printf("OK\n");
else
  printf("ERROR\n");

/* Consulta si la fecha almacenada en la posición 0 (primera
   fecha) del vector vf es el 12 de febrero de 1900 */
```

Ejemplo 4:

```
printf("Introduzca una fecha (dd-mm-aa): ");

scanf(("%d-%d-%d%*c", &lf[2].dia, &lf[2].mes, lf[2].anyo);

/* Lee del teclado una fecha y la almacena en la posición 2
   (tercer elemento) del vector lf */
```

Operaciones con vectores completos

A diferencia de las estructuras, el lenguaje C no permite realizar ninguna operación con vectores completos. Por tanto, si se quiere realizar una misma operación con todos los elementos de un vector, se debe recorrer todo el vector y aplicar la operación elemento a elemento.

Ejemplo: Supóngase que se quiere sumar dos vectores de números reales de dimensión `10` denominados `v1` y `v2`, que han sido previamente inicializados. El resultado de la suma se quiere dejar en un tercer vector, denominado `v3`.

Los tres vectores estarían declarados de la manera siguiente:

```
float v1[10]={1,2,3,4,5,6,7,8,9,10},
      v2[10]={2,4,6,8,0,1,3,5,7,9}, v3[10];
```

Para realizar la suma, el lenguaje C no permite hacer:

```
v3 = v1 + v2;       /* CÓDIGO INCORRECTO */
```

Hay que recorrer todos los elementos de los tres vectores y realizar la suma elemento a elemento. Esto se implementa con la sentencia iterativa `for` e indexando los vectores con la variable de control del bucle (variable `i`).

```
/* CÓDIGO CORRECTO */
for (i=0; i<10; i++)
   v3[i] = v1[i] +  v2[i];
```

De esta forma, después de realizar la operación, el vector `v3` tiene los elementos siguientes: `3, 6, 9, 12, 5, 7, 10, 13, 16, 19.`

7.4 Algoritmos básicos de vectores

En esta sección, se muestran algunos algoritmos básicos que manipulan vectores. En particular, se describen los algoritmos siguientes:

- Búsqueda de un elemento en un vector.
- Inserción de un elemento en un vector.
- Eliminación de un elemento del vector.
- Ordenación de los elementos de un vector.

Para algunos de estos algoritmos, se muestran dos versiones, en función de si el vector se encuentra o no previamente ordenado bajo algún criterio. Si se sabe que el vector ya está ordenado, algunos de estos algoritmos se pueden implementar de manera más eficiente.

Para simplificar la exposición de los algoritmos, se manipula una estructura de datos, llamada `tvector`, compuesta por dos campos: el campo `nelem`, que

indica el número de elementos almacenados en el vector, y el campo `vector`, que almacena como máximo `100` enteros. La definición de esta estructura de datos es la siguiente:

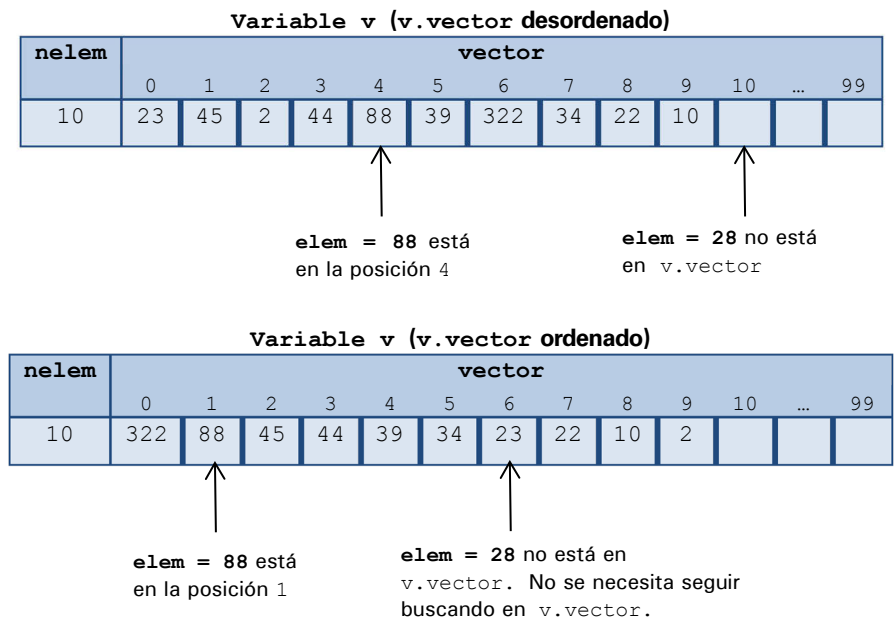
```
#define MAX 100
typedef struct
{
   int nelem;         /* Número de elementos en el vector */
   int vector[MAX];   /* Vector de enteros */
} tvector;
```

Búsqueda de un elemento en un vector

A continuación, se muestra el código para buscar un elemento concreto en un vector. Se busca únicamente la primera aparición del elemento (`elem`) en el vector (`v.vector`) y se hacen dos versiones diferentes de código, en función de si el vector está o no ordenado bajo algún criterio.

La figura 7.3 muestra un ejemplo de cómo se realiza la búsqueda del elemento `elem` en el vector `v.vector`, asumiendo dos valores distintos para `elem`, `88` y `28`. En el dibujo superior de la figura, `v.vector` está desordenado, mientras que en el dibujo inferior `v.vector` está ordenado de mayor a menor.

Fig. 7.3
Búsqueda de un elemento en un vector



```
/* Búsqueda (primera aparición) en un vector DESORDENADO */

main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int elem, i, encontrado;

  printf("Introduzca el elemento que buscas: ");
  scanf("%d%*c", &elem);

  encontrado = 0;
  i = 0;
  while (i<v.nelem && encontrado==0)
  {
    if (v.vector[i]==elem)
      encontrado = 1;
    else
      i = i+1;
  }

  if (encontrado==0)
    printf("Elemento NO encontrado\n");
  else
    printf("Elemento encontrado en la posicion %d\n", i);
}
```

El código anterior resaltado en negrita es la parte principal del código donde se realiza la búsqueda. Obsérvese que se utiliza la variable `encontrado` como una variable entera con valor `0` o con el valor `1`, que indica si se ha encontrado el elemento buscado. En cuanto se encuentra el elemento (`encontrado` valdrá `1`), se sale del `while` y no se sigue buscando. Por tanto, este algoritmo encuentra la primera aparición del elemento.

Nótese también que, para determinar que el elemento no se encuentra en el vector, es preciso recorrer el vector completo. Como el vector no está ordenado, no se puede afirmar que el elemento no está en el vector hasta que se hayan recorrido todas sus posiciones. Sin embargo, si se sabe que el vector está ordenado bajo algún criterio, se podría concluir que el elemento no se encuentra en el vector sin necesidad de recorrerlo todo. A continuación, se muestra la búsqueda de un elemento en un vector cuyos elementos están ordenados de mayor a menor.

```
/* Búsqueda (primera aparición) en un vector ORDENADO */

main()
{
  tvector v={10, {322, 88, 45, 44, 39, 34, 23, 22, 10, 2 }};
  int elem, i;

  printf("Introduzca el elemento que buscas: ");
  scanf("%d%*c", &elem);

  i = 0;
  while (i<v.nelem && v.vector[i]>elem)
```

```
    i = i+1;

  if (i<v.nelem && v.vector[i]==elem)
    printf("Elemento encontrado en la posicion %d\n", i);
  else
    printf("Elemento NO encontrado\n");
}
```

El código anterior resaltado en negrita es la parte principal del código donde se realiza la búsqueda en un vector ordenado. Obsérvese que, en cuanto se encuentra un elemento en el vector inferior o igual al que se está buscando, el bucle `while` finaliza (ya que la condición `v.vector[i]>elem` del `while` es falsa). Por tanto, no hará falta recorrer todos los elementos del vector para concluir que el elemento no está en el vector. Simplemente es necesario hallar un elemento que sea inferior al que buscamos.

Nótese también que, una vez finalizado el bucle `while`, no es suficiente que se cumpla la condición `v.vector[i]==elem` para afirmar que el elemento está en el vector. Primero es preciso asegurarse de que no se ha llegado al final del vector y que el índice `i` se encuentra entre `0` y `v.nelem-1`. Por eso, en la sentencia condicional `if` se tiene también en cuenta que se cumpla la condición `i<v.nelem`.

Obsérvese que si el orden de los elementos del vector fuese de menor a mayor, en lugar de mayor a menor, simplemente se tendría que cambiar en el programa anterior la condición `v.vector[i]>elem` del `while` por la condición `v.vector[i]<elem` del `while`.

Inserción de un elemento en un vector

A continuación, se muestra el código para insertar un elemento en un vector desordenado. Si el vector no sigue ningún criterio de ordenación, es suficiente con insertar el elemento en la primera posición libre del vector. La figura 7.4 muestra un ejemplo de cómo se inserta el elemento `elem` en el vector `v.vector`, asumiendo que `elem` vale `28`.

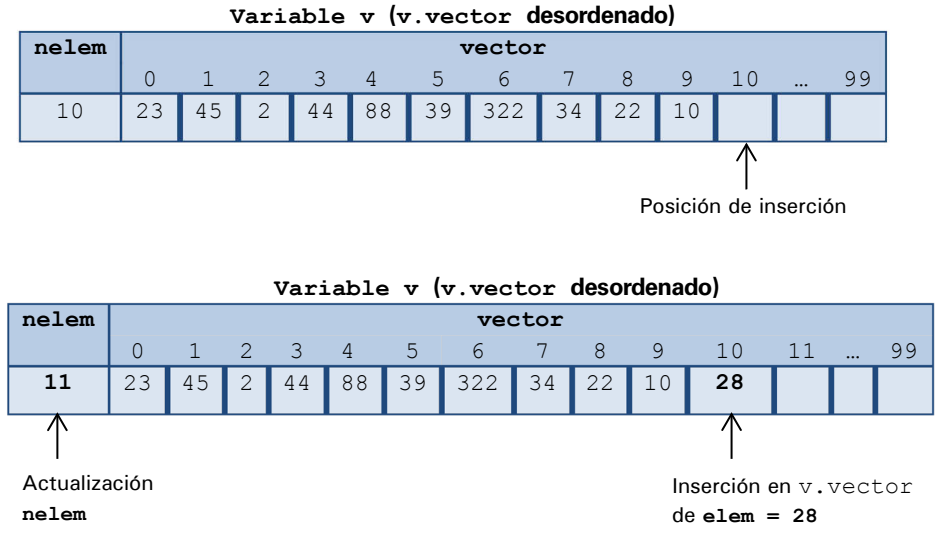


Fig. 7.4
Inserción de un elemento en un vector desordenado

```
/* Inserta un elemento en un vector DESORDENADO. La inserción se
   realiza en la primera posición libre del vector */
main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int elem, i;

  printf("Introduzca el elemento a insertar:");
  scanf("%d%*c", &elem);

  /* Se comprueba si el vector está lleno */
  if (v.nelem == MAX)
    printf ("No se pudo insertar. El vector esta lleno.\n");
  else
  {
    /* Se inserta el elemento al final */
    v.vector[v.nelem] = elem;
    v.nelem = v.nelem+1;
    printf("El elemento ha sido insertado en la ultima
            posicion\n");

    printf ("VECTOR: ");
    for (i=0; i< v.nelem-1; i++)
      printf("%d, ", v.vector[i]);
    printf("%d\n", v.vector[i]);
  }
}
```

El código anterior resaltado en negrita es la parte principal del código donde se realiza la inserción. Obsérvese que primero hay que asegurarse de que el elemento puede insertarse en el vector. Si el vector no tuviese posiciones libres, la inserción no se realizaría. Téngase en cuenta también que, cuando se inserta

un elemento en un vector, no solo hay que guardar el elemento en el campo `vector` de `v`, sino que también hay que actualizar el campo `nelem` para que la información en la estructura `v` sea coherente.

A continuación, se muestra el código para insertar un elemento en el vector `v.vector` suponiendo que los elementos del vector están ordenados de mayor a menor. El programa realiza la inserción del elemento en el vector manteniendo el orden de los elementos del vector.

La figura 7.5 muestra gráficamente un ejemplo de cómo se realiza la inserción suponiendo que `elem` vale `28`.

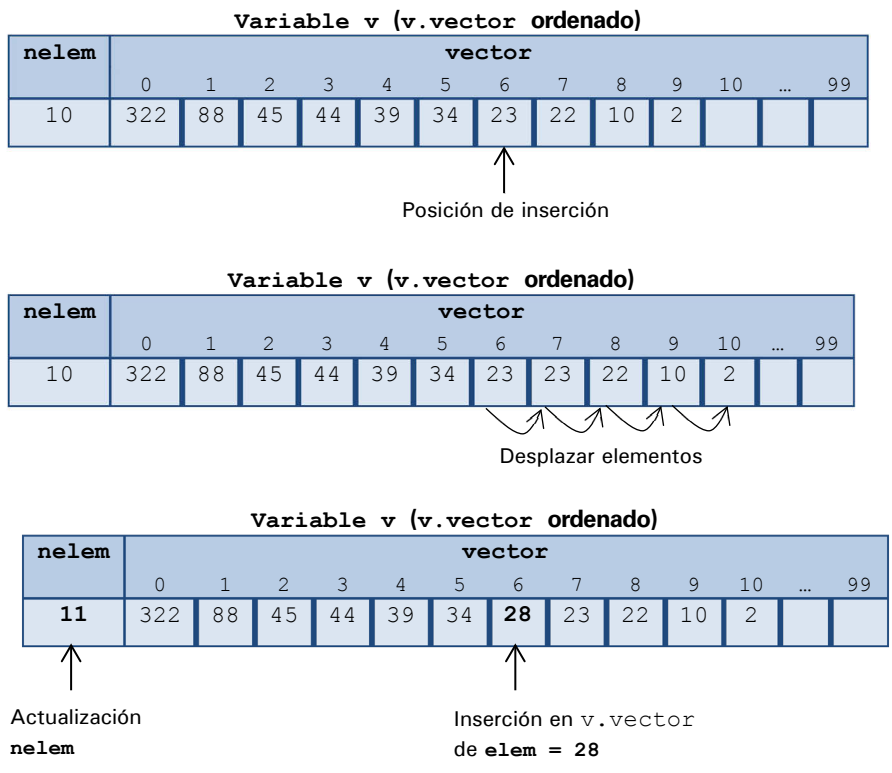


Fig. 7.5
Inserción de un elemento en un vector ordenado

```
/* Inserta un elemento en un vector ORDENADO de mayor a menor.
   La inserción del elemento se realiza manteniendo el orden
   de los elementos del vector */
main()
{
  tvector v={10, {322, 88, 45, 44, 39, 34, 23, 22, 10, 2}};
  int elem, i, pos;

  printf("Introduzca el elemento a insertar:");
  scanf("%d%*c", &elem);

  /* Se comprueba si el vector está lleno */
```

```
  if (v.nelem == MAX)
    printf ("No se pudo insertar. El vector esta lleno.\n");
  else
  {
    /* Se busca la posición a insertar para mantener el orden */
    i = 0;
    while (i<v.nelem && v.vector[i]>elem)
      i = i+1;

    pos = i;  /* Posición donde hay que insertar el elemento */

    /* Se desplazan los elementos una posición a la derecha */
    for (i=v.nelem-1; i>=pos; i--)
      v.vector[i+1] = v.vector[i];

    /* Se inserta el elemento */
    v.vector[pos] = elem;
    v.nelem = v.nelem+1;
    printf("El elemento ha sido insertado\n");

    printf ("VECTOR: ");
    for (i=0; i< v.nelem-1; i++)
      printf("%d, ", v.vector[i]);
    printf("%d\n", v.vector[i]);
  }
}
```

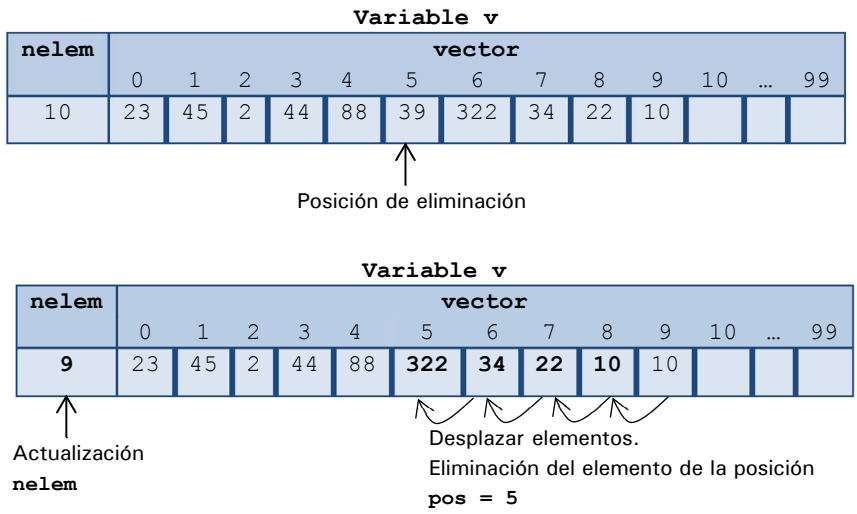
El código anterior resaltado en negrita es la parte principal del código donde se realiza la inserción de un elemento en un vector ordenado. Obsérvese que primero es preciso asegurarse de que el elemento puede incluirse en el vector. A continuación, se busca la posición donde se debe insertar el elemento para mantener el orden (variable `pos`). Después, se desplazan todos los elementos una posición a la derecha empezando por el último elemento en el vector y finalizando con el elemento de la posición donde se quiere insertar. Finalmente, se inserta el elemento en la posición correspondiente y se incrementa la cantidad de elementos del vector.

Eliminación de un elemento del vector

Cuando se quiere eliminar un elemento de un vector, simplemente es necesario conocer la posición donde se encuentra este elemento en el vector para eliminarlo. Si la posición es válida, es decir, está dentro del rango `[0,v.nelem-1]`, se puede eliminar el elemento.

La figura 7.6 muestra gráficamente un ejemplo de cómo se realiza la eliminación, suponiendo que se quiere eliminar el elemento de la posición `5` del vector.

Fig. 7.6
Eliminación de un elemento de un vector



Seguidamente, se muestra el código para realizar la eliminación de un elemento en un vector. Obsérvese que este programa es independiente del orden que tienen los elementos en el vector.

```
/* Elimina un elemento del vector */
main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int i, pos;

  printf("Introduzca la posicion del elemento a eliminar:");
  scanf("%d%*c", &pos);

  /* Se comprueba si la posición es válida */
  if (pos<0 || pos>=v.nelem)
    printf ("No se pudo eliminar. Posicion no valida.\n");
  else
  {
    /* Eliminar elemento del vector */
    for (i=pos; i<v.nelem-1; i++)
      v.vector[i] = v.vector[i+1];
    v.nelem = v.nelem-1;
    printf("El elemento ha sido eliminado\n");

    printf ("VECTOR: ");
    for (i=0; i< v.nelem-1; i++)
      printf("%d, ", v.vector[i]);
    printf("%d\n", v.vector[i]);
  }
}
```

Obsérvese que los desplazamientos de los elementos en el vector para eliminar el elemento se realizan en sentido contrario al que se ha realizado en

la inserción ordenada. Ahora el desplazamiento empieza en `pos` hasta `v.nelem-2`, mientras que en la inserción ordenada empezaba en `v.nelem-1` hasta `pos`.

Téngase en cuenta también que, cuando se elimina un elemento de un vector, hay que decrementar el campo `nelem` para que la información en la estructura `v` sea correcta.

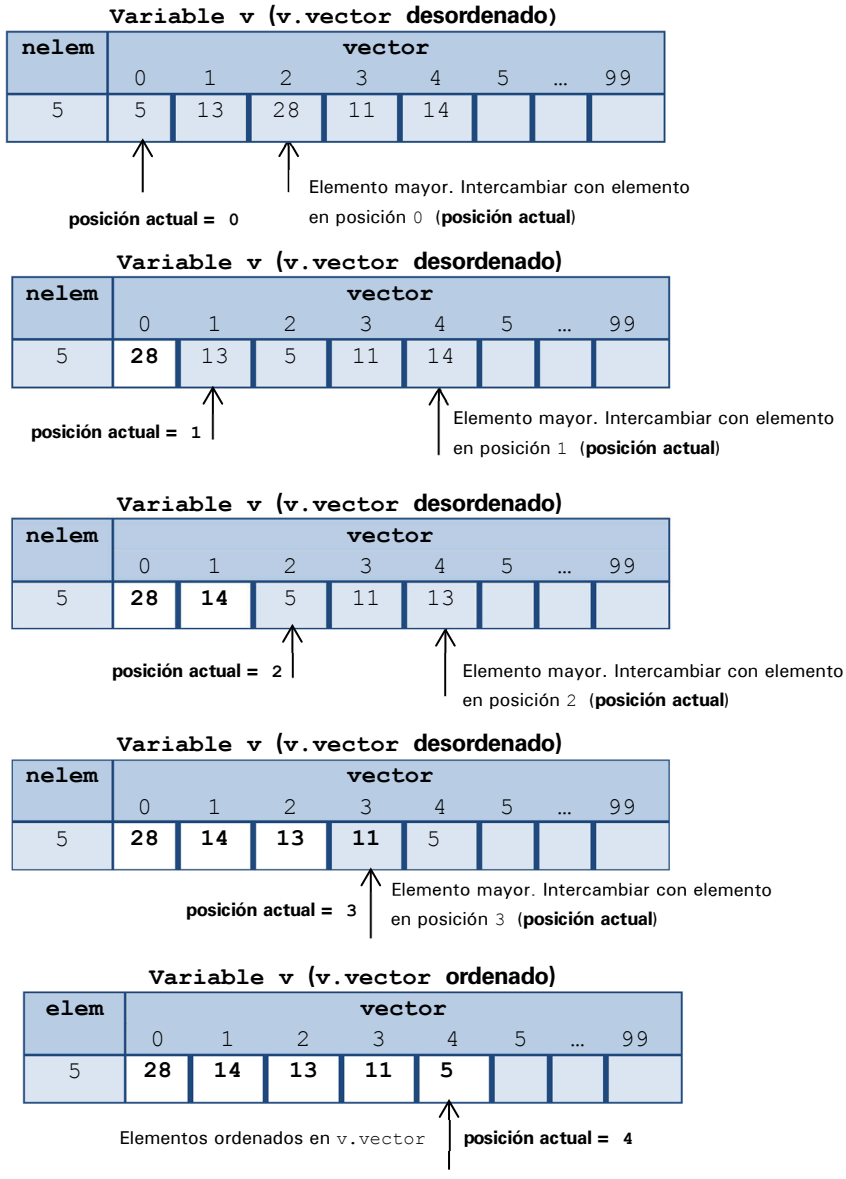
Ordenación de los elementos de un vector

Finalmente, se muestra un algoritmo para ordenar de mayor a menor los elementos de un vector de enteros. Existen diversos algoritmos de ordenación, entre los cuales cabe mencionar: selección directa, inserción directa, burbuja, *quicksort*, etc. En este libro, se ve solamente el algoritmo de selección directa, puesto que se considera el algoritmo de ordenación más sencillo en comparación con el resto. Este algoritmo consiste en repetir los pasos siguientes:

- Buscar el elemento mayor del vector entre los elementos siguientes al que ocupa la posición *actual*. La posición *actual* corresponde a la cantidad de elementos en el vector que ya se han ordenado en pasos anteriores.
- Intercambiar el elemento mayor con el elemento que ocupa la posición *actual* del vector.
- Después de estos dos pasos, ya se ha ordenado un elemento del vector por lo que se incrementa la posición *actual* en 1.
- Repetir los pasos anteriores hasta que la posición *actual* sea igual al número de elementos del vector menos `1` (`v.nelem-1`).

La figura 7.7 muestra un ejemplo de funcionamiento de este algoritmo, suponiendo un vector con los elementos siguientes: `5, 13, 28, 11, 14.`

Fig. 7.7
Ejemplo de ordenación por el método de la selección directa



A continuación, se muestra el programa para ordenar los elementos de un vector de mayor a menor.

```
/* Ordena los elementos del vector de mayor a menor */
main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int aux, posmax, i, j;
```

```
  printf("Vector original: ");
  for (i=0; i<v.nelem-1; i++)
    printf ("%d - ", v.vector[i]);
  printf ("%d\n", v.vector[i]);

  /* Ordenación por el método de la selección directa */
  for (i=0; i<v.nelem-1; i++)
  {
   /* Se busca la posición del elemento mayor desde i
      a v.nelem-1 */
      posmax = i;
      for (j=i+1; j<v.nelem; j++)
        if (v.vector[j]>v.vector[posmax])
          posmax = j;

      /* Se intercambia la posición i y posmax */
      aux = v.vector[i];
      v.vector[i] = v.vector[posmax];
      v.vector[posmax] = aux;
   }

   printf("Vector ordenado: ");
   for (i=0; i<v.nelem-1; i++)
     printf ("%d - ", v.vector[i]);
   printf ("%d\n", v.vector[i]);
}
```

Obsérvese que si se quisiera cambiar el criterio de ordenación de los elementos del vector para ordenarlos de menor a mayor, simplemente se tendría que cambiar la condición `(v.vector[j]>v.vector[posmax])` de la sentencia condicional `if` por la condición `(v.vector[j]<v.vector[posmax])`.

Por último, nótese que adaptar los algoritmos anteriores para manipular vectores de otro tipo de datos es muy sencillo y se propone como ejercicio para el lector (v. sección 7.6).

7.5 Ejemplos de uso de vectores

A continuación, se proporcionan varios ejemplos de uso de vectores.

Ejemplo 1

En primer lugar, se muestra cómo implementar el programa planteado en la sección 7.1 utilizando vectores. El programa consiste en escribir un código que muestre por pantalla el número de veces que el usuario introduce cada uno de los números del `0` al `100` (ambos inclusive) en una secuencia de números que finaliza con el valor `-1`. A continuación, se muestra un ejemplo de ejecución del programa.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca una secuencia de numeros acabada en -1: 0 2 4 12 45 3
3 67 2 23 2 -1
Valor de los contadores:
contador0 = 1
contador2 = 3
contador3 = 2
contador4 = 1
contador12 = 1
contador23 = 1
contador45 = 1
contador67 = 1
```

Obsérvese que sólo se muestran por pantalla los contadores con valor mayor o igual a `1` (contadores asociados a los números de la secuencia), y no los contadores con valor `0`.

La figura 7.8 representa gráficamente la variable `cont` que se declarará y manipulará en el programa.

Fig. 7.8 Representación gráfica de la variable `cont`

El código del programa es:

```
#include <stdio.h>
main()
{
  int cont[101]; /* Almacena los contadores del 0 al 100 */
  int num, i;
  /* Inicializar a 0 el vector de contadores */
  for (i=0; i<=100; i++)
     cont[i] = 0;
  printf ("Introduzca una secuencia de numeros acabada en -1:");
  scanf ("%d", &num);
  /* Leer la secuencia y actualizar contadores */
  while (num!=-1)
  {
    if (num>=0 && num<=100)
      cont[num] = cont[num]+1;

    scanf("%d", &num);
  }
  scanf("%*c");
  /* Mostrar contadores por pantalla */
```

```
  printf("\nValor de los contadores:\n");
  for (i=0; i<=100; i++)
    if (cont[i]!=0)
      printf("contador%d = %d\n", i, cont[i]);
}
```

Obsérvese que la variable `cont` es de dimensión `101`. Es preciso guardar el número de las apariciones de todos los valores entre `0` y `100` (ambos inclusive); por tanto, se necesitan `101` contadores. En la posición `0` del vector `cont`, se almacena el número de apariciones del `0`; en la posición `1` del vector, las apariciones del `1`, y así sucesivamente, hasta la posición `100` del vector, que almacena las apariciones del `100`.

Obsérvese que la sentencia condicional `if` dentro del `while` está garantizando que solamente se accede a posiciones válidas del vector, es decir, a posiciones entre la `0` y la `100`.

Ejemplo 2

A continuación, se muestra un código que aparece muy frecuentemente en programas que manipulan cadenas de caracteres. El código consiste en leer del teclado una frase acabada en el carácter punto (`.`), almacenar la frase en la variable `fr` de tipo vector y, finalmente, mostrar en pantalla la frase leída. La frase puede tener, como máximo `100` caracteres. A continuación, se proporciona un ejemplo de ejecución del programa.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca una frase acabada en '.':  Esto es un ejemplo.
Esto es un ejemplo.
```

La figura 7.9 muestra cómo ha de quedar almacenada la frase introducida en la variable `fr`.

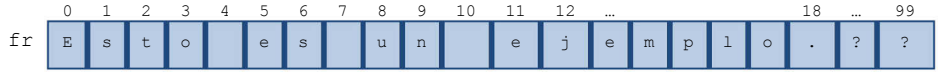


Fig. 7.9
Representación gráfica de la variable `fr`

Téngase en cuenta que es necesario identificar dónde acaba la frase para poder recorrer posteriormente solo las posiciones del vector que almacenan caracteres de la frase. En este ejemplo, se identifica con el carácter punto (`.`) que se almacenará al final de la frase, pero se podría identificar con otros caracteres (típicamente se utilizan los caracteres `.`, `\n` y `\0`) o, incluso, con una variable entera que almacene la longitud de la frase.

El código del programa es:

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

```
#define MAXCAR 100
typedef char tfrase[MAXCAR];

main()
{
  tfrase fr;            /* Almacena la frase */
  int  i;


  /* Se lee la frase del teclado */
  i = 0;
  printf("Introduzca una frase acabada en '.': ");
  scanf("%c", &fr[i]);
  while (i<MAXCAR-1 && fr[i]!='.')
  {
    i = i+1;
    scanf("%c", &fr[i]);
  }

  scanf("%*c");

  /* Si la frase tiene más de MAXCAR caracteres, se deja de
     leer y se guarda el carácter punto en la última posición
     del vector */
  if (i==MAXCAR-1)
    fr[i] = '.';

  /* Se muestra la frase en pantalla */
  for (i=0; fr[i]!='.'; i++)
    printf("%c", fr[i]);
  printf("%c\n", fr[i]);    /* Muestra el caracter punto en
                               pantalla */
}
```

En este código, se ha definido primero el tipo de dato `tfrase` como vector de caracteres y posteriormente se ha declarado la variable `fr` de este tipo.

Obsérvese que, durante la lectura de la frase, se controla que no se accede a ninguna posición fuera del vector. La condición `i<MAXCAR-1` de la sentencia `while` controla que solo se acceda a posiciones válidas del vector (posiciones entre `0` y `MAXCAR-1`). En el caso de que la frase tenga más de `100` caracteres, el programa solamente lee los `99` primeros y almacena un `'.'` en la última posición del vector (`fr[MAXCAR-1]`).

Ejemplo 3

A continuación se muestra un programa que manipula un vector de estructuras. En concreto, se quiere implementar un programa que, dado un vector inicializado con `10` DNI, muestre en pantalla todos los DNI que tienen una letra determinada, que se pedirá al usuario por teclado. A continuación, se eliminan del vector todos los DNI que tienen esta letra y se muestra el vector por pantalla con los DNI correspondientes eliminados.

Si, por el contrario, no se elimina ningún DNI del vector, ya que no hay DNI que tengan la letra introducida, el programa mostrará en pantalla un mensaje para indicar que no se ha encontrado ningún DNI con esta letra.

A continuación, se muestran dos ejemplos de ejecución del programa, suponiendo que el vector con los `10` DNI ha sido inicializado como sigue: `11111111A, 63282991R, 46382917L, 22222222C, 12345678T, 33333333S, 16399222L, 97532987E, 98765432A, 46382917A.`

Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Introduzca la letra de los DNI que buscas: D

DNI con letra D:
No se ha encontrado ningun DNI con la letra D
```

Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Introduzca la letra de los DNI que buscas: A

DNI con letra A:
11111111A
98765432A
46382917A

Vector sin los DNI con letra A:
63282991R
46382917L
22222222C
12345678T
33333333S
16399222L
97532987E
```

La figura 7.10 muestra gráficamente la variable `v` que se declara, inicializa y manipula en el programa. Esta variable guardará en el campo `ndnis` el valor `10` (cantidad de DNI) y en el campo `vdni`, la información de los DNI (número y letra).

Variable v

ndnis	**vdni**						
	0		1		…	9	
	num	**letra**	**num**	**letra**		**num**	**letra**
10	11111111	'A'	63282991	'R'	…	46382917	'A'

Fig. 7.10
Representación gráfica de la variable `v`

El código del programa es:

```
#include <stdio.h>
#define NUMDNI 10
typedef struct
{
  int num;              /* Número del DNI */
```

```
  char letra;           /* Letra del DNI */
} tdni;

typedef struct
{
  int  ndnis;           /* Número de DNI en el vector */
  tdni vdni[NUMDNI];    /* Vector de DNI */
} tvector_dni;


main()
{
  tvector_dni v={10,{{11111111, 'A'}, {63282991, 'R'},
                {46382917, 'L'},{22222222, 'C'}, {12345678, 'T'},
                {33333333, 'S'},{16399222, 'L'},{97532987, 'E'},
                {98765432, 'A'}, {46382917, 'A'}}};

  int  i, j, cont;
  char letra;

  printf("Introduzca la letra de los DNI que buscas: ");
  scanf("%c%*c", &letra);
  printf("\nDNI con letra %c:\n", letra);
  cont = 0;
  i = 0;
  while (i<v.ndnis)
  {
    if (v.vdni[i].letra==letra)
    {
      printf("%d%c\n", v.vdni[i].num, v.vdni[i].letra);
      cont = cont + 1;

      /* Se elimina DNI del vector */
      for (j=i; j<v.ndnis-1; j++)
        v.vdni[j] = v.vdni[j+1];
      v.ndnis = v.ndnis-1;
    }
    else
      i = i+1;
  }
  if (cont==0)/* No se ha encontrado ningún DNI con esta letra */
    printf("No se ha encontrado ningun DNI con la
            letra %c\n", letra);
  else
  {
    /* Se muestra el nuevo vector */
    printf("\nVector sin los DNI con letra %c:\n", letra);
    for (i=0; i<v.ndnis; i++)
      printf("%d%c\n", v.vdni[i].num, v.vdni[i].letra);
  }
}
```

Ejemplo 4

Finalmente, se muestra otro ejemplo más complejo donde el tipo de datos manipulados es una estructura y la estructura, a su vez, contiene como campo un vector de estructuras. En particular, se escribirá un programa que pida por

teclado la cantidad de personas de las cuales se quiere leer información y, para cada una de ellas, el programa pida el nombre, los apellidos y la edad. Luego, el programa calculará la edad media de esas personas y mostrará por pantalla el nombre, los apellidos y la edad de las personas que tengan una edad inferior o igual a la media. Se supondrá que, como mínimo, el usuario introducirá la información de una persona. A continuación, se muestra un ejemplo de ejecución.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Numero de personas a introducir datos: 6

Introduzca los datos de las 5 personas:
Nombre de la persona 0: Elena Villa
Edad de la persona 0: 15
Nombre de la persona 1: Victor Rodriguez
Edad de la persona 1: 30
Nombre de la persona 2: Mar Vieira
Edad de la persona 2: 22
Nombre de la persona 3: Manuel Hernandez
Edad de la persona 3: 54
Nombre de la persona 4: Maria Gomez
Edad de la persona 4: 1
Nombre de la persona 5: Joan Alsina
Edad de la persona 5: 67

La edad media de las personas es 31.50 (31 anyos y 6 meses)

Personas con edad inferior o igual a la media:
Elena Villa (15 anyos)
Victor Rodriguez (30 anyos)
Mar Vieira (22 anyos)
Maria Gomez (1 anyos)
```

La figura 7.11 representa gráficamente la variable `v`, que se declara y manipula en el programa. Esta variable guardará, en el campo `nper`, el número de personas y, en el campo `vp`, la información leída de cada persona (nombre, apellidos y edad).

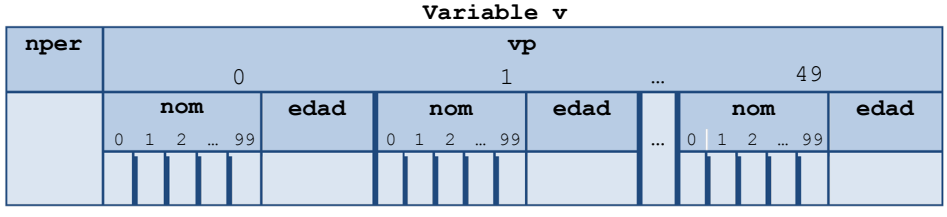


Fig. 7.11
Representación gráfica de la variable `v`

El código del programa es:

```
#include <stdio.h>
#define MAXCAR 100
```

```
#define NUMPERSONAS 50

typedef char tcadena[MAXCAR];

typedef struct
{
  tcadena nom;         /* Almacena nombre y apellidos */
  int edad;            /* Almacena la edad de la persona */
} tpersona;

typedef struct
{
  int nper;                  /* Número de personas en el vector */
  tpersona vp[NUMPERSONAS]; /* Vector de personas */
} tvector_per;

main()
{
  tvector_per v;
  int  i,j, anyos, meses;
  char c;
  float edad_media;

  printf("\nNumero de personas a introducir datos: ");
  scanf("%d%*c", &v.nper);

  /* Se lee del teclado la información de las personas */
  printf("\nIntroduzca los datos de las %d personas:\n",
          v.nper);
  for (j=0; j<v.nper; j++)
  {
    printf("Nombre persona %d: ", j);
    i = 0;
    scanf("%c", &v.vp[j].nom[i]);
    while (i<MAXCAR-1 && v.vp[j].nom[i]!='\n')
    {
       i = i+1;
       scanf("%c", &v.vp[j].nom[i]);
    }

    /* Si nombre y apellidos tiene más de MAXCAR caracteres, se
       continua leyendo todos los caracteres del teclado hasta
       \n, pero no se almacena en el vector. Finalmente, se
       almacena el carácter \n en la última posición del
       vector */
    if (i==MAXCAR-1)
    {
       c = v.vp[j].nom[i];
       while (c!='\n')
         scanf("%c", &c);
       v.vp[j].nom[i] = '\n';
    }
```

```
    printf("Edad persona %d: ", j);
    scanf("%d%*c", &v.vp[j].edad);
  }

  /* Se calcula la edad media de las personas */
  edad_media = 0.0;
  for (i=0; i<v.nper; i++)
    edad_media = edad_media + v.vp[i].edad;

  edad_media = edad_media/v.nper;

  /* Se asigna a la variable anyos la parte entera de
     edad_media */
  anyos = (int) edad_media;
  meses = (edad_media-anyos)*12;
  if ((edad_media-anyos)*12-meses>=0.5)
      meses = meses + 1;
  printf ("\nLa edad media de las personas es %.2f (%d
           anyos y %d meses)\n",edad_media, anyos, meses);

 /* Se muestra por pantalla el nombre, apellidos y edad de
    las personas con edad inferior o igual a la media */
  printf ("\nPersonas con edad inferior o igual a
            la media:\n");
  for (j=0; j<v.nper; j++)
  {
    if (v.vp[j].edad <= edad_media)
    {
      for (i=0; v.vp[j].nom[i]!='\n'; i++)
        printf ("%c", v.vp[j].nom[i]);
      printf(" (%d anyos)\n", v.vp[j].edad);
    }
  }
}
```

7.6 Ejercicios

1. Escriba un programa en lenguaje C que lea dos palabras acabadas en `'\n'` e indique mostrando un mensaje por pantalla, si las palabras son iguales o no. Suponga que las palabras tienen, como máximo, `30` caracteres.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

   ```
   Introduzca la primera palabra: Martin
   Introduzca la segunda palabra: Martinez

   Las palabras son diferentes
   ```

2. Escriba el trozo de código que falta del programa siguiente para que invierta los elementos de un vector de enteros. Suponga que, como máximo, el vector tiene `100` elementos.

```
#include <stdio.h>
#define MAXELEM 100
typedef struct
{
  int nelem;
  int vector[MAXELEM];
} tvector;

main()
{
  tvector v;
  int i, j, aux;

  /* Leer el vector */
  printf("Introduzca el numero de elementos del
          vector: ");
  scanf("%d%*c", &v.nelem);

  printf("Introduzca los elementos del vector: ");
  for (i=0; i<v.nelem; i++)
    scanf("%d%*c", &v.vector[i]);

  /* ESCRIBA AQUÍ EL CÓDIGO QUE FALTA */

  /* Mostrar el vector */
  printf("El vector invertido es: ");
  for (i=0; i<v.nelem-1; i++)
     printf("%d, ", v.vector[i]);
  printf("%d\n", v.vector[i]);
}
```

Observe que primero se pide por teclado al usuario que introduzca el número de elementos del vector y, a continuación, que introduzca los elementos del vector de enteros.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca el numero de elementos del vector: 8
Introduzca los elementos del vector: 7, 15, 34, 36, 2, 999,
45, 34

El vector invertido es: 34, 45, 999, 2, 36, 34, 15, 7
```

3. Escriba un programa en lenguaje C que determine cuántas palabras palíndromas hay en una frase acabada en `'.'`. Tenga en cuenta que, entre palabras, puede haber más de un espacio o signos de puntuación.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca una frase acabada en punto:
Ellos pueden reconocer a Ana,   cuando narran la historia.

Palabras palindromas:
reconocer
a
Ana
```

```
narran
En total hay 4 palabras palindromas.
```

4. Escriba un programa en lenguaje C que lea del teclado los elementos de un vector de enteros desordenado y que, a medida que los lea, los vaya dejando ordenados de mayor a menor en el vector. Para realizar esto no utilice el algoritmo de ordenación, sino los algoritmos de búsqueda y de inserción ordenada (v. sección 7.4) para mantener el vector ordenado.

   Una vez leído el vector, se han de eliminar todos los elementos que son pares. El programa mostrará en pantalla el vector ordenado antes y después de eliminar estos elementos. El programa pedirá al usuario que introduzca desde el teclado el número de elementos del vector. Suponga que, como máximo, el vector tiene `100` elementos.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

   ```
   Introduzca el numero de elementos del vector (MAX. 100): 10
   Introduzca los elementos del vector: 23, 34, 11, 677, 12, 33,
   55, 97, 88, 102
   Vector ordenado: 677, 102, 97, 88, 55, 34, 33, 23, 12, 11
   Vector ordenado sin numeros pares: 677, 97, 55, 33, 23, 11
   ```

5. Dada la estructura de datos siguiente, definida para representar polinomios de grado inferior o igual a `8`:

   ```
   #define MAX_GRADO 8
   typedef struct
   {
     int grado;                /* Grado del polinomio */
     float coef[MAX_GRADO+1];  /* Coeficientes */
   } tpolinomio;
   ```

   Escriba un programa en lenguaje C que permita sumar `10` polinomios introducidos por teclado. El formato de entrada para leer los polinomios es el siguiente: `coefxgrado coefxgrado...` Cada uno de los polinomios finaliza con el carácter `\n` y sus coeficientes no están ordenados por el grado del término del polinomio.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

   ```
   Introduzca los polinomios (coefxgrado ...) finalizados en
   '\n':
   Polinomio 0: 3x2 4x0
   Polinomio 1: 4x5 2x3 5.2x4
   Polinomio 2: 4x2 3x0
   Polinomio 3: 3.4x0
   Polinomio 4: -4x5 3x4 2x1
   Polinomio 5: 3x3
   Polinomio 6: 78x0
   Polinomio 7: 3x4 3x2 3.2x0
   ```

```
Polinomio 8: 45x1
Polinomio 9: 32.2x1 33x0

El polinomio suma es de grado 4:
124.60x0  79.20x1  10.00x2  5.00x3  11.20x4
```

6. Escriba un programa en lenguaje C que, dado un vector de matrículas de vehículos previamente inicializado, las ordene primero alfabéticamente por las letras y, en caso de empate, las ordene por el número de matrícula de menor a mayor. Utilice el algoritmo de selección directa para la ordenación que se ha visto en la sección 7.4. El programa debe mostrar por pantalla las matrículas ordenadas.

   Ejemplo de ejecución (suponiendo que el vector está inicializado con la información siguiente: `v={10, {{2343,"BZG"}, {2343,"FKD"}, {1203,"KAS"}, {2666,"AHJ"}, {2300,"BGZ"}, {2343,"AHJ"}, {3433,"BZG"}, {5677,"CZF"}, {8766,"BZZ"}, {0001,"BZG"} }};`

```
Vector de matriculas original:
2343 BZG
2343 FKD
1203 KAS
2666 AHJ
2300 BGZ
2343 AHJ
3433 BZG
5677 CZF
8766 BZZ
0001 BZG

Vector de matriculas ordenado:
2343 AHJ
2666 AHJ
2300 BGZ
0001 BZG
2343 BZG
3433 BZG
8766 BZZ
5677 CZF
2343 FKD
1203 KAS
```

7.7 Respuesta a los ejercicios propuestos

1.

```
#include <stdio.h>
#define MAXCAR 100

main()
{
  char pal1[MAXCAR], pal2[MAXCAR];
  int  i;
  char c;

  /* Se lee la primera palabra */
```

```
  i = 0;
  printf("Introduzca la primera palabra: ");
  scanf("%c", &pal1[i]);
  while (i<MAXCAR-1 && pal1[i]!='\n')
  {
    i = i+1;
    scanf("%c", &pal1[i]);
  }

  /* Si la palabra tiene más de MAXCAR caracteres, se
     continua leyendo los caracteres del teclado hasta
     el '\n', pero estos caracteres no se almacenan en
     la palabra. Finalmente, se almacena el carácter \n
     en la última posición de la palabra */
  if (i==MAXCAR-1)
  {
    c = pal1[i];
    while (c!='\n')
      scanf("%c", &c);
    pal1[i] = '\n';
  }

  /* Se lee la segunda palabra */
  i = 0;
  printf("Introduzca la segunda palabra: ");
  scanf("%c", &pal2[i]);
  while (i<MAXCAR-1 && pal2[i]!='\n')
  {
    i = i+1;
    scanf("%c", &pal2[i]);
  }

  if (i==MAXCAR-1)
  {
    c = pal2[i];
    while (c!='\n')
      scanf("%c", &c);
    pal2[i] = '\n';
  }

  /* Se determina si son iguales */
  i = 0;
  while (pal1[i]==pal2[i] && pal1[i]!='\n')
    i = i+1;

  if (pal1[i]==pal2[i])
    printf ("\nLas palabras son iguales\n");
  else
    printf ("\nLas palabras son diferentes\n");
}
```

2.

```
#include <stdio.h>
#define MAXELEM 100
```

```
typedef struct
{
   int nelem;
   int vector[MAXELEM];
} tvector;

main()
{
  tvector v;
  int i, j, aux;

  /* Se lee el vector */
  printf("Introduzca el numero de elementos del vector: ");
  scanf("%d%*c", &v.nelem);

  printf("Introduzca los elementos del vector: ");
  for (i=0; i<v.nelem; i++)
    scanf("%d%*c", &v.vector[i]);

  /* Se invierten los elementos del vector */
  i = 0;
  j = v.nelem-1;
  while (i<j)
  {
    aux = v.vector[i];
    v.vector[i] = v.vector[j];
    v.vector[j] = aux;
    i = i+1;
    j = j-1;
  }

  /* Se muestran los elementos del vector */
  printf("El vector invertido es: ");
  for (i=0; i<v.nelem-1; i++)
    printf("%d, ", v.vector[i]);
  printf("%d\n", v.vector[i]);
}
```

3.

```
#include <stdio.h>
#define MAXFR 300
#define MAXPAL 20
typedef char tfrase[MAXFR];

main()
{
  tfrase f;
  char pal[MAXPAL];
  int i, j, k, lonp, cont_pal;

  /* Se lee la frase acabada en '.' */
  printf("Introduzca una frase acabada en punto: ");
  i = 0;
  scanf("%c", &f[i]);
```

```
while (i<MAXFR-1 && f[i]!='.')
{
  i = i+1;
  scanf("%c", &f[i]);
}
if (i==MAXFR-1)
  f[i] = '.';

printf("\nPalabras palindromas:\n");
cont_pal = 0;
i = 0;
while (f[i]!='.')
{
  /* Se extrae una palabra */
  j = 0;
  while ((f[i]>='a' && f[i]<='z') ||
         (f[i]>='A' && f[i]<='Z'))
  {
    pal[j] = f[i];
    j = j+1;
    i = i+1;
  }

  lonp = j;  /* Longitud de la palabra extraída */

  /* Se avanzan espacios y signos de puntuación y
     se deja el índice i al inicio de la
     palabra siguiente */
   while (f[i]!='.' &&
         (f[i]<'a' || f[i]>'z') &&
         (f[i]<'A' || f[i]>'Z'))
     i = i+1;

   /* Se determina si pal es palíndroma */
   j = 0;
   k = lonp-1;
   while (j<k && (pal[j]==pal[k] ||
         (pal[j]>='a' && pal[j]<='z'
          && pal[j]==pal[k]+('a'-'A')) || (pal[j]>='A'
          && pal[j]<='Z' && pal[j]==pal[k]-('a'-'A'))))
   {
     j = j+1;
     k = k-1;
   }

  if (j>=k)              /* Es palíndroma */
  {
    /* Se muestra la palabra */
    for (j=0; j<lonp; j++)
      printf("%c", pal[j]);
    printf("\n");

    cont_pal = cont_pal+1;
  }
}
```

```
  printf("En total hay %d palabras palindromas\n\n",
          cont_pal);
}
```

4.

```
#include <stdio.h>
#define MAXELEM 100
typedef struct
{
  int vec[MAXELEM];  /* Vector de enteros */
  int nelem;         /* Número de elementos en vector */
} tvector;

main()
{
  tvector v;
  int el, n, i, j, pos;

  v.nelem = 0;
  printf("Introduzca el numero de elementos del
          vector (MAX. 100): ");
  scanf("%d%*c", &n);
  printf("Introduzca los elementos del vector: ");

  /* Se lee e inserta ordenadamente */
  for (i=0; i<n; i++)
  {
    scanf("%d%*c", &el);

    /* Se busca la posición a insertar para mantener
       el orden */
      j = 0;
      while (j<v.nelem && el<v.vec[j])
        j = j+1;
      pos = j;

      /* Se inserta el elemento en el vector ordenado */
      for (j=v.nelem; j>pos; j--)
        v.vec[j] = v.vec[j-1];

      v.vec[pos] = el;
      v.nelem = v.nelem+1;
  }

  printf("\nVector ordenado: ");
  for (i=0; i<v.nelem-1; i++)
    printf("%d, ", v.vec[i]);
  printf("%d\n", v.vec[i]);

  /* Se eliminan los elementos pares */
  i = 0;
  while (i<v.nelem)
  {
    if (v.vec[i]%2==0)
    {
      for (j=i; j<v.nelem-1; j++)
        v.vec[j] = v.vec[j+1];
```

```
      v.nelem = v.nlem-1;
    }
    else
      i = i+1;
  }

  printf("\nVector ordenado sin numeros pares: ");
  for (i=0; i<v.nelem-1; i++)
    printf("%d, ", v.vec[i]);
  printf("%d\n", v.vec[i]);
}
```

5.

```
#include <stdio.h>
#define MAX_GRADO 8
#define NUM_POLINOMIO 10
#define ERROR 0.0001
typedef struct
{
  int grado;                /* Grado del polinomio */
  float coef[MAX_GRADO+1];  /* Coeficientes */
} tpolinomio;

main()
{
  tpolinomio vp[NUM_POLINOMIO], suma;
  int i, j, exp;
  float coef;
  char car;

  /* Se inicializan los polinomios y suma */
  for (i=0; i<NUM_POLINOMIO; i++)
  {
    vp[i].grado = 0;
    for (j=0; j<=MAX_GRADO; j++)
      vp[i].coef[j] = 0.0;
  }

  suma.grado = 0;
  for (j=0; j<=MAX_GRADO; j++)
    suma.coef[j] = 0.0;

  /* Se leen los polinomios */
  printf("\nIntroduzca los polinomios (coefxgrado ...)
         finalizados en '\\n':\n");
  for (i=0; i<NUM_POLINOMIO; i++)
  {
    printf("Polinomio %i: ", i);
    car = ' ';
    while ( car != '\n')
    {
      scanf ("%fx%d%c",&coef, &exp, &car);
   if (exp > vp[i].grado)
     vp[i].grado = exp;
   vp[i].coef[exp] = coef;
```

```
    }
  }

  /* Se calcula la suma */
  for (j=0; j<=MAX_GRADO; j++)
  {
    for (i=0; i<NUM_POLINOMIO; i++)
           suma.coef[j] = suma.coef[j] + vp[i].coef[j];

    if (suma.coef[j] >= 0.0 + ERROR ||
        suma.coef[j] <= 0.0 - ERROR)
        suma.grado = j;
  }

  /* Se muestra la suma en pantalla */
  printf("\nEl polinomio suma es de grado %d: ",
         suma.grado);
  for (j=0; j<=suma.grado; j++)
  {
    if (suma.coef[j] >= 0.0 + ERROR ||
      suma.coef[j] <= 0.0 - ERROR)
    printf (" %.2fx%d ",suma.coef[j], j);
  }
  printf("\n\n");
}
```

6.

```
#include <stdio.h>
#define MAX 100
typedef struct
{
  int num;          /* Número de la matrícula */
  char letras[3];   /* Letras de la matrícula */
} tmatricula;

typedef struct
{
  int nmat;        /* Número de matrículas en el vector */
  tmatricula vmat[MAX];  /* Vector de matrículas */
} tvector_matricula;

main()
{
  tvector_matricula v={10, {{2343,"BZG"}, {2343,"FKD"},
                 {1203,"KAS"}, {2666,"AHJ"}, {2300,"BGZ"},
                 {2343,"AHJ"}, {3433,"BZG"}, {5677,"CZF"},
                 {8766,"BZZ"}, {0001,"BZG"} }};
  int posmin, i, j, k;
  tmatricula aux;

  printf("Vector de matriculas original:\n");
  for (i=0; i<v.nmat; i++)
  {
    printf ("%04d ", v.vmat[i].num);
    for (j=0; j<3; j++)
```

```
      printf ("%c", v.vmat[i].letras[j]);
    printf ("\n");
  }
  /* Se ordenan los elementos del vector utilizando
     el método de la selección directa */
  for (i=0; i<v.nmat-1; i++)
  {
    posmin = i;
    for (j=i+1; j<v.nmat; j++)
    {
      k = 0;
      while (k<3 &&
             v.vmat[j].letras[k]==v.vmat[posmin].letras[k])
        k = k+1;
        /* k almacena la posición del carácter en que
           difieren las dos matrículas. Si vale 3, es que
           tienen las mismas letras */


      if ((k==3 && v.vmat[j].num<v.vmat[posmin].num)||(k<3
          && v.vmat[j].letras[k]<v.vmat[posmin].letras[k]))
         posmin = j;
    }
    /* Se intercambian los elementos de las
       posiciones i y posmin */
     aux = v.vmat[i];
     v.vmat[i] = v.vmat[posmin];
     v.vmat[posmin] = aux;
  }
  printf("\nVector de matriculas ordenado:\n");
  for (i=0; i<v.nmat; i++)
  {
    printf ("%04d ", v.vmat[i].num);
    for (j=0; j<3; j++)
      printf ("%c", v.vmat[i].letras[j]);
    printf ("\n");
  }
}
```

Funciones: paso de parámetros por valor

Hasta ahora, se han visto algunas funciones predefinidas por el lenguaje tales como las funciones `printf` y `scanf`. Además de estas dos funciones, existen muchas otras que pueden ser utilizadas por el programador. Este tipo de funciones predefinidas se denominan *funciones de librería*. Sin embargo, el programador puede necesitar definir sus propias funciones o utilizar funciones implementadas por otros programadores. Estas funciones las denominaremos *funciones de usuario* y en este capítulo explicaremos cómo definirlas e implementarlas.

En la mayoría de los casos, las funciones requieren información para poder ejecutar la tarea para la cual fueron creadas. Esta información se pasa a la función utilizando unos identificadores conocidos como *parámetros*. Como se verá, se puede pasar información a las funciones utilizando parámetros por valor o por referencia. En este capítulo, se explica el paso por valor y, en el siguiente, el paso por referencia. Primero, se muestra la utilidad de las funciones y posteriormente se explica cómo llamar y definir una función. A continuación, se explican en detalle los pasos en la evaluación de una función y, finalmente, se proporciona un ejemplo de uso de las funciones.

8.1 Función

Una función se define como un conjunto de sentencias al cual se asigna un nombre y que realiza una tarea específica sobre unos datos determinados. Algunos ejemplos de tareas que una función concreta puede realizar son los siguientes: leer desde el teclado un carácter, calcular el factorial de un número, calcular la distancia entre dos puntos del plano, dividir números complejos, etc.

Las funciones también sirven para facilitar la implementación de los programas ya que, tal como se ha visto en el capítulo 1, sección 1.3, permiten descompo-

ner el programa original en tareas más sencillas. En definitiva, la utilización de las funciones de usuario permitirá construir programas bien estructurados y, en consecuencia, hacer el código más legible, más fácil de depurar y menos costoso de mantener. Adicionalmente, otras ventajas del uso de funciones es que permite reducir la extensión del código y facilita su reutilización.

A continuación, se muestra la utilidad del uso de funciones. Supóngase, por ejemplo, que se quiere implementar un programa que muestre por pantalla el valor numérico del número combinatorio C_m^n representado por $\binom{m}{n} = \frac{m!}{n!(m-n)!}$.

La implementación del programa sin utilizar funciones sería:

```
#include <stdio.h>

main()
{
  int m, n, i;
  float c, fact_m, fact_n, fact_mn; /* Se declaran de tipo
                                       float para tener un mayor
                                       rango de representación */

  printf("Introduzca los valores de m y n (con m>=n): ");
  scanf("%d %d%*c", &m, &n);

  /* Cálculo del factorial de m */
  fact_m = 1.0;
  for(i=m; i>1; i--)
    fact_m = fact_m*i;

  /* Cálculo del factorial de n */
  fact_n = 1.0;
  for(i=n; i>1; i--)
    fact_n = fact_n*i;

  /* Cálculo del factorial de m-n */
  fact_mn = 1.0;
  for(i=m-n; i>1; i--)
    fact_mn = fact_mn*i;

  c = fact_m/(fact_n*fact_mn);

  printf("El número de combinaciones de %d elementos tomados de
          %d en %d es: %.0f\n", m, n, n, c);
}
```

En el programa anterior, obsérvese que el código que realiza el cálculo del factorial se repite tres veces: para `m`, para `n` y para `m-n`; lo único que cambia es el número del cual se quiere calcular el factorial. De esta forma, si se dispusiera de una función `factorial` que calculara y devolviera el factorial de un número dado, el programa anterior podría escribirse como:

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

```
  int m, n;
  float c, fact_m, fact_n,fact_mn;
  printf("Introduzca los valores de m y n (con m>=n): ");
  scanf("%d %d%*c", &m, &n);
  fact_m = factorial(m);     /* Calcula el factorial de m y lo
                                almacena en la variable fact_m */
  fact_n = factorial(n);     /* Calcula el factorial de n y lo
                                almacena en la variable fact_n */
  fact_mn = factorial(m-n); /* Calcula el factorial de m-n y lo
                                almacena en la variable fact_mn */
  c = fact_m/(fact_n*fact_mn);
  printf("El numero de combinaciones de %d elementos tomados de
          %d en %d es: %.0f\n", m, n, n, c);
}
```

Obsérvese que, en esta versión, el número de sentencias del `main` se ha reducido con respecto a la versión inicial del programa. Además, la claridad y la legibilidad del programa han mejorado y no se triplican las sentencias asociadas al cálculo del factorial. En este caso, se ha utilizado una función denominada `factorial` para calcular el factorial de cada uno de los tres números requeridos (`m`, `n` y `m-n`) y se han realizado las llamadas correspondientes a la función. Si la función `factorial` estuviese definida en alguna librería, entonces en el programa anterior solo se tendría que incluir dicha librería para que el programa se ejecutara correctamente. Sin embargo, la función `factorial` no está definida en las librerías que provee el lenguaje C; es una función de usuario y, por tanto, nos corresponde a nosotros implementarla. En este capítulo, se explica cómo utilizar y manipular las funciones de usuario.

8.2 Llamada a una función

Primero, veamos cómo se realiza una llamada a una función que se supone previamente definida por el usuario, o bien que es una función de librería.

Tal como ya se ha visto en el capítulo 2, sección 2.5, para llamar a una función para su ejecución es necesario escribir su nombre y, a continuación, y entre paréntesis, colocar adecuadamente los datos con los cuales operará la función. Además, si la función devuelve un valor, este debe asignarse a una variable previamente declarada.

La sintaxis general para llamar a una función es la siguiente:

```
nombre_función(preal1, preal2,…);
                    /* Si la función no devuelve un valor */

nom_var = nombre_función(preal1, preal2,…);
                    /* Si la función devuelve un valor */
```

En la llamada a una función, se especifican su nombre (`nombre_función`), los parámetros reales (`preal1, preal2,…`), y la variable (`nom_var`) donde se almacenará el resultado de la función (en su caso). Los parámetros reales (`preal1, preal2,…`) son expresiones cuyos resultados son los valores que se pasan a la función para que opere con ellos.

La llamada a una función se puede efectuar desde cualquier parte del programa, ya sea desde el programa principal `main` o desde cualquier otra función. A continuación, se muestran algunos ejemplos de llamadas a funciones.

Ejemplo 1:

```
double raiz;
raiz = sqrt(18.0);
```

La función `sqrt` calcula la raíz cuadrada del parámetro real `18.0` y devuelve un resultado que se almacena en la variable `raiz`. La función `sqrt` es una función de la librería `math.h`.

Ejemplo 2:

```
int m=5;
float fact_m;
fact_m = factorial(2*m+1);
```

La función calcula el factorial del parámetro real `2*m+1` (en este caso, con valor `11`). Obsérvese que el parámetro real puede ser una expresión. El resultado que devuelve la función se almacena en la variable `fact_m` de tipo `float`. La función `factorial` es una función de usuario.

Ejemplo 3:

```
tdni dni={47882309, 'N'};
mostrar_dni(dni);
```

La función `mostrar_dni` muestra en pantalla el número y la letra del parámetro real `dni` de tipo `tdni` (estructura definida en el capítulo 6, sección 6.2). La función `mostrar_dni` no devuelve ningún valor y es una función de usuario.

8.3 Definición de una función

Ahora ya sabemos cómo llamar a una función para su ejecución. Veamos, a continuación, cómo crear o definir nuestras propias funciones, las funciones de usuario.

La definición de una función está formada por la cabecera y el cuerpo de la función. La cabecera de la función contiene el tipo de dato del resultado que devuelve la función, su nombre y la lista de parámetros que necesita para reali-

zar la tarea para la cual fue diseñada. El cuerpo de la función contiene las sentencias para realizar dicha tarea.

La sintaxis general de la definición de una función es la siguiente:

```
tipo_result nombre_función(tipo1 pformal1,…,tipon pformaln)
{
  declaración de variables locales;

  sentencias de la función;
  return(expresión);
}
```

donde

1. `tipo_result` es el tipo de dato del resultado que devuelve la función. Si la función no retorna ningún valor, se utiliza la palabra reservada `void`. Si, por el contrario, la función retorna un valor, este puede ser de tipo elemental o estructura, pero no puede ser de tipo vector.

2. `nombre_función` es el identificador de la función. Es muy recomendable elegir un nombre escrito en letras minúsculas que describa lo que hace la función. Si, para indicarlo, es preciso utilizar más de una palabra, entonces será necesario separar cada palabra utilizando guiones (- o _). No se pueden utilizar espacios en el nombre de la función.

3. `tipo1 pformal1,…,tipon pformaln` es la lista de parámetros de la función. Estos parámetros son los parámetros formales y almacenan los valores de los parámetros reales indicados en la llamada a la función. Para cada parámetro formal, es necesario indicar su tipo de dato y su nombre. El número y el tipo de los parámetros formales han de coincidir con el número y el tipo de los parámetros reales de la llamada. Los parámetros formales son variables que se crean en el momento de la llamada y se inicializan con los valores de los parámetros reales correspondientes. Cuando finaliza la ejecución de la función, los parámetros formales desaparecen, es decir, ya no es posible acceder a los valores que almacenaban.

4. `sentencias de la función` son el cuerpo de la función y contienen el conjunto de sentencias necesarias para llevar a cabo la tarea para la cual la función fue creada. Estas sentencias pueden ser de cualquier tipo (asignación, sentencias condicionales, sentencias iterativas, llamada a función). El cuerpo de la función puede contener `declaraciones de variables` que se denominan *locales*, ya que se mantienen vivas (activas) en la memoria mientras se ejecuta la función pero que desaparecen (inactivas), una vez finalizada la ejecución de la misma. De esta forma, y debido a que las variables locales tienen un ámbito de actuación limitado a la función que contiene las declaraciones, es posible que existan variables locales con el mismo nombre en otras funciones, pero estas no tienen ninguna relación entre sí.

Las sentencias de la función únicamente manipulan los parámetros formales y las variables locales definidas en el cuerpo de la función, es decir, no pueden acceder a variables declaradas en el `main` u otras funciones.

5. `return` es una palabra reservada del lenguaje C que se utiliza para devolver el resultado que calcula la función y que corresponde al valor de la `expresión`. Si la función no devuelve ningún valor, entonces la sentencia `return(expresión)` no se incluye en el cuerpo de la función.

Ejemplo:

```
float factorial(int num)
{
  int i;
  float f=1.0;

  for(i=num; i>1; i--)
    f = f*i;

  return (f);
}
```

La función `factorial` recibe como parámetro un valor de tipo entero que almacena en el parámetro formal `num` y retorna un valor de tipo `float` correspondiente al factorial de `num`. El cuerpo de la función está formado por la sentencia `for`, que calcula el factorial de `num` y lo almacena en la variable local `f`. La sentencia `return` retorna el valor de `f`.

Las variables locales `i` y `f` de la función `factorial`, al igual que el parámetro formal `num`, se mantienen vivos mientras se ejecuta la función. Una vez finalizada la ejecución de la función, estas variables y el parámetro desaparecen, y ya no se pueden manipular.

8.4 Prototipo de una función

En la estructura de un programa, las definiciones de funciones de usuario pueden ir antes o después del `main`. En este libro se escribirán siempre después del `main` y se utilizarán prototipos de funciones para informar al compilador de la existencia de éstas.

En concreto, los prototipos de las funciones sirven para indicar al compilador la cantidad de parámetros formales, el tipo de dato de cada parámetro formal y el tipo del resultado de la función. De esta forma, el compilador puede comprobar si la función es llamada correctamente dentro del programa, sin conocer la definición completa de esta.

El prototipo de una función contiene únicamente la cabecera de la función y el carácter punto y coma. En la estructura del programa, los prototipos de las funciones se colocan justo antes del `main` y después de la definición de los tipos de datos (v. capítulo 2, sección 2.6).

En el lenguaje C, el uso de los prototipos no es obligatorio pero sí recomendable para facilitar la compilación. Si no se utilizan prototipos, hay que vigilar el orden en que se definen las funciones. En este libro, y como norma de estilo (v. anexo 9.B), se utilizan prototipos al implementar los programas y se definen todas las funciones de usuario después del programa principal (`main`).

La sintaxis general del prototipo de una función es la siguiente:

```
tipo_result nombre_función(tipo1 pformal1,…, tipon pformaln);
```

donde el `tipo_result`, el `nombre_función` y la lista de parámetros (`tipo1 pformal1, ..., tipon pformaln`) son los mismos que los indicados en la definición de la función.

Ejemplos:

A continuación, se muestran algunos ejemplos de prototipos de funciones y, para cada uno de ellos, se describe brevemente lo que realiza la función.

```
void mostrar_tablas();
```

Esta función muestra en pantalla las tablas de multiplicar desde la tabla del `2` hasta la tabla del `10`. La función `mostrar_tablas` no tiene parámetros formales y no devuelve ningún resultado.

```
void mostrar_dni(tdni dni);
```

Esta función muestra en pantalla el número y la letra del parámetro formal `dni`, que es de tipo `tdni`. La función `mostrar_dni` no devuelve ningún resultado.

```
tfecha leer_fecha();
```

Esta función devuelve una estructura de tipo `tfecha`, que contiene la fecha que el usuario introduce desde el teclado. La función `leer_fecha` no tiene parámetros formales.

```
float factorial(int num);
```

Esta función recibe como parámetro un valor de tipo entero, que almacena en el parámetro formal `num`, y retorna el valor real correspondiente al factorial de `num`.

```
char mayuscula(char c);
```

Esta función recibe como parámetro un carácter, que almacena en el parámetro formal `c`, y retorna el carácter correspondiente al carácter almacenado en `c` en letra mayúscula.

```
double vol_cili (float r, float h);
```

Esta función recibe dos valores reales, que almacena en los parámetros formales `r` y `h`, que representan, respectivamente, el radio (`r`) y la altura (`h`) de un cilindro. La función `vol_cili` retorna el valor real correspondiente al volumen del cilindro para los parámetros recibidos.

La tabla 8.1 muestra el prototipo y la llamada de cada una de las funciones anteriores, e indica en cada caso el tipo de función que representa: función con parámetros, función sin parámetros, función que no retorna valor o función que retorna valor.

Tabla 8.1
Resumen de prototipos y llamadas a funciones

Tipo de función	Prototipo de la función	Llamada a la función

No retorna valor y no tiene parámetros	`void mostrar_tablas();`	`main()`
		`{`
		`  mostrar_tablas();`
		`}`

No retorna valor y tiene parámetros	`void mostrar_dni(tdni dni);`	`main()`
		`{`
		`  tdni d={47882309, 'N'};`
		`  mostrar_dni(d);`
		`}`

Retorna valor y no tiene parámetros	`tfecha leer_fecha();`	`main()`
		`{`
		`  tfecha fecha;`
		`  fecha = leer_fecha();`
		`}`

Retorna valor y tiene parámetros	`float factorial(int num);`	`main()`
		`{`
		`  int n=5;`
		`  float f;`
		`  f = factorial(n);`
		`}`

Retorna valor y tiene parámetros	`char mayuscula(char c);`	`main()`
		`{`
		`  char m,min='e';`
		`  m = mayuscula(min);`
		`}`

Retorna valor y tiene parámetros	`double vol_cili(float r,`
	`  float h);`
		`main()`
		`{`
		`  float r=3.4, h=5.3;`
		`  double v;`
		`  v = vol_cili(r,h);`
		`}`

Obsérvese que en la tabla 8.1, en la llamada a la función (v. tercera columna):

- En las funciones que retornan un resultado, este se asigna a una variable del mismo tipo que el que retorna la función. Si no fueran del mismo tipo, habría una conversión de tipo implícita (v. capítulo 3, sección 3.4).

- Las variables que aparecen en las expresiones de los parámetros reales han de estar inicializadas antes de realizar la llamada a la función.

8.5 Evaluación de una función paso a paso

Retomemos ahora el programa que realiza el cálculo del número combinatorio C_m^n planteado en la sección 8.1. Ahora que ya sabemos cómo crear funciones de usuario, podemos crear la función `factorial` y reescribir el código completo de la forma siguiente:

```
#include <stdio.h>
/* Prototipo de la función */
float factorial(int num);

main()
{
  int m, n;
  float c, fact_m, fact_n,fact_mn;

  printf("Introduzca los valores de m y n (con m>=n): ");
  scanf("%d %d%*c", &m, &n);

  fact_m = factorial(m);      /* Calcula el factorial de m y lo
                                 almacena en la variable fact_m */

  fact_n = factorial(n);      /* Calcula el factorial de n y lo
                                 almacena en la variable fact_n */

  fact_mn = factorial(m-n);   /* Calcula el factorial de m-n y lo
                                 almacena en la variable fact_mn */

  c = fact_m/(fact_n*fact_mn);

  printf("El numero de combinaciones de %d elementos tomados de
          %d en %d es: %.0f\n", m, n, n, c);
}

/* Definición de las funciones de usuario */
float factorial(int num)
{
  int i;
  float f=1.0;

  for(i=num; i>1; i--)
    f = f*i;
  return (f);
}
```

Obsérvese que el cuerpo de la función `factorial` contiene las sentencias para realizar el cálculo del factorial (sentencia `for`), sabiendo que el valor del cual queremos calcular el factorial está almacenado en el parámetro formal `num`. Cuando se hace una llamada a la función `factorial` desde el programa principal (`main`), se está llamando a la función para su ejecución y, a la vez, se asigna al parámetro formal `num`, el valor del parámetro real correspondiente (`m` en la primera llamada, `n` en la segunda y `m-n` en la última).

Para comprender mejor el mecanismo de paso de parámetros y llamadas a funciones, veamos qué ocurre internamente en el computador cuando se llama a una función para su ejecución:

1. Se reserva espacio en la memoria para almacenar cada uno de los parámetros formales de la función.
2. Se asigna a cada parámetro formal el valor que tiene su correspondiente parámetro real en el momento de la llamada.
3. Se reserva espacio en la memoria para almacenar cada una de las variables locales.
4. Se ejecutan todas las sentencias del cuerpo de la función, hasta el final de la función o hasta la sentencia `return`, en su caso. Después de la sentencia `return`, no se ejecuta ninguna sentencia más de la función.
5. Se vuelve al punto del programa donde se llamó a la función. Si la función devuelve un valor, este es asignado a la variable indicada en la llamada.
6. Se libera el espacio reservado en la memoria para los parámetros formales y para las variables locales.
7. Finalmente, el programa prosigue su ejecución, ejecutando la sentencia siguiente, después de la llamada a la función.

La tabla 8.2 muestra un ejemplo de ejecución del programa anterior que incluye la evaluación paso a paso de cada una de las llamadas a la función `factorial`. La evaluación paso a paso de cada una de las funciones se detalla en las tablas 8.3, 8.4 y 8.5. Para facilitar la explicación de la ejecución del programa, se numeran las líneas del código como sigue:

```
 1: #include <stdio.h>
 2: float factorial(int num);
 3: main()
 4: {
 5:   int m, n;
 6:   float fact_m, fact_n,fact_mn, c;
 7:
 8:   printf("Introduzca los valores de m y n (con m>=n): ");
 9:   scanf("%d %d%*c", &m, &n);
10:   fact_m = factorial(m);
11:   fact_n = factorial(n);
12:   fact_mn = factorial(m-n);
13:   c = fact_m/(fact_n*fact_mn);
14:   printf("El número de combinaciones de %d elementos tomados
              de %d en %d es: %.0f\n", m, n, n, c);
15: }
16:
17: float factorial(int num)
18: {
19:   int i;
```

```
20:    float f=1.0;
21:
22:    for(i=num; i>1; i--)
23:      f = f*i;
24:    return (f);
25: }
```

Línea	Valor de las variables						Entrada por teclado	Salida en pantalla
	m	n	fact_m	fact_n	fact_mn	c		
5	?	?	?	?	?	?		
6	?	?	?	?	?	?		
8	?	?	?	?	?	?		Introduzca los valores de m y n (con m>=n):
9	5	3	?	?	?	?	5 3	
10 tabla 8.3	5	3	120.0	?	?	?		
11 tabla 8.4	5	3	120.0	6.0	?	?		
12 tabla 8.5	5	3	120.0	6.0	2.0	?		
13	5	3	120.0	6.0	2.0	10.0		
14	5	3	120.0	6.0	2.0	10.0		El numero de combinaciones de 5 elementos tomados 3 en 3 es: 10

Tabla 8.2 Ejecución del programa que calcula C_m^n.

Obsérvese que la tabla 8.2 referencia las tablas 8.3, 8.4 y 8.5, que indican, respectivamente, la ejecución paso a paso de la evaluación de la función `factorial` correspondiente a las líneas `10`, `11` y `12` del programa principal. Cada una de estas tablas muestra el valor que retorna la función al programa principal, además de los valores del parámetro formal y de las variables locales.

Línea	Valor del parámetro formal	Valor de las variables locales		Valor que devuelve la función
	num	i	f	
19	5	?	?	
20	5	?	1.0	
22 (c)	5	5	1.0	
23	5	5	5.0	
22 (c)	5	4	5.0	
23	5	4	20.0	
22 (c)	5	3	20.0	
23	5	3	60.0	
22 (c)	5	2	60.0	
23	5	2	120.0	
22 (f)	5	1	120.0	
24	5	1	120.0	120.0

Tabla 8.3 Ejecución paso a paso de la llamada a `factorial` de la línea 10 del programa

Tabla 8.4 Ejecución paso a paso de la llamada a `factorial` de la línea 11 del programa

Línea	Valor del parámetro formal	Valor de las variables locales		Valor que devuelve la función
	num	i	f	
19	3	?	?	
20	3	?	1.0	
22 (c)	3	3	1.0	
23	3	3	3.0	
22 (c)	3	2	3.0	
23	3	2	6.0	
22 (f)	3	1	6.0	
24	3	1	6.0	6.0

Tabla 8.5 Ejecución paso a paso de la llamada a `factorial` de la línea 12 del programa

Línea	Valor del parámetro formal	Valor de las variables locales		Valor que devuelve la función
	num	i	f	
19	2	?	?	
20	2	?	1.0	
22 (c)	2	2	1.0	
23	2	2	2.0	
22 (f)	2	1	2.0	
24	2	1	2.0	2.0

8.6 Ejemplo de uso de funciones

En esta sección, se muestra un ejemplo de uso de funciones utilizando el mismo ejemplo de uso que se ha presentado en el capítulo 6.

Recuérdese que, en este ejemplo, el programa lee desde el teclado la información de `5` personas y muestra por pantalla el DNI de la persona más joven de sexo femenino y el de la persona más adulta de sexo masculino.

A continuación, se muestran en negrita los cambios introducidos en el programa al utilizar funciones para diferenciarlo del código presentado en el ejemplo de uso del capítulo 6, sección 6.4.

```
#include <stdio.h>
#define NUMPERSONAS 5
typedef struct
{
  int dia,mes,anyo;
} tfecha;

typedef struct
{
  int num;
  char letra;
} tdni;

typedef struct
{
  tdni dni;    /* DNI (número y letra) */
```

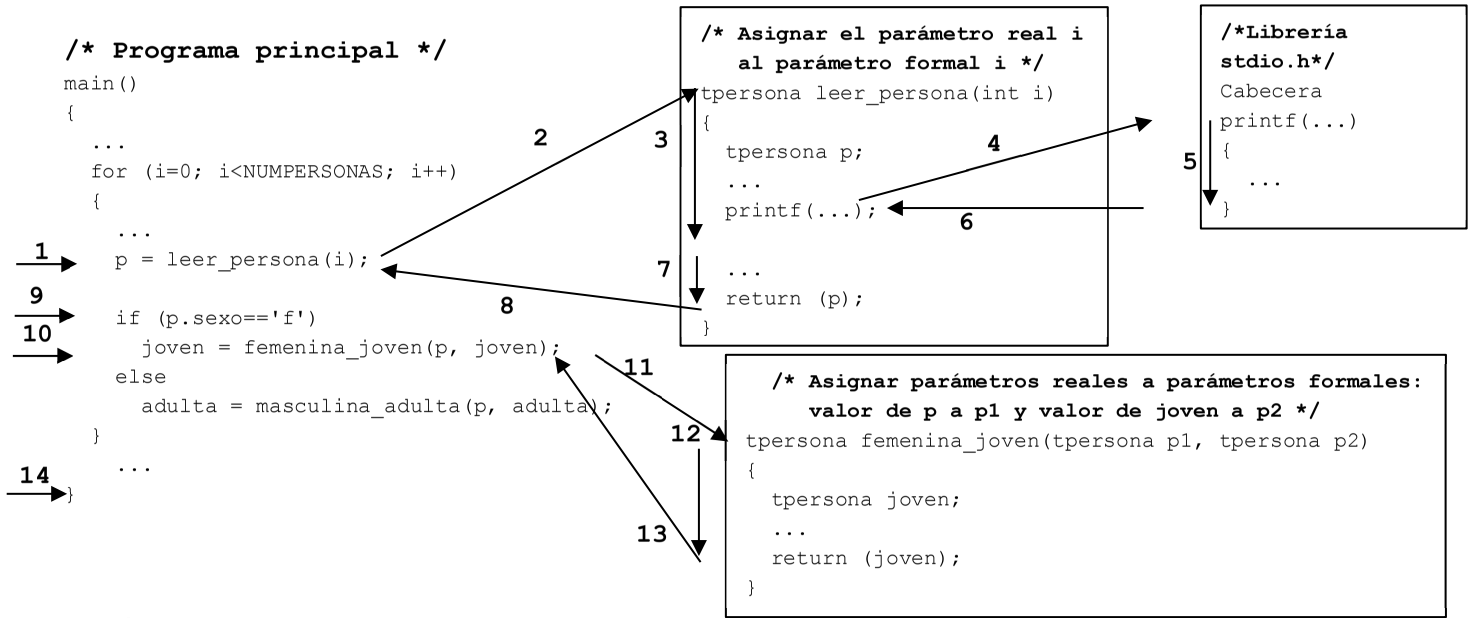
```
  tfecha fn;    /* Fecha de nacimiento */
  char sexo;    /* 'm'-masculino, 'f'-femenino */
} tpersona;

/* Prototipos de las funciones */

void mostrar_dni(tdni d);     /* Función que muestra por pantalla
                                 el número y la letra del DNI d */

tpersona leer_persona(int i); /* Función que lee desde el teclado
                                 los datos de una persona. La
                                 función retorna la información
                                 leída */

tpersona femenina_joven(tpersona p1, tpersona p2);
/* Función que retorna la persona más joven entre p1 y p2. Si el
campo num de p2 es -1 (indica que no almacena información de una
persona), la función retorna p1 */

tpersona masculina_adulta(tpersona p1, tpersona p2);
/* Función que retorna la persona más adulta entre p1 y p2. Si el
campo num de p2 es -1 (indica que no almacena información de una
persona), la función retorna p1 */

main()
{
  tpersona p, joven, adulta;
  int i;

  joven.dni.num = -1;   /* Indica que no se ha encontrado ninguna
                           persona femenina */
  adulta.dni.num = -1;  /* Indica que no se ha encontrado ninguna
                           persona masculina */

  for (i=0; i<NUMPERSONAS; i++)
  {
    /* Se leen los datos de una persona */
    p = leer_persona(i);

    if (p.sexo=='f')
      joven = femenina_joven(p, joven);
    else
      adulta = masculina_adulta(p, adulta);
  }

   /* Mostrar el DNI de la mujer más joven y del hombre
      más adulto */
   if (joven.dni.num==-1)
     printf ("No se han introducido personas del sexo
              femenino\n");
   else
   {
     printf ("DNI de la persona femenina más joven: ");
     mostrar_dni(joven.dni);
   }
```

```
   if (adulta.dni.num==-1)
     printf ("No se han introducido personas del sexo
              masculino\n");
   else
   {
     printf ("DNI de la persona masculina más adulta: ");
     mostrar_dni(adulta.dni);
   }
}

void mostrar_dni(tdni d)
{
  printf ("%d%c\n", d.num, d.letra);
}

tpersona leer_persona(int i)
{
  tpersona p;

  printf("DNI de la persona %d: ", i);
  scanf("%d%c%*c", &p.dni.num, &p.dni.letra);
  printf("Fecha de nacimiento de la persona %d (d/m/a): ", i);
  scanf("%d%*c%d%*c%d%*c", &p.fn.dia, &p.fn.mes, &p.fn.anyo);
  printf("Sexo de la persona %d (m o f): ", i);
  scanf("%c%*c", &p.sexo);

  return (p);
}

tpersona femenina_joven(tpersona p1, tpersona p2)
{
  tpersona joven = p2;

  if (p2.dni.num==-1)
    joven =  p1;
  else if (p1.fn.anyo>p2.fn.anyo || (p1.fn.anyo==p2.fn.anyo &&
           p1.fn.mes>p2.fn.mes) || (p1.fn.anyo==p2.fn.anyo &&
           p1.fn.mes==p2.fn.mes && p1.fn.dia>p2.fn.dia))
    joven = p1;

  return (joven);
}

tpersona masculina_adulta(tpersona p1, tpersona p2)
{
  tpersona adulta = p2;

  if (p2.dni.num==-1)
    adulta = p1;
  else if (p1.fn.anyo<p2.fn.anyo || (p1.fn.anyo==p2.fn.anyo &&
           p1.fn.mes<p2.fn.mes)|| (p1.fn.anyo==p2.fn.anyo &&
           p1.fn.mes==p2.fn.mes && p1.fn.dia<p2.fn.dia))
    adulta = p1;

  return (adulta);
}
```

Obsérvese que, en este caso, y a diferencia del programa presentado en el capítulo 6, se han incluido la declaración y la definición de cuatro funciones, tres de las cuales retornan un tipo de dato estructurado y la otra no retorna ningún valor (función `mostrar_dni`). Además, se ha modificado el programa principal sustituyendo las sentencias correspondientes por la llamada a cada función, según corresponda.

Las funciones implementadas incluyen las mismas sentencias incluidas anteriormente en el `main`, pero ahora estas operan con los parámetros formales y las variables locales de la función.

La figura 8.1 muestra la secuencia de pasos de ejecución para la llamada de las funciones del programa anterior, suponiendo que se están leyendo los datos de la primera persona y que esta es de sexo femenino.



Obsérvese que:

- Los pasos `1` y `10` realizan la llamada a las funciones `leer_persona` y `femenina_joven`.
- Los pasos `2` y `11` realizan la reserva del espacio en la memoria para almacenar los parámetros formales e inicializar sus valores con los valores de los parámetros reales correspondientes en el momento de la llamada. También realizan la reserva del espacio en la memoria para las variables locales, si las hay.
- Los pasos `3`, `5`, `7` y `12` ejecutan el cuerpo de la función.
- Los pasos `6`, `8` y `13` liberan el espacio reservado para los parámetros formales y las variables locales; además, devuelven el control al punto del programa que provocó su llamada, reemplazando el valor de la llamada por el valor devuelto por la función cuando corresponda (pasos `8` y `13`).

8.7 Ejercicios

1. Complete el código siguiente indicando las sentencias, los parámetros o las variables locales necesarias (dependiendo de cada caso) para ejecutar correctamente el programa. El programa realiza la conversión a pulgadas de un valor leído (desde el teclado) expresado en centímetros:

```
#include <stdio.h>
#define CMT_POR_INCH 25.4

/* Complete el prototipo de la función leer_valor */
       ______ leer_valor ();

/* Complete el prototipo de la función cmts_a_pulgadas */
      float cmts_a_pulgadas(________);
main()
{
         float cm, pul;

  /* Realice la llamada a la función leer_valor para
     leer el valor de los centímetros a convertir */
    ___________________________________

   /* Complete la llamada a la función cmts_a_pulgadas */
______ cmts_a_pulgadas(cm);

printf("%.2f cmts son %.2f puldadas\n", cm, pul);
}
float leer_valor()
{
  float valor;

  printf("Introduzca cmts: ");

  /* Complete la llamada a la función scanf */

  scanf("%f%*c", _________);

  return (valor);
}
 /* Complete la cabecera de la función cmts_a_pulgadas */
float cmts_a_pulgadas(________)
{
  float pulgadas;

  /* Complete el cuerpo de la función cmts_a_pulgadas
     sabiendo que 1 pulgada = 1 cm * CMT_POR_INCH */
  ___________________________________

  /* Indique el valor que retorna la función*/
return (________);
}
```

2. Dado el prototipo de la función siguiente:

```
float grados_a_fahrenheit(float centi);
/* Esta función realiza y retorna la conversión de la
   temperatura (centi) expresada en grados centígrados a
   grados Fahrenheit */
```

a. Defina la función `grados_a_fahrenheit`.
Nota: Recuerde que $1° F = \frac{9}{5}° C + 32$.

b. Escriba el programa en lenguaje C que lea una secuencia de `n` números reales que representan los valores en grados centígrados de las temperaturas de un sistema. El programa ha de mostrar en pantalla un mensaje que indique el valor medio de las `n` temperaturas leídas, expresado en grados Fahrenheit. El valor `n` es leído desde el teclado.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca la cantidad de temperaturas (n): 3
Introduzca la temperatura 1 (C): 32.2
Introduzca la temperatura 2 (C): 29.5
Introduzca la temperatura 3 (C): 31.4

Temperatura media (F) = 87.86
```

3. Dados los prototipos de las funciones siguientes:

```
float factorial(int num);
/* Esta función calcula y retorna el factorial de num */

float variaciones(int m, int n);
/* Esta función calcula y retorna la cantidad de
   variaciones de m elementos tomados de n en n */
```

y, conocida la definición de la función `factorial`:

```
float factorial(int num)
{
  int i;
  float f=1.0;

  for(i=num; i>1; i--)
    f = f*i;

  return (f);
}
```

a. Defina la función `variaciones` sabiendo que las variaciones de `m` elementos tomados de `n` en `n`, denotada por V_m^n, está determinada por la fórmula siguiente: $V_m^n = \frac{m!}{(m-n)!}$.

b. Escriba el programa en lenguaje C, utilizando la función `variaciones`, para mostrar por pantalla la cantidad de variaciones de dos valores enteros leídos desde el teclado. El programa ha de comprobar que el primer valor leí-

do es mayor o igual que el segundo y que ambos valores son enteros positivos. En caso contrario, el programa ha de leer nuevamente los números enteros requeridos hasta que el primer valor sea mayor o igual que el segundo y ambos sean enteros positivos.

Ejemplo de ejecución 1 (en negrita, los datos que el usuario introduce):

```
Introduzca m y n (separados por un espacio): 2 4
Introduzca de nuevo m y n, con m>=n, m>=0 y n>=0: 4 2
Variaciones de 4 elementos tomados de 2 en 2 = 12
```

Ejemplo de ejecución 2 (en negrita, los datos que el usuario introduce):

```
Introduzca m y n (separados por un espacio): 2 1
Variaciones de 2 elementos, tomados de 1 en 1 = 2
```

4. Dados los prototipos de las funciones siguientes:

```
float factorial(int num);
/* Esta función calcula y retorna el factorial de num */

float potencia(int e, float b);
/* Esta función calcula y retorna la potencia de la base b
   elevada al exponente e */

float exponencial_x(int n, float x);
/* Esta función calcula y retorna la aproximación a n
   términos de la función exponencial evaluada en x */
```

y, conocida la definición de la función `factorial`:

```
float factorial(int num)
{
  int i;
  float f=1.0;

  for(i=num; i>1; i--)
    f = f*i;

  return (f);
}
```

a. Defina la función `potencia`, sabiendo que la potencia b^e está determinada por la fórmula siguiente: $\prod_{i=0}^{e-1} b$.

b. Defina la función `exponencial_x`, sabiendo que la aproximación e^x a `n` términos está determinada por la expresión siguiente:

$$\sum_{i=0}^{n-1} \frac{x^i}{i!}$$

Nota: Utilice las funciones `factorial` y `potencia` para realizar la implementación.

c. Escriba el programa en lenguaje C, utilizando la función `exponencial_x`, para mostrar por pantalla el valor de la aproximación a `n` términos de la función, evaluada en un valor real `x`. La cantidad de términos y el valor real donde se evalúa la función son leídos desde el teclado. Suponga que estos valores son válidos.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca n y x (separados por un espacio): 4  4.5
e^(4.5) a 4 terminos es: 30.81
```

5. Dado el tipo de dato `tfraccion`, que almacena la información de una fracción con numerador y denominador mayor estricto que cero,

```
typedef struct
{
  unsigned int num;    /* Numerador */
  unsigned int den;    /* Denominador */
} tfraccion;
```

y considerando ahora los prototipos de funciones siguientes:

```
unsigned int mcd(unsigned int a, unsigned int b);
unsigned int mcm(unsigned int a, unsigned int b);
tfraccion leer_frac(int i);
void mostrar_frac(tfraccion f);
tfraccion sumar_frac(tfraccion f1, tfraccion f2);
tfraccion restar_frac(tfraccion f1, tfraccion f2);
tfraccion multiplicar_frac(tfraccion f1, tfraccion f2);
tfraccion dividir_frac(tfraccion f1, tfraccion f2);
```

a. Defina la función `mcd` para que retorne el máximo común divisor de los parámetros `a` y `b`.

b. Defina la función `mcm` para que retorne el mínimo común múltiplo de los parámetros `a` y `b`.

Nota: Recuerde que `mcm(a,b) = a*b/mcd(a,b)`.

c. Defina la función `leer_frac` para que pida al usuario que introduzca desde el teclado la información de la fracción `i` (siendo `i=1` o `i=2`, para indicar respectivamente si se refiere a la información de la fracción 1 o la fracción 2). La función retorna la fracción leída.

d. Defina la función `mostrar_frac` para que muestre en pantalla la información de la fracción `f` pasada como parámetro.

e. Defina la función `sumar_frac` para que retorne la fracción resultante de sumar las dos fracciones pasadas como parámetro (`f1` y `f2`). No es necesario simplificar la fracción obtenida.

f. Defina la función `restar_frac` para que retorne la fracción resultante de restar las dos fracciones pasadas como parámetro (`f1` y `f2`). Esta función

resta la fracción `f2` a la fracción `f1` (`f1-f2`). No es necesario simplificar la fracción obtenida.

g. Defina la función `multiplicar_frac` para que retorne la fracción resultante de multiplicar las dos fracciones pasadas como parámetro (`f1` y `f2`). No es necesario simplificar la fracción obtenida.

h. Defina la función `dividir_frac` para que retorne la fracción resultante de dividir las dos fracciones pasadas como parámetro (`f1` y `f2`). Esta función divide la fracción `f1` entre la fracción `f2` (`f1/f2`). No es necesario simplificar la fracción obtenida.

i. Escriba un programa en C que lea del teclado dos fracciones con numerador y denominado mayor estricto que cero y muestre en pantalla el resultado de sumarlas, restarlas, multiplicarlas y dividirlas. Para realizar esta implementación, utilice las funciones anteriores.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Fracción 1 (formato x/y): 2/3
Fracción 2 (formato x/y): 5/8

RESULTADO DE LAS OPERACIONES:
Suma: 2/3 + 5/8 = 31/24
Resta: 2/3 - 5/8 = 1/24
Multiplicacion: 2/3 * 5/8 = 10/24
Division: 2/3 / 5/8 = 16/15
```

6. Dado el siguiente tipo de dato `tlibro`, que almacena la información de un libro:

```
typedef struct
{
  int mes, anyo;
} tfecha;

typedef struct
{
  int isbn;     /* Número de ISBN (9 dígitos) */
  tfecha fp;    /* Fecha de publicación */
  int nump;     /* Número de páginas */
} tlibro;
```

y, considerando los prototipos de funciones siguientes:

```
tlibro leer_libro();
void mostrar_libro(tlibro lib);
int comparar_fechas(tfecha f1, tfecha f2);
tlibro mas_antiguo(tlibro lib1, tlibro lib2);
```

a. Defina la función `leer_libro` para que retorne la información de un libro leído desde el teclado.

b. Defina la función `mostrar_libro` para que muestre por pantalla la información del libro `lib` pasado como parámetro. La función ha de mostrar la información del libro utilizando el formato siguiente:

```
ISBN: XXXXXXXXX
Fecha de publicacion: XX/XXXX
Num. paginas: XXX
```

c. Defina la función `comparar_fechas` para que retorne `1` si la fecha `f1` es más antigua que la fecha `f2`; en caso contrario, la función retorna `0`.

d. Defina la función `mas_antiguo` para que retorne la información del libro más antiguo entre `lib1` y `lib2`.

e. Escriba un programa en lenguaje C que lea del teclado la información de `10` libros y muestre por pantalla la información del libro que tiene más de `150` páginas y la fecha de publicación más antigua. En caso de empate, el programa ha de mostrar el último libro introducido. Para realizar esta implementación, utilice las funciones anteriores.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca la informacion de 10 libros:
(Formato isbn:fp_mes/fp_any:num_paginas)
929222222:12/2000:130
872372882:12/2000:151
323782833:11/1966:200
863826346:6/1988:256
553457882:6/1983:111
934545453:12/2000:183
844455531:5/2003:351
354531233:1/2005:325
666445343:11/1966:193
544334445:3/1998:89

El libro mas antiguo con mas de 150 paginas es:
ISBN: 666445343
Fecha de publicacion: 11/1966
Num. paginas: 193
```

8.8 Respuesta a los ejercicios propuestos

1.

```
#include <stdio.h>
#define CMT_POR_INCH 25.4

/* Complete el prototipo de la función leer_valor */

float leer_valor();

/* Complete el prototipo de la función cmts_a_pulgadas */

float cmts_a_pulgadas(float cm);
```

```
main()
{
  /* Complete la declaración de variables locales */
  float cm, pul;
  /* Realice la llamada a la función leer_valor para
     leer el valor de los centímetros a convertir */
  cm = leer_valor();
  /* Complete la llamada a la función cmts_a_pulgadas */
  pul = cmts_a_pulgadas(cm);
  printf("%.2f cmts son %.2f puldadas\n", cm, pul);
}
float leer_valor()
{
  float valor;
  printf("Introduzca cmts: ");
/* Complete la llamada a la función scanf */
  scanf("%f%*c", &valor);
  return (valor);
}

/* Complete la cabecera de la función cmts_a_pulgadas */
float cmts_a_pulgadas(float cm)
{
  float pulgadas;
  /* Complete el cuerpo de la función cmts_a_pulgadas
     sabiendo que 1 pulgada = 1 cm * CMT_POR_INCH */
  pulgadas = cm*CMT_POR_INCH;
  /* Indique el valor que retorna la función*/
  return (pulgadas);
}
```

2.

a.

```
float grados_a_fahrenheit(float centi)
  {
    float fah;
    fah = (centi * 9.0)/5.0 + 32.0;
    return (fah);
  }
```

b.

```
#include <stdio.h>
float grados_a_fahrenheit(float centi);
```

```
main()
{
  int n,i;
  float t,f,r=0.0;

   printf("Introduzca la cantidad de temperaturas (n): ");
   scanf("%d%*c", &n);
  if (n>0)
  {
    for(i=1; i<=n; i++)
    {
       printf("Introduzca temperatura %d (C): ", i);
       scanf("%f%*c", &t);
      f = grados_a_fahrenheit(t);
      r = r+f;
    }
     printf("\nTemperatura media (F) = %.2f\n", r/n);
  }
  else
     printf("No se han introducido temperaturas\n");
}

/* Coloque aquí la definición de la función
   grados_a_fahrenheit del apartado anterior */
```

3.

a.

```
float variaciones (int m, int n)
{
  float facto_m, facto_mn;

  facto_m = factorial(m);
  facto_mn = factorial(m-n);

  return (facto_m/facto_mn );
}
```

b.

```
#include <stdio.h>

float factorial(int num);
float variaciones (int m, int n);

main()
{
  int m, n;
  float vari;

  printf("Introduzca m y n (separados por un
          espacio): ");
  scanf("%d %d%*c", &m, &n);
  while ((m<n)||((m<0)||(n<0)))
  {
    printf("Introduzca de nuevo m y n, con
            m>=n, m>=0 y n>=0: ");
    scanf("%d %d%*c", &m, &n);
  }
```

```
  vari = variaciones(m,n);
  printf("\nVariaciones de %d elementos tomados de %d en
         %d = %.0f\n", m, n, n, vari);
}

/* Coloque aquí la definición de la función factorial y
   de la función variaciones (del apartado anterior) */
```

4.

a.

```
float potencia(int e, float b)
{
  int i;
   float r=1.0;

  for(i=0; i<e; i++)
     r = r*b;

  return (r);
}
```

b.

```
float exponencial_x(int n, float x)
{
  int i;
   float r=0.0;

  for(i=0; i<n; i++)
     r = r + potencia(i,x)/factorial(i);

  return (r);
}
```

c.

```
#include <stdio.h>

float factorial(int num);
float potencia(int e, float b);
float exponencial_x(int n, float x);

main()
{
  int n;
  float x, e;

  printf("Introduzca n y x (separados por
          un espacio): ");
  scanf("%d %f%*c", &n, &x);

  e = exponencial_x(n,x);

  printf("e^(%.1f) a %d terminos es: %.2f\n", x,n,e);
}

  /* Coloque aquí la definición de las funciones de los
apartados anteriores, y de la función factorial */
```

5.

a.

```
unsigned int mcd(unsigned int a, unsigned int b)
{
  unsigned int r;

  r = a%b;
  while (r!=0)
  {
    a = b;
    b = r;
    r = a%b;
  }

  return (b);
}
```

b.

```
unsigned int mcm(unsigned int a, unsigned int b)
{
  unsigned int c;

  c = mcd(a,b);

  return (a*b/c);
}
```

c.

```
tfraccion leer_frac(int i)
{
  tfraccion f;

  printf("Fracción %d (formato x/y): ", i);
  scanf("%u/%u%*c", &f.num, &f.den);

  return (f);
}
```

d.

```
void mostrar_frac(tfraccion f)
{
  printf("%u/%u ", f.num, f.den);
}
```

e.

```
tfraccion sumar_frac(tfraccion f1, tfraccion f2)
{
  tfraccion res;

  res.den = mcm(f1.den, f2.den);
  res.num = (res.den/f1.den)*f1.num +
            (res.den/f2.den)*f2.num;

  return (res);
}
```

f.

```
tfraccion restar_frac(tfraccion f1, tfraccion f2)
{
  tfraccion res;

  res.den = mcm(f1.den, f2.den);
  res.num = (res.den/f1.den)*f1.num -
            (res.den/f2.den)*f2.num;

  return (res);
}
```

g.

```
tfraccion multiplicar_frac(tfraccion f1, tfraccion f2)
{
    tfraccion res;

    res.num = f1.num*f2.num;
    res.den = f1.den*f2.den;

    return (res);
}
```

h.

```
tfraccion dividir_frac(tfraccion f1, tfraccion f2)
{
  tfraccion res;

  res.num = f1.num*f2.den;
  res.den = f1.den*f2.num;

  return (res);
}
```

i.

```
#include <stdio.h>

typedef struct
{
  unsigned int num;    /* Numerador */
  unsigned int den;    /* Denominador */
} tfraccion;

unsigned int mcd(unsigned int a, unsigned int b);
unsigned int mcm(unsigned int a, unsigned int b);
tfraccion leer_frac();
void mostrar_frac(tfraccion f);
tfraccion sumar_frac(tfraccion f1, tfraccion f2);
tfraccion restar_frac(tfraccion f1, tfraccion f2);
tfraccion multiplicar_frac(tfraccion f1, tfraccion f2);
tfraccion dividir_frac(tfraccion f1, tfraccion f2);

main()
{
  tfraccion f1, f2, f;

  f1 = leer_frac(1);
```

```
  f2 = leer_frac(2);

  printf("\nRESULTADO DE LAS OPERACIONES:\n");

  f = sumar_frac(f1,f2);
  printf("\nSuma: ");
  mostrar_frac(f1);
  printf(" + ");
  mostrar_frac(f2);
  printf(" = ");
  mostrar_frac(f);
  printf("\n");

  f = restar_frac(f1,f2);
  printf("\nResta: ");
  mostrar_frac(f1);
  printf(" - ");
  mostrar_frac(f2);
  printf(" = ");
  mostrar_frac(f);
  printf("\n");

  f = multiplicar_frac(f1,f2);
  printf("\nMultiplicacion: ");
  mostrar_frac(f1);
  printf(" * ");
  mostrar_frac(f2);
  printf(" = ");
  mostrar_frac(f);
  printf("\n");

  f = dividir_frac(f1,f2);
  printf("\nDivision: ");
  mostrar_frac(f1);
  printf(" / ");
  mostrar_frac(f2);
  printf(" = ");
  mostrar_frac(f);
  printf("\n");
}

/* Coloque aquí la definición de las funciones de los
   apartados anteriores */
```

6.

a.

```
tlibro leer_libro()
{
  tlibro lib;

   scanf("%d:%d/%d:%d%*c", &lib.isbn, &lib.fp.mes,
          &lib.fp.anyo, &lib.nump);

   return (lib);
}
```

b.

```
void mostrar_libro(tlibro lib)
{
  printf("ISBN: %d\nFecha de publicacion: %d/%d\n
          Num. paginas: %d\n", lib.isbn,
          lib.fp.mes, lib.fp.anyo, lib.nump);
}
```

c.

```
int comparar_fechas(tfecha f1, tfecha f2)
{
  int antiguo=0;

  if (f1.anyo < f2.anyo || (f1.anyo == f2.anyo &&
      f1.mes <= f2.mes))
    antiguo = 1;

  return (antiguo);
}
```

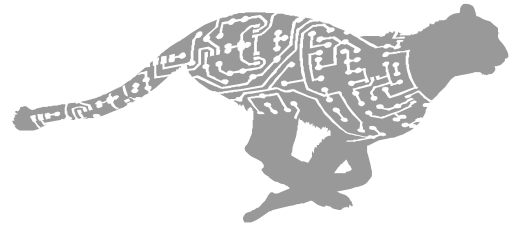
d.

```
 tlibro mas_antiguo(tlibro lib1, tlibro lib2)
 {
   int antiguo;
   tlibro lib=lib2;

   antiguo = comparar_fechas(lib1.fp, lib2.fp);
   if (antiguo==1)
     lib = lib1;

  return (lib);
 }
```

e.

```
#include <stdio.h>
typedef struct
{
  int mes, anyo;
} tfecha;

typedef struct
{
  int isbn;     /* Número de ISBN (9 dígitos) */
  tfecha fp;    /* Fecha de publicación */
  int nump;     /* Número de páginas */
} tlibro;

tlibro leer_libro();
void mostrar_libro(tlibro lib);
int comparar_fechas(tfecha f1, tfecha f2);
tlibro mas_antiguo(tlibro lib1, tlibro lib2);

main()
{
  tlibro el, antic;
  int i;
```

```
  antic.nump = -1; /* Para identificar si se encuentra al
                      menos un libro con más de 150 pág. */

  printf("\nIntroduzca la informacion de 10 libros:\n");
  printf ("(Formato isbn:fp_mes/fp_any:num_paginas)\n");

  for (i=0; i<10; i++)
  {
    el = leer_libro();
    if (el.nump>150)
    {
      if (antic.nump==-1)
        antic = el; /*Primer libro con más de 150 pág.*/
      else
       /* El libro más antiguo hasta ahora */
        antic = mas_antiguo(el, antic);
    }
  }

  if (antic.nump == -1)
    printf("\nNo hay libros con mas de 150 páginas\n ");
  else
  {
    printf("\nEl libro mas antiguo con mas de 150 paginas
              es:\n");
    mostrar_libro(antic);
  }
}

/* Coloque aquí la definición de las funciones de los
   apartados anteriores */
```



Funciones: paso de parámetros por referencia

En el capítulo anterior, hemos visto cómo definir funciones de usuario utilizando el paso de parámetros por valor. Con este tipo de paso de parámetros, se pueden diseñar una gran variedad de funciones, todas ellas con la característica de que retornan un único resultado en función de unos valores concretos. Algunos ejemplos de funciones con estas características son `factorial`, `raiz_cuadrada`, `potencia`, etc.

Sin embargo, existen otras funciones de características diferentes, que no pueden ser implementadas con el paso de parámetros por valor. Son funciones que generan más de un resultado o que necesitan modificar el valor de variables declaradas en el programa principal o en la función que realiza la llamada. Algunos ejemplos de este tipo de funciones son `resolver_ecuacion_segundo_grado`, `intercambiar_dos_valores`, `leer_datos`, etc. En este capítulo, se verá el paso de parámetros por referencia, que permite implementar funciones de estas características.

El capítulo comienza motivando el uso de las funciones con paso de parámetros por referencia. A continuación, se explica cómo realizar el paso de parámetros por referencia para los tipos de datos elementales, las estructuras y los vectores, y al final se proporcionan algunos ejemplos de uso para cada caso.

9.1 Paso de parámetros por referencia

El paso de parámetros por referencia se utiliza cuando en una función se quiere generar más de un resultado o modificar el valor de una variable que está declarada en el programa principal o en la función que realiza la llamada.

Mediante un ejemplo, se muestra cuándo es necesario utilizar el paso de parámetros por referencia. Considérese el programa siguiente, que intercambia el valor de dos variables `v1` y `v2`. Para su implementación, se diseña una función llamada `intercambio`, que realiza el intercambio de los valores pasados como parámetros. Si se diseña la función pasando los parámetros por valor, se obtiene el código siguiente:

```
#include <stdio.h>
void intercambio(int val1, int val2);

main()
{
  int v1, v2;

  printf("Introduzca dos valores enteros (separados por
          un espacio): ");
  scanf("%d %d%*c", &v1, &v2);
  printf("Antes de la funcion intercambiar v1 = %d
          v2 = %d\n", v1, v2);
  intercambio(v1, v2);
  printf("Despues de la funcion intercambiar v1 = %d
          v2 = %d\n", v1, v2);
}

void intercambio(int val1, int val2)
{
  int aux;

  aux = val1;
  val1 = val2;
  val2 = aux;
  printf("Dentro de la funcion intercambiar val1 = %d
          val2 = %d\n", val1, val2);
}
```

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca dos valores enteros (separados por un espacio): 3 5
Antes de la funcion intercambiar v1 = 3 v2 = 5
Dentro de la funcion intercambiar val1 = 5 val2 = 3
Despues de la funcion intercambiar v1 = 3 v2 = 5
```

Obsérvese en el ejemplo de ejecución que los valores de las variables `v1` y `v2` no se han intercambiado en el programa principal; sin embargo, dentro de la función `intercambio`, los valores de los parámetros formales `val1` y `val2` sí se han intercambiado. Por tanto, los cambios en el valor de los parámetros formales (`val1` y `val2`) no quedan reflejados en las variables utilizadas como parámetros reales (`v1` y `v2`).

Como se ha visto en el capítulo anterior, el paso de parámetros por valor consiste en copiar el valor de una variable o expresión (el parámetro real) sobre el parámetro formal de la función que ha sido llamada. A continuación, dentro de la función, se pueden manipular los parámetros formales según convenga

(modificando su valor, si es preciso), pero, una vez finaliza la ejecución de la función, se libera el espacio en la memoria reservado para los parámetros formales, y se pierden así sus valores. Por tanto, este tipo de paso de parámetros, por su propio mecanismo, no puede modificar el valor de la variable utilizada como parámetro real y declarada en el programa principal o en la función que realiza la llamada (`v1` y `v2`, en nuestro ejemplo).

¿Cómo es posible entonces implementar la función `intercambio`? Ya se ha observado que el paso de parámetros por valor no va bien porque la información que se pasa a la función `intercambio` a través del paso de parámetros es el "*valor*" de las variables que se quiere modificar (`v1` y `v2`). En este ejemplo de ejecución, la información que se copia en los parámetros formales `val1` y `val2` es `3` y `5`, respectivamente. Luego, dentro de la función `intercambio`, se manipula `val1` y `val2`, pero, al finalizar la ejecución de la función, los valores de `val1` y `val2` se pierden.

Para implementar la función `intercambio`, es necesario el paso de parámetros por referencia, que consiste simplemente en pasar a la función otro tipo de información: en lugar de pasar el "valor" de las variables `v1` y `v2`, se pasa la "dirección de memoria" de estas variables, es decir, el lugar en la memoria donde se almacenan sus valores. De esta manera, los parámetros formales `val1` y `val2` ya no almacenarán un `3` y un `5`, sino la dirección de memoria de `v1` y la dirección de memoria de `v2`, respectivamente. Ahora, durante la ejecución de la función `intercambio`, es posible acceder a las direcciones de memoria que almacenan `val1` y `val2` y, a través de ellas, modificar los valores de las variables `v1` y `v2`.

Por tanto, si desde una función se quiere modificar el valor de variables declaradas en otras funciones, es necesario realizar el paso de parámetros por referencia. Nótese que puede utilizarse este mecanismo también para diseñar funciones que generan más de un resultado. Uno de los resultados se puede devolver a través de la sentencia `return` y el otro, a través de un parámetro por referencia, pasando la dirección de memoria de la variable que guardará el resultado.

A continuación, se explica cómo realizar el paso de parámetros por referencia en el lenguaje de programación C. Primero, se muestra cómo manipular direcciones de memoria a través de los punteros y, posteriormente, cómo se pasan los parámetros por referencia según su tipo de dato.

9.2 Punteros

En el capítulo 1, sección 1.1, hemos visto que la memoria de un computador es un conjunto de bytes, cada uno de los cuales es identificado por una única dirección. Posteriormente, en el capítulo 2, sección 2.2, se ha explicado que, al declarar las variables en el programa, se reserva espacio en la memoria para almacenar los valores que tomarán dichas variables durante la ejecución del

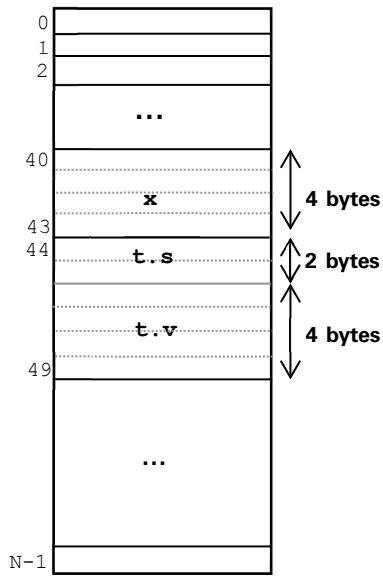
programa. Por tanto, todas las variables de un programa ocupan uno o más bytes en la memoria (según el tipo de dato) y tienen asignada una dirección de memoria. La dirección de memoria de una variable que ocupa más de un byte se identifica siempre con la dirección más baja de todos los bytes que ocupa.

Ejemplo:

```
typedef struct
{
  short s;      /* El campo s ocupa 2 bytes */
  int v;        /* El campo v ocupa 4 bytes */
} ttipo;

main()
{
  int x;        /* La variable x ocupa 4 bytes */
  ttipo t;      /* La variable t ocupa 6 bytes */

}
```

Supóngase que las variables de este programa están almacenadas en la memoria tal como muestra la figura 9.1:

Fig. 9.1
Representación en la memoria de las variables del ejemplo



La dirección de la variable `x` es `40`, la de la variable `t` es `44`, la del campo `s` de la variable `t` también es `44` y la del campo `v` de la variable `t` es `46`. Obsérvese que el número de bytes que ocupa cada variable depende de su tipo. La variable `x`, de tipo `int`, ocupa `4` bytes, mientras que la variable `t`, de tipo estructura, ocupa `6` bytes, `2` bytes para el campo `s` de tipo `short` y 4 bytes para el campo `v` de tipo `int`.

En el lenguaje de programación C, se pueden declarar variables de tipo puntero. Un puntero es una variable que almacena una dirección de memoria. Entre otros usos, los punteros se utilizan para el paso de parámetros por referencia.

Como los punteros guardan direcciones de memoria, se dice que "apuntan" a un dato, en concreto, al dato almacenado en la dirección de memoria que guarda el puntero. Así, por ejemplo, si en el programa anterior tuviéramos un puntero que guardara la dirección de memoria `40`, diríamos que el puntero "apunta" a la variable `x`.

A continuación, se muestra cómo se declara una variable de tipo puntero y los operadores relacionados con punteros.

Declaración de punteros

La sintaxis general para declarar una variable de tipo puntero es la siguiente:

```
tipo *nom_var;        /* Declaración de la variable nom_var de
                         tipo puntero */
```

donde:

- `nom_var` : Es el nombre de la variable de tipo puntero.
- `tipo *` : Indica que la variable `nom_var` es de tipo puntero y que apunta a un dato del tipo especificado en `tipo`.

Ejemplos:

```
char *pcar;     /* pcar es un puntero que apunta a un dato de
                   tipo char */

float *pres;     /* pres es un puntero que apunta a un dato de
                    tipo float */

tdni *pdni;      /* pdni es un puntero que apunta a un dato de
                    tipo tdni (v. capítulo 6, sección 6.2) */
```

Como en todas las declaraciones de variables, al declarar una variable de tipo puntero también se reserva espacio en la memoria para guardar los valores que irá tomando la variable de tipo puntero durante la ejecución del programa. En este caso, estos valores son direcciones de memoria que ocupan 4 bytes.

La figura 9.2 muestra la representación en la memoria de las variables de tipo puntero declaradas en los ejemplos anteriores. Obsérvese que la variable `pcar` apunta a un dato de tipo `char`, `pres` apunta a un dato de tipo `float` y `pdni` apunta a un dato de tipo `tdni`.

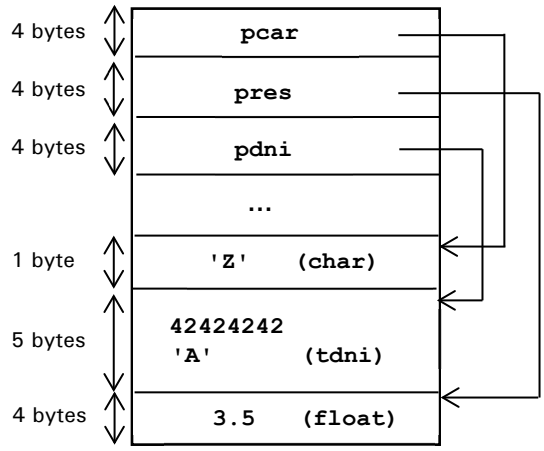


Fig. 9.2
Representación en la memoria de las variables de tipo puntero

Operadores relacionados con los punteros

Hay dos operadores relacionados con los punteros: el operador de referencia (`&`) y el operador de indirección (`*`). A continuación, se analizan estos dos operadores en detalle.

El operador de referencia `&` se puede utilizar delante de cualquier variable y proporciona la dirección de memoria de dicha variable. Así, el resultado de la expresión `&x` es la dirección de memoria de la variable `x` (dirección de memoria `40`, según el ejemplo de la figura 9.1).

Por otro lado, *el operador de indirección `*` solo se puede utilizar delante de una variable de tipo puntero y proporciona el contenido de la dirección de memoria que almacena dicha variable*, es decir, el dato al cual "apunta" el puntero. Así, el resultado de la expresión `*pcar` es el carácter `Z`, según la figura 9.2.

Ejemplo 1:

```
char car='c';
char *pcar;

pcar = &car;    /* La variable pcar es inicializada con
                   la dirección de la variable car */

printf("Son direcciones: pcar = %p y &car = %p\n", pcar, &car);
/* pcar y &car son direcciones de memoria. El especificador de
   formato %p se utiliza para mostrar una dirección de memoria
   en hexadecimal */

printf("Son caracteres: *pcar = %c y car = %c\n", *pcar, car);
/* *pcar y car son caracteres */
```

La figura 9.3 muestra cómo se almacenan en la memoria las variables de este ejemplo y a qué hace referencia cada una de las diferentes expresiones rela-

cionadas con los punteros. El valor almacenado en cada posición de la memoria se indica entre paréntesis. Así, las expresiones `&car` y `&pcar` se refieren a las direcciones de memoria de las variables `car` y `pcar`, respectivamente. La expresión `pcar` se refiere al valor de esta variable que, después de ejecutar el código del ejemplo, vale la dirección de la variable `car`. La expresión `car` se refiere al valor de esta variable, que, según el código anterior, está inicializada en el carácter `'c'`. Finalmente, la expresión `*pcar` hace referencia al contenido de la dirección que guarda `pcar`, es decir, al valor al cual apunta `pcar`. Según el código, `pcar` apunta a `car` y, por tanto, `*pcar` es también el carácter `'c'`.

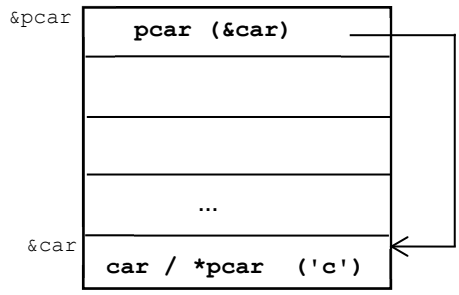


Fig. 9.3
Representación en la memoria de las variables del ejemplo 1

Ejemplo 2:

```
int i=5,j=3;
int *p;

p = &i;    /* La variable p es inicializada con la dirección de
              la variable i. p apunta a i */

j = *p;    /* Se asigna a la variable j el valor al cual apunta
              p, es decir, se asigna el valor de i. Estas dos
              sentencias son equivalentes a j = i; */
```

La figura 9.4 muestra cómo se almacenan en la memoria las variables de este ejemplo y a qué hace referencia cada una de las diferentes expresiones relacionadas con los punteros. De nuevo, el valor almacenado en cada posición de la memoria se indica entre paréntesis.

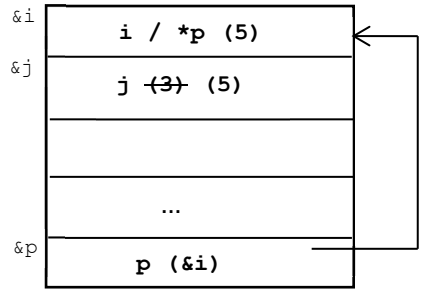


Fig 9.4
Representación en la memoria de las variables del ejemplo 2

Inicialmente, las variables `i` y `j` están inicializadas en `5` y `3`, respectivamente. La variable `p` se inicializa con la dirección de la variable `i`, por lo que `p` "apunta" a la variable `i`. A continuación, a `j` se le asigna `*p`, es decir, el valor al cual apunta `p`, que es precisamente el valor de la variable `i`, que vale `5`. Obsérvese que estas sentencias simplemente copian el valor de la variable `i` en la variable `j`, a través de un puntero.

Los punteros se utilizan con frecuencia en el lenguaje C para diferentes tareas: recorrido de estructuras de datos, gestión dinámica de la memoria, paso de parámetros por referencia, etc. Como ya se ha comentado, en este libro se utilizan los punteros únicamente para realizar el paso de parámetros por referencia. Para realizar otras tareas, se requieren conocimientos más avanzados de programación, que quedan fuera del objetivo de este libro. A continuación, se muestra cómo realizar el paso de parámetro por referencia para diferentes tipos de datos.

9.3 Paso de parámetros por referencia de tipos elementales

Recuérdese que la idea del paso de parámetros por referencia consiste en pasar a la función la "dirección de memoria" de las variables, en lugar de pasar el "valor" de las variables. De esta manera, los parámetros formales de la función son punteros que almacenan la dirección de memoria de otras variables, es decir, los parámetros formales "apuntan" a variables declaradas en otras funciones o en el programa principal. Así, durante la ejecución de la función se puede modificar el valor de dichas variables a través del puntero (parámetro formal).

Véase, como ejemplo, el programa que se ha mostrado en la sección 9.1, realizando ahora el paso de parámetros por referencia (en negrita, se muestran los cambios con respecto al paso de parámetros por valor):

```
#include <stdio.h>
void intercambio(int *val1, int *val2);

main()
{
  int v1, v2;

  printf("Introduzca dos valores enteros (separados por
          un espacio): ");
  scanf("%d %d%*c", &v1, &v2);
  printf("Antes de la funcion intercambiar v1 = %d
          v2 = %d\n", v1, v2);
  intercambio(&v1, &v2);
  printf("Despues de la funcion intercambiar v1 = %d
          v2 = %d\n", v1, v2);
}
```

```
void intercambio(int *val1, int *val2)
{
  int aux;
  aux = *val1;
  *val1 = *val2;
  *val2 = aux;
  printf("Dentro de la funcion intercambiar *val1 = %d
          *val2 = %d\n", *val1, *val2);
}
```

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca dos valores enteros (separados por un espacio): 3 5
Antes de la funcion intercambiar v1 = 3 v2 = 5
Dentro de la funcion intercambiar *val1 = 5 *val2 = 3
Despues de la funcion intercambiar v1 = 5 v2 = 3
```

Obsérvese que:

- A diferencia de lo que ocurría en el paso de parámetros por valor, ahora sí se ha realizado el intercambio de los valores de las variables `v1` y `v2` del programa principal.

- En la llamada a la función `intercambio`, se utiliza el operador de referencia `&` para pasar a la función la dirección de las variables `v1` y `v2`.

  ```
  intercambio(&v1, &v2);
  ```

- En la cabecera de la función `intercambio`, se indica que los parámetros formales `val1` y `val2` son punteros del tipo `int *` (puntero a entero), porque almacenarán las direcciones de `v1` y `v2`, respectivamente, en el momento de la llamada.

  ```
  void intercambio(int *val1, int *val2)
  ```

- En el cuerpo de la función `intercambio`, se accede a los valores de las variables `v1` y `v2` del programa principal a través de los punteros `val1` y `val2` (parámetros formales), utilizando el operador de indirección `*`. Obsérvese también que las únicas variables que se pueden manipular dentro del cuerpo de la función son los parámetros formales (`val1` y `val2`) y la variable local (`aux`).

  ```
  aux = *val1;
  *val1 = *val2;
  *val2 = aux;
  ```

La tabla 9.1 muestra un ejemplo de ejecución del programa anterior que incluye la evaluación paso a paso de la llamada a la función `intercambio`. Los pasos para evaluar la función `intercambio` son los mismos que los que se han presentado en el capítulo 8, sección 8.5. Para facilitar la explicación de la ejecución del programa, se numeran las líneas del código como sigue:

```
 1: #include <stdio.h>
 2: void intercambio(int *val1, int *val2);
 3: main()
 4: {
 5:   int v1, v2;
 6:
 7:   printf("Introduzca dos valores enteros (separados por
              un espacio): ");
 8:   scanf("%d %d%*c", &v1, &v2);
 9:   printf("Antes de la funcion intercambiar v1 = %d
                 v2 = %d\n", v1, v2);
10:   intercambio(&v1, &v2);
11:   printf("Despues de la funcion intercambiar v1 = %d
              v2 = %d\n",v1, v2);
12: }
13:
14: void intercambio(int *val1, int *val2)
15: {
16:   int aux;
17:
18:   aux = *val1;
19:   *val1 = *val2;
20:   *val2 = aux;
21:   printf("Dentro de la funcion intercambiar *val1 = %d
              *val2 = %d\n", *val1, *val2);
22: }
```

Tabla 9.1 Ejecución paso a paso del programa que intercambia el valor de dos variables.

Línea	Valor de las variables		Entrada por teclado	Salida en pantalla
	v1	v2		
5	?	?		
7	?	?		Introduzca dos valores enteros (separados por un espacio):
8	5	3	5 3	
9	5	3		Antes de la funcion intercambiar v1 = 5 v2 = 3
10 tabla 9.2	3	5		Dentro de la funcion intercambiar *val1 = 3 *val2 = 5
11	3	5		Despues de la funcion intercambiar v1 = 3 v2 = 5

Obsérvese que la tabla 9.1 hace referencia a la tabla 9.2, que contiene la ejecución paso a paso de la función `intercambio` correspondiente a la línea `10` del programa principal. La tabla muestra los valores de los parámetros formales (`val1` y `val2`), el valor de la variable local (`aux`) y el valor de las posiciones de memoria a las cuales apuntan los parámetros formales (`*val1` y `*val2`). La función no retorna ningún valor al programa principal a través de la sentencia `return`, por lo que no se incluye esta columna en la tabla.

Línea	Valor de los parámetros formales		Valor de la variable local	Valor al cual apuntan los parámetros formales		Salida en pantalla
	val1	val2	aux	*val1	*val2	
16	&v1	&v2	?	5	3	
18	&v1	&v2	5	5	3	
19	&v1	&v2	5	3	3	
20	&v1	&v2	5	3	5	
21	&v1	&v2	5	3	5	Dentro de la funcion intercambiar *val1 = 3 *val2 = 5

Tabla 9.2 Ejecución paso a paso de la llamada a `intercambio` de la línea 10 del programa

Obsérvese que el valor de los parámetros formales `val1` y `val2` son ahora las direcciones de las variables `v1` y `v2`, en lugar de los valores de estas variables (como ocurría en el paso de parámetros por valor). La función `intercambio` no modifica el valor de los parámetros formales, pero sí el contenido de las posiciones de memoria a los que apuntan, y por tanto, el valor de las variables `v1` y `v2` del programa principal.

La figura 9.5 ilustra la representación en la memoria de las variables `v1` y `v2` y de los parámetros formales `val1` y `val2`. En algunas posiciones de la memoria, se ha indicado la expresión para acceder a ese valor desde el `main` (expresión de la izquierda, `v1` y `v2`) y desde la función `intercambio` (expresión de la derecha, `*val1` y `*val2`).

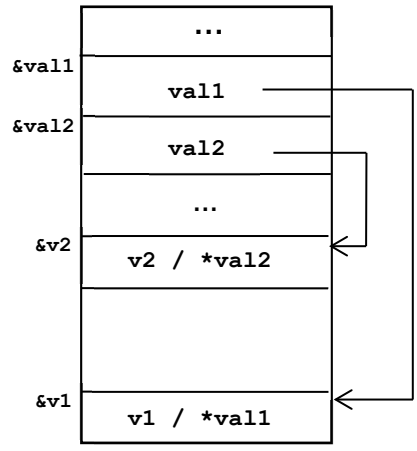


Fig. 9.5 Representación en la memoria de las variables `v1` y `v2` y de los parámetros formales `val1` y `val2`

9.4 Paso de parámetros por referencia de estructuras

En la sección anterior, se ha visto un ejemplo del paso de parámetros por referencia para tipos elementales. Ahora se verá cómo realizar el paso de parámetros por referencia de las estructuras, también con un ejemplo. El programa siguiente permite leer desde el teclado los datos del DNI de una persona, guardarlos en una variable de tipo estructura `tdni` y mostrar en pantalla un

mensaje con la información leída. Para implementar este programa, se utilizan dos funciones: la función `mostrar_dni`, que muestra por pantalla el DNI pasado como parámetro, y la función `leer_dni`, que lee del teclado los datos del DNI que introduce el usuario.

A la función `mostrar_dni` simplemente hay que pasarle como parámetro el DNI que se quiere mostrar en pantalla. Es suficiente con hacer una copia de los datos del DNI sobre el parámetro formal de la función. Por tanto, el parámetro de esta función lo pasaremos por valor.

Sin embargo, a la función `leer_dni` no es necesario pasarle datos; simplemente la función ha de devolver un resultado (el DNI leído del teclado). Este resultado lo puede devolver a través de la sentencia `return`, como se ha visto en el capítulo 8, o a través de un parámetro pasado por referencia, como se verá en este código. En este último caso, se pasa a la función la dirección de memoria de la variable donde ha de dejar el resultado (el DNI que lee del teclado).

```
#include <stdio.h>
typedef struct
{
  int num;
  char letra;
} tdni;

/* Prototipos de funciones */
void mostrar_dni(tdni d);    /* Función que muestra en pantalla
                                el número y la letra del dni d */

void leer_dni(tdni *d);      /* Función que lee desde el teclado
                                el número y la letra de un dni
                                y lo guarda en el dni apuntado
                                por d */

main()
{
  tdni nif;
  leer_dni(&nif);
  mostrar_dni(nif);
}

void mostrar_dni(tdni d)
{
  printf ("%d%c\n", d.num, d.letra);
}

void leer_dni(tdni *d)
{
  printf("Introduzca el numero y la letra del dni (separados
          por un espacio): ");
  scanf("%d %c%*c", &d->num, &d->letra);
}
```

Obsérvese que:

- En la llamada a la función `leer_dni` se utiliza el operador de referencia `&` para pasar a la función la dirección de la variable `nif`, que es donde se quiere que la función deje el resultado (el DNI que lee del teclado). Nótese además que la sintaxis es la misma que se ha visto en la sección anterior para realizar el paso de parámetros por referencia de tipos elementales.

  ```
  leer_dni(&nif);
  ```

- En la cabecera de la función `leer_dni` se indica que el parámetro formal `d` es un puntero a `tdni` porque almacenará la dirección de `nif` en el momento de la llamada.

  ```
  void leer_dni(tdni *d);
  ```

- En el cuerpo de la función `leer_dni`, para acceder a los campos de la estructura se utiliza el operador flecha `(->)` en lugar del operador punto (`.`). En el capítulo 6, sección 6.3, se ha visto que el operador punto (`.`) permite acceder a los campos de una estructura. Sin embargo, el parámetro formal `d` no es una estructura, sino un puntero a una estructura. Para acceder a los campos de la estructura a la cual apunta, hay que acceder primero al contenido de la dirección de memoria que guarda `d` (usando el operador de indirección `*`) y, a continuación, acceder al campo correspondiente con el operador punto. Así, para acceder a los campos `num` y `letra` de la estructura a la cual apunta `d`, hay que escribir `(*d).num` y `(*d).letra`, respectivamente. Los paréntesis son necesarios por una cuestión de precedencia entre el operador de indirección y el operador punto. El operador punto (`.`) tiene mayor precedencia que el operador `*` y primero es necesario acceder a la estructura (`*d`) y luego al campo que corresponda usando el operador punto (`.`).

  Sin embargo, el lenguaje C ofrece un operador para acceder a los campos de una estructura desde un puntero a esta. Es el operador flecha (`->`). Por tanto, también se puede acceder a los campos `num` y `letra` de la estructura a la cual apunta `d` de la manera siguiente:

  `d->num` y `d->letra`

  En este libro, se utiliza el operador flecha (`->`) en lugar del operador de indirección (`*`) y el operador punto (`.`) para acceder a los campos de una estructura desde un puntero a dicha estructura, puesto que su escritura es más simple. El anexo 9.A muestra la tabla de precedencia con todos los operadores que se utilizan en este libro.

- En el cuerpo de la función `leer_dni`, se llama a la función `scanf` a la cual se debe pasar por referencia el segundo y el tercer parámetro. Obsérvese que hay que pasar la dirección de memoria donde ha de dejar los datos que lee del teclado, es decir, la dirección de memoria

de cada uno de los campos de la estructura a la cual apunta el parámetro formal `d`, es decir, `&(d->num)` y `&(d->letra)`.

A diferencia del punto anterior, en este caso los paréntesis no son necesarios, ya que el operador `->` tiene mayor precedencia que el operador `&`.

```
scanf("%d %c%*c", &d->num, &d->letra);
```

– A la función `mostrar_dni` se le pasa el parámetro por valor. Por ello, el parámetro formal `d` es de tipo `tdni` y no puntero a `tdni`. En este caso, el acceso a los campos de la estructura se realiza utilizando el operador punto (`.`), como se ha visto en el capítulo 6, sección 6.3.

```
printf ("%d%c\n", d.num, d.letra);
```

La tabla 9.3 muestra un ejemplo de ejecución del programa anterior, que incluye la evaluación paso a paso de la llamada a las funciones: `leer_dni` y `mostrar_dni`. Los pasos para evaluar las funciones `leer_dni` y `mostrar_dni` son los mismos que los que se han presentado en el capítulo 8, sección 8.5.

Para facilitar la explicación de la ejecución del programa, se numeran las líneas del código como sigue:

```
 1: #include <stdio.h>
 2: typedef struct
 3: {
 4:   int num;
 5:   char letra;
 6: } tdni;
 7:
 8: void mostrar_dni(tdni d);
 9: void leer_dni(tdni *d);
10: main()
11: {
12:   tdni nif;
13:
14:   leer_dni(&nif);
15:   mostrar_dni(nif);
16: }
17:
18: void mostrar_dni(tdni d)
19: {
20:   printf ("%d%c\n", d.num, d.letra);
21: }
22:
23: void leer_dni(tdni *d)
24: {
25:   printf("Introduzca el numero y la letra del DNI (separados
            por un espacio): ");
26:   scanf("%d %c%*c", &d->num, &d->letra);
27: }
```

Línea	Valor de la variable		Entrada por teclado	Salida en pantalla
	nif.num	nif.letra		
12	?	?		
14 tabla 9.4	47882302	'N'	47882302 N	Introduzca el numero y la letra del DNI (separados por un espacio):
15 tabla 9.5	47882302	'N'		47882302N

Tabla 9.3 Ejecución del programa que lee y muestra el DNI.

Obsérvese que la tabla 9.3 hace referencia a las tablas 9.4 y 9.5, que indican, respectivamente, la ejecución paso a paso de las funciones `leer_dni` y `mostrar_dni`, correspondientes a las líneas `14` y `15` del programa principal. Cada una de estas tablas muestra el valor del parámetro formal (`d`), y en la tabla 9.4 también se muestra el valor de la posición de memoria a la cual apunta el parámetro formal (`d->num` y `d->letra`). Las funciones no retornan ningún valor al programa principal a través de la sentencia `return`, por lo que no se incluye en las tablas una columna con esta información.

Línea	Valor del parámetro formal	Valor al cual apunta el parámetro formal		Entrada por teclado	Salida en pantalla
	d	d->num	d->letra		
25	&nif	?	?		Introduzca el numero y la letra del DNI (separados por un espacio):
26	&nif	47882302	'N'	47882302 N	

Tabla 9.4 Ejecución paso a paso de la llamada a `leer_dni` de la línea 14 del programa

Línea	Valor del parámetro formal		Entrada por teclado	Salida en pantalla
	d.num	d.letra		
20	**47882302**	**'N'**		47882302N

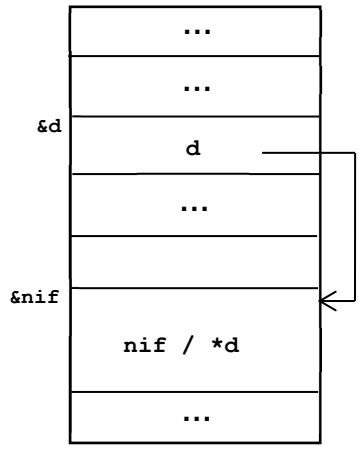
Tabla 9.5 Ejecución paso a paso de la llamada a `mostrar_dni` de la línea 15 del programa

Obsérvese que en la tabla 9.4 el valor del parámetro formal `d` es la dirección de memoria de la variable `nif`. La función `leer_dni` modifica el contenido de la posición de memoria a la que apunta el parámetro formal `d`. Sin embargo, obsérvese que en la tabla 9.5 el parámetro formal `d` es ahora una estructura (y no un puntero) que contiene la copia del contenido de la variable `nif` del programa principal (y no la dirección de esta variable).

La figura 9.6 ilustra la representación en la memoria de la variable `nif` y del parámetro formal `d` de la función `leer_dni`. En la posición de memoria donde se encuentra la estructura, se ha indicado la expresión para acceder a ese valor

desde el `main` (expresión de la izquierda, `nif`) y desde la función (expresión de la derecha, `*d`).

Fig. 9.6 Representación en memoria del parámetro formal `d` de la función `leer_dni`



En resumen, al diseñar una función hay que decidir primero cómo se pasa cada uno de los parámetros. El parámetro se pasa por referencia cuando la función necesita modificar la información almacenada en la variable declarada en la función que hace la llamada. El parámetro se pasa por valor cuando la función únicamente necesita consultar (y no modificar) el valor de la variable declarada en la función que realiza la llamada.

No obstante, cuando los parámetros son de tipo estructura y su tamaño en la memoria es grande (estructuras cuyos campos contienen vectores, por ejemplo), para que el programa se ejecute de manera eficiente, conviene pasar el parámetro por referencia, aunque no se modifique el contenido de la estructura. A pesar de esta sugerencia, en este libro no se aplica porque se quiere hacer más hincapié en saber distinguir cuándo un parámetro se debe pasar por valor y cuando por referencia, que en la eficiencia del programa. Por tanto, aquí se realiza el paso de parámetros por referencia de estructuras solo cuando se necesita modificar el valor de la estructura desde la función que ha sido llamada, sin importar que se trate de estructuras grandes que ocupan mucho espacio en la memoria.

9.5 Paso de parámetros por referencia de vectores

En las secciones anteriores, se ha visto cómo realizar el paso de parámetros por referencia de los tipos de datos elementales y de las estructuras. Ahora se explica cómo realizar el paso de parámetros por referencia de los vectores. En primer lugar, cabe indicar que los vectores en el lenguaje C solamente se pue-

den pasar por referencia y nunca por valor. La idea del paso de parámetros por referencia de vectores es exactamente la misma que la de otros tipos de datos (elementales y estructuras), pero se verá que la sintaxis es ligeramente diferente y, a veces, ello puede conllevar a confusiones.

Al igual que los tipos elementales y las estructuras, al pasar un vector por referencia lo que se hace es pasar a la función la dirección de memoria del vector, de manera que el parámetro formal de la función "apunte" al vector declarado en la función que hizo la llamada. A través del parámetro formal, se pueden consultar y/o modificar las diferentes posiciones del vector. A continuación, se proporciona un ejemplo que describe cómo realizar esto en el lenguaje de programación C.

El programa siguiente inicializa los elementos de un vector de enteros con el valor de una secuencia de números enteros introducida desde el teclado. El programa muestra por pantalla un mensaje con la información leída.

```
#include <stdio.h>
#define MAX 10

void mostrar_vector(int vec[MAX]);
/* Función que muestra por pantalla los MAX elementos de vec */

void inic_vector(int vec[MAX]);
/* Función que lee desde el teclado MAX elementos enteros
   y los guarda en vec */

main()
{
  int v[MAX];

  inic_vector(v);
  mostrar_vector(v);
}

void mostrar_vector(int vec[MAX])
{
  int i;

  for(i=0; i<MAX; i++)
    printf ("%d ", vec[i]);
  printf ("\n");
}

void inic_vector(int vec[MAX])
{
  int i;

  printf("Introduzca %d elementos enteros (separados
          por un espacio): ", MAX);
  for(i=0;i<MAX;i++)
    scanf("%d ", &vec[i]);

  scanf("%*c");
}
```

Obsérvese que:

- El paso de parámetros en la función `mostrar_vector` se realiza de la misma manera que en la función `inic_vector`. En este ejemplo, la función `mostrar_vector` no modifica el contenido de los elementos del vector dentro de la función (solo lo consulta), por lo que iría bien realizar el paso de parámetros por valor. Sin embargo, como se ha comentado anteriormente, en el lenguaje C los vectores siempre se pasan por referencia, independientemente de que sus elementos se modifiquen o no dentro del cuerpo de la función.

- En la llamada a las funciones `mostrar_vector` e `inic_vector`, no se utiliza explícitamente el operador de referencia `&` para pasar a la función la dirección del vector `v`, sino que directamente se escribe el nombre de la variable (`v`). En el lenguaje C, el nombre de una variable de tipo vector es equivalente a la dirección del elemento `0` del vector (la dirección del vector). Por tanto, `v` es equivalente a `&v[0]`. Las llamadas

  ```
  inic_vector(v);
  mostrar_vector(v);
  ```

  son equivalentes a

  ```
  inic_vector(&v[0]);
  mostrar_vector(&v[0]);
  ```

  donde se observa explícitamente que se está pasando la dirección del vector `v`. En este libro, sin embargo, se utiliza la expresión `v` en lugar de `&v[0]` para pasar la dirección del vector.

- La especificación del parámetro formal en la cabecera de las funciones `mostrar_vector` e `inic_vector` se realiza de la misma forma que la declaración de un vector. Obsérvese que también se incluye la longitud del vector.

  ```
  void mostrar_vector(int vec[MAX])
  void inic_vector(int vec[MAX])
  ```

  El lenguaje C permite especificar un parámetro formal que apunta a un vector de varias maneras: `int *vec`, `int vec[MAX]` o `int vec[]`. En cualquiera de las tres formas, el parámetro formal `vec` es un puntero a un vector. Sin embargo, dependiendo de cómo se especifique el parámetro formal `vec`, el acceso a los elementos del vector se debe realizar de una manera u otra. Si se especifica el parámetro formal como `int vec[MAX]` o `int vec[]`, dentro del cuerpo de la función se accederá a los elementos del vector utilizando directamente los corchetes e indicando el índice del elemento del vector que interesa consultar o modificar. Es decir, el acceso a los elementos del vector se realiza igual que como se ha explicado en el capítulo 7, sección 7.3.

```
printf ("%d ", vec[i]);
```

Sin embargo, si se especifica el parámetro formal como `int *vec`

```
void inic_vector(int *vec)
```

entonces, para referenciar el elemento que ocupa la posición `i` del vector, ahora es preciso utilizar aritmética de punteros.

```
printf ("%d ", *(vec+i));
```

En este libro, y por simplicidad, siempre se utiliza la primera notación para el paso de parámetros de vectores, es decir, los corchetes en lugar del puntero para referenciar un elemento del vector dentro del cuerpo de la función.

La figura 9.7 ilustra la representación en la memoria de la variable `v` y del parámetro formal `vec`**.** En cada posición de la memoria, se ha indicado la expresión para acceder a ese valor desde el `main` (expresión de la izquierda) y desde la función (expresión de la derecha). Obsérvese que el parámetro `vec` almacena la dirección de la variable `v` y, por lo que `vec` "apunta" al vector `v`. Además, la referencia a los elementos del vector dentro del cuerpo de la función se realiza utilizando el corchete e indicando el índice del elemento al cual interesa acceder. De esta forma, si se modifica `vec[i]` dentro del cuerpo de la función, se está modificando la posición de la memoria asociada al elemento que ocupa la posición `i` de la variable `v`, es decir, `v[i]`.

Por último, cabe añadir que, si ahora los elementos del vector son estructuras o vectores, el paso de parámetros se realiza igual que como se ha descrito en este ejemplo.

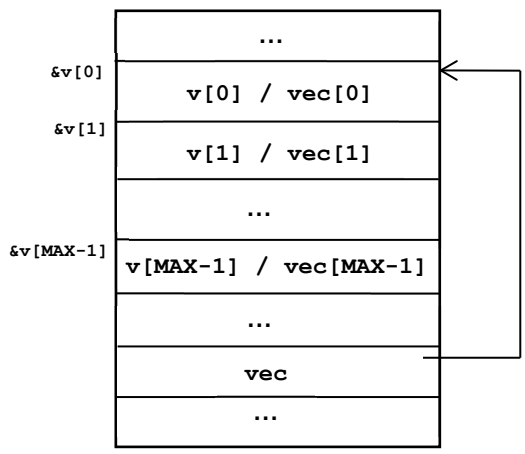


Fig. 9.7
Representación en la memoria del parámetro formal `vec` de la función `mostrar_vector` o `inic_vector`

Finalmente, en la tabla 9.6 se hace un resumen de los tres ejemplos anteriores, que muestra la diferencia sintáctica del paso de parámetros por referencia

de tipos elementales, estructuras y vectores. En la tabla, se muestra la llamada a cada una de las funciones y la definición de estas, donde se observa el acceso a los datos a los cuales apunta el parámetro formal dentro del cuerpo de la función.

Tabla 9.6 Resumen de prototipos y llamadas a funciones: paso por referencia

Paso por referencia de:	Llamada a función	Definición de la función

Tipo elemental	`main()`
	`{`
	`  int v1, v2;`
	
	`  intercambio(&v1,&v2);`
	`}`
		`void intercambio (int *val1,`
		`  int *val2)`
		`{`
		`  ...`
		`  aux  = *val1;`
		`  *val1 = *val2;`
		`  *val2 = aux;`
		`}`

Estructura	`main()`
	`{`
	`  tdni nif;`
	
	`  leer_dni(&nif);`
	`}`
		`void leer_dni(tdni *d)`
		`{`
		`  ...`
		`  scanf("%d %c%*c",`
		`  &d->num, &d->letra);`
		`}`

Vector	`main()`
	`{`
	`  int v[MAX];`
	
	`  inic_vector(v);`
	`}`
		`void inic_vector(int vec[MAX])`
		`{`
		`  ...`
		`  scanf("%d",&vec[i]);`
		`}`

Obsérvese que, en las llamadas a las funciones, el operador `&` se utiliza de forma explícita cuando se pasa por referencia un dato de tipo elemental o de tipo estructura (`&v1`, `&v2`, `&nif`). Sin embargo, en el caso de los vectores, el operador `&` no se utiliza de forma explícita, porque se usa simplemente el nombre del vector (`v`) que equivale a la dirección del primer elemento (`&v[0]`).

Obsérvense también, en la definición de la función, los tres puntos siguientes:

- Si se ha pasado por referencia un dato de tipo elemental, entonces el parámetro formal es un puntero a dicho tipo (`int *val1`, `int *val2`). Si se quiere modificar el valor al cual apunta el parámetro formal, entonces se debe utilizar el operador `*`, por ejemplo `*val1`.

  Por otra parte, si se quisiera obtener la dirección del valor al cual apunta, entonces hay que aplicar el operador `&` a la expresión anterior (`&(*val1)`) o simplemente utilizar el nombre del parámetro formal, ya que este almacena dicha dirección (`val1`).

- Si se ha pasado por referencia un dato de tipo estructura, entonces el parámetro formal es un puntero a dicha estructura (`tdni *d`). Si se quiere modificar el valor de alguno de los campos de la estructura a la cual apunta el parámetro formal, entonces se debe utilizar el operador `->`, por ejemplo `d->letra`. Pero si se quisiera obtener la dirección de alguno de los campos, entonces habría que aplicar el operador `&` a la expresión anterior (`&d->letra`).

Por otra parte, si se quisiera la dirección de toda estructura, la expresión sería `&(*d)`, o simplemente `d`, el nombre del parámetro formal, ya que este almacena dicha dirección.

– Si se ha pasado por referencia un dato de tipo vector, entonces el parámetro formal es un puntero a dicho vector y se especifica en la cabecera como la declaración normal de un vector (`int vec[MAX]`). Si se quiere modificar el valor de alguno de los elementos del vector al cual apunta el parámetro formal `vec`, entonces se han de utilizar los corchetes `[]` e indicar el índice del elemento que se quiere modificar, por ejemplo `vec[i]`. Pero si se quiere obtener la dirección de algún elemento concreto del vector, entonces hay que aplicar el operador `&` a la expresión anterior (`&vec[i]`). Por último, si se quisiera la dirección de todo el vector, la expresión sería simplemente `vec`, el nombre del parámetro formal, ya que este almacena dicha dirección.

9.6 Ejemplos de uso de funciones con paso por referencia

En esta sección se muestran algunos ejemplos de uso de funciones con paso de parámetros por referencia. En primer lugar, se presentan los algoritmos básicos que se han explicado en el capítulo 7, sección 7.4, utilizando funciones en la implementación. En estos ejemplos, se usa el paso de parámetros por referencia de tipos elementales y estructuras. En segundo lugar, se muestra un ejemplo de uso en el que es necesario pasar vectores por referencia.

Ejemplo 1

Los algoritmos que se presentan a continuación son los algoritmos de búsqueda de un elemento en un vector, inserción de un elemento en un vector, eliminación de un elemento del vector y ordenación de los elementos del vector. Para el algoritmo de búsqueda, se muestran dos versiones diferentes, en función de si los elementos del vector se encuentran previamente ordenados bajo algún criterio. Para el algoritmo de inserción, solo se muestra la versión para un vector desordenado y se deja como ejercicio (sección 9.7, ejercicio 5) la versión para un vector ordenado.

Para simplificar la exposición de los algoritmos, se utiliza, igual que en el capítulo 7, sección 7.4, una estructura `tvector` con dos campos: un vector de enteros y un entero que indica el número de elementos almacenados en el vector. La estructura `tvector` se define de la manera siguiente:

```
#define MAX 100

typedef struct
{
   int nelem;         /* Número de elementos en el vector */
   int vector[MAX];   /* Vector de enteros */
} tvector;
```

Búsqueda de un elemento en un vector

A continuación, se muestra el código para buscar la primera aparición de un elemento en un vector desordenado.

```
/* Búsqueda (primera aparición) en un vector DESORDENADO */

#include <stdio.h>

int busqueda_vector_desordenado(tvector v, int elem, int *pos);

main()
{
  tvector v={10,{23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int elem, pos, encontrado;

  printf("Introduzca el elemento que buscas: ");
  scanf("%d%*c", &elem);

  encontrado = busqueda_vector_desordenado(v, elem, &pos);
  if (encontrado==0)
    printf("Elemento NO encontrado\n");
  else
    printf("Elemento encontrado en la posicion %d\n", pos);
}

/* La función busqueda_vector_desordenado busca el elemento elem
   en el campo vector de la estructura v. Si lo encuentra,
   devuelve 1 a través del return y la posición donde se
   encuentra a través del parámetro pos. Si no lo encuentra,
   devuelve 0 a través del return */

int busqueda_vector_desordenado(tvector v, int elem, int *pos)
{
  int encontrado, i;

  encontrado = 0;
  i = 0;
  while (i<v.nelem && encontrado==0)
  {
    if (v.vector[i]==elem)
      encontrado=1;
    else
      i = i+1;
  }

  *pos = i;
  return(encontrado);
}
```

A continuación, se ilustra la búsqueda de un elemento en un vector cuyos elementos están ordenados de mayor a menor.

```
/* Búsqueda (primera aparición) en un vector ORDENADO */

#include <stdio.h>

int busqueda_vector_ordenado(tvector v, int elem, int *pos);
```

```
main()
{
  tvector v={10, {322, 88, 45, 44, 39, 34, 23, 22, 10, 2 }};
  int elem, encontrado, pos;

  printf("Introduzca el elemento que buscas: ");
  scanf("%d%*c", &elem);

  encontrado = busqueda_vector_ordenado(v, elem, &pos);
  if (!encontrado)
    printf("Elemento NO encontrado\n");
  else
    printf("Elemento encontrado en la posicion %d\n", pos);
}

/* La función busqueda_vector_ordenado busca el elemento elem en
   el campo vector de la estructura v, sabiendo que está ordenado
   de mayor a menor. Si lo encuentra, devuelve 1 a través del
   return y la posición donde se encuentra a través del parámetro
   pos. Si no lo encuentra, devuelve 0 a través del return */

int busqueda_vector_ordenado(tvector v, int elem, int *pos)
{
  int i, encontrado=0;

  i = 0;
  while (i<v.nelem && v.vector[i]>elem)
    i = i+1;

  if (i<v.nelem && v.vector[i]==elem)
  {
    *pos = i;
    encontrado = 1;
  }

 return(encontrado);
}
```

Obsérvese que, tanto en las funciones de `busqueda_vector_desordenado` como en la `busqueda_vector_ordenado`, las funciones retornan dos valores: uno a través de la sentencia `return`, que indica si se ha encontrado o no el elemento buscado y otro a través del parámetro `pos` (pasado por referencia), que indica la posición donde está el elemento, en caso de encontrarlo.

Inserción de un elemento en un vector

A continuación, se indica el código para insertar un elemento en un vector desordenado. Si el vector no sigue ningún criterio de ordenación, es suficiente con insertar el elemento en la primera posición libre del vector.

```
/* Inserta un elemento en un vector DESORDENADO. La inserción se
   realiza en la primera posición libre del vector */

#include <stdio.h>

int insertar_en_vector_desordenado(tvector *v, int elem);
```

```
main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int elem, i, ok;

  printf("Introduzca el elemento a insertar:");
  scanf("%d%*c", &elem);

  ok = insertar_en_vector_desordenado(&v, elem);
  if (ok==0)
    printf ("No se pudo insertar. El vector esta lleno.\n");
  else
  {
    printf("El elemento ha sido insertado en la ultima
            posicion\n");

    printf ("VECTOR: ");
    for (i=0; i< v.nelem-1; i++)
      printf("%d, ", v.vector[i]);
    printf("%d\n", v.vector[i]);
  }
}

/* La función insertar_en_vector_desordenado inserta el elemento
   elem en el campo vector de la estructura apuntada por el
   parámetro v. Si no puede hacer la inserción porque el vector
   está lleno, devuelve 0. En caso contrario, inserta elem en la
   primera posición libre del vector y devuelve 1 */
int insertar_en_vector_desordenado (tvector *v, int elem)
{
  int insertado;

  if (v->nelem == MAX)
   insertado = 0;
  else
  {
    /* Como el vector no está ordenado, se inserta el elemento
       al final */
    v->vector[v->nelem] = elem;
    v->nelem = v->nelem+1;
    insertado = 1;
  }

  return(insertado);
}
```

Obsérvese que, en la función `insertar_en_vector_desordenado`, se realiza el paso por referencia de la estructura de tipo `tvector`. Además, en el cuerpo de esta función se accede a los campos de la estructura utilizando el operador `->`. Finalmente, el resultado devuelto a través de la sentencia `return` indica si el elemento `elem` (pasado a la función por valor) ha podido insertarse en el vector.

Eliminación de un elemento

Cuando se quiere eliminar un elemento de un vector, simplemente hay que conocer la posición que ocupa dicho elemento en el vector. Si la posición es válida, es decir, está dentro del rango `[0, v.nelem-1]`, el elemento se puede eliminar.

Nótese que el código es independiente de si el vector está ordenado o no.

```
/* Elimina un elemento del vector */

#include <stdio.h>

int eliminar_elemento_de_vector(tvector *v, int pos);

main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};
  int i, pos, ok;

  printf("Introduzca la posicion del elemento a eliminar:");
  scanf("%d%*c", &pos);

  ok = eliminar_elemento_de_vector(&v, pos);
  /* Se comprueba si la posición es válida  */

  if (ok==0)
    printf ("No se pudo eliminar. Posicion no valida.\n");
  else
  {
    printf("Elemento ha sido eliminado\n");

    printf ("VECTOR: ");
    for (i=0; i< v.nelem-1; i++)
      printf("%d, ", v.vector[i]);
    printf("%d\n", v.vector[i]);
  }
}

/* La función eliminar_elemento_de_vector elimina el elemento
   que ocupa la posición pos del campo vector de la estructura
   apuntada por el parámetro v. Si no puede hacer la eliminación
   porque la posición no es válida, la función devuelve 0. En
   caso contrario, devuelve 1 */

int eliminar_elemento_de_vector(tvector *v, int pos)
{
  int i, eliminado;

  if (pos<0 || pos>=v->nelem)
    eliminado = 0;
  else
  {
    /* Eliminar elemento del vector */
    for (i=pos; i<v->nelem-1; i++)
      v->vector[i] = v->vector[i+1];
    v->nelem = v->nelem-1;
    eliminado = 1;
```

```
  }
  return(eliminado);
}
```

Igual que en el caso de la función `insertar_en_vector_desordenado`, obsérvese que, en la función `eliminar_elemento_de_vector`, también se realiza el paso por referencia de la estructura de tipo `tvector`. Además, el acceso a los campos de la estructura en el cuerpo de la función se realiza mediante el operador `->`.

Nótese también que el parámetro `pos`, que indica la posición del elemento a eliminar del vector, está pasado por valor, ya que simplemente es necesario conocer el valor del índice y no es preciso modificarlo.

Finalmente, el resultado devuelto a través de la sentencia `return` indica si el elemento de la posición `pos` se ha podido eliminar del vector.

Ordenación

Finalmente, se muestra el algoritmo de selección directa para ordenar los elementos de un vector de enteros de mayor a menor.

```
/* Ordenar los elementos de un vector de mayor a menor */

#include <stdio.h>

void mostrar_vector(tvector v);
void ordenar(tvector *v);

main()
{
  tvector v={10, {23, 45, 2, 44, 88, 39, 322, 34, 22, 10}};

  printf("Vector original: ");
  mostrar_vector(v);

  ordenar(&v);

  printf("Vector ordenado: ");
  mostrar_vector(v);
}

/* Se muestra el vector por pantalla */
void mostrar_vector (tvector v)
{
  int i;

  for (i=0; i<v.nelem-1; i++)
    printf ("%d - ", v.vector[i]);
  printf ("%d\n", v.vector[i]);
}

/* La función ordenar, realiza la ordenación de los elementos
   del campo vector de la estructura apuntada por el parámetro v
   utilizando el método de selección directa */
```

```
void ordenar(tvector *v)
{
  int i,j, posmax, aux;
  for (i=0; i<v->nelem-1; i++)
  {
    /* Se busca la posición del elemento
       mayor desde i a v.nelem-1 */
    posmax = i;
    for (j=i+1; j<v->nelem; j++)
      if (v->vector[j]>v->vector[posmax])
        posmax = j;

    /* Se intercambia la posición i y posmax */
    aux = v->vector[i];
    v->vector[i] = v->vector[posmax];
    v->vector[posmax] = aux;
  }
}
```

Igual que en los casos de inserción y de eliminación, en la ordenación también se realiza el paso por referencia de la estructura de tipo `tvector`, ya que se requiere modificar el orden de los elementos del campo `vector` para realizar la ordenación. Igual que antes, los accesos a los campos de la estructura en el cuerpo de la función se realizan utilizando el operador `->`.

Finalmente, la función `ordenar` no retorna ningún valor al programa principal a través de la sentencia `return`.

Ejemplo 2

Como segundo ejemplo de uso, se muestra un programa que manipula directamente vectores de caracteres. El programa lee una frase acabada en punto desde el teclado, la invierte y la vuelve a mostrar por pantalla. El punto debe quedar almacenado al final de la frase invertida. Si la frase introducida por el teclado es muy larga, solamente se leen los primeros `MAXFR-1` caracteres.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca una frase acabada en punto: Esto es una frase.
La frase invertida es: esarf anu se otsE.
```

El código del programa es:

```
#include <stdio.h>
#define MAXFR 300

void leer_frase(char f[MAXFR]);
void invertir_frase(char f[MAXFR]);
void mostrar_frase(char f[MAXFR]);
```

```
main()
{
  char frase[MAXFR];

  leer_frase(frase);

  invertir_frase(frase);

  printf("La frase invertida es: ");
  mostrar_frase(frase);
}

/* Se lee desde el teclado una frase acabada en '.' y
   se almacena en f */
void leer_frase(char f[MAXFR])
{
  int i;

  printf("Introduzca una frase acabada en punto: ");
  i = 0;
  scanf("%c", &f[i]);

  while (i<MAXFR-1 && f[i]!='.')
  {
    i = i+1;
    scanf("%c", &f[i]);
  }
  if (i==MAXFR-1)
    f[i] = '.';
}

/* Se invierte la frase. Se debe tener en cuenta que el punto
   ha de quedar al final de la frase */
void invertir_frase(char f[MAXFR])
{
  int i, j;
  char c;

  /* Se guarda en j la posición donde está el punto */
  j = 0;
  while (f[j]!='.')
    j = j+1;

  /* Se invierten los elementos de la frase */
  i = 0;
  j = j-1;
  while (i<j)
  {
    c = f[j];
    f[j] = f[i];
    f[i] = c;
    j = j-1;
    i = i+1;
  }
}

/* Se muestra la frase por pantalla (incluido el punto) */
```

```
void  mostrar_frase(char f[MAXFR])
{
   int j;

   for (j=0; f[j]!='.'; j++)
      printf("%c", f[j]);
   printf("%c\n", f[j]);
}
```

Obsérvese que, en las tres funciones, se ha pasado el vector `frase` de la misma manera (por referencia), ya que el lenguaje C no permite pasarlo por valor. Además, el acceso a los elementos del vector dentro de las funciones se realiza de la misma manera que el acceso a un vector declarado localmente dentro de la función, es decir, indicando entre corchetes la posición del elemento al cual se quiere acceder.

9.7 Ejercicios

1. Complete los códigos siguientes, escribiendo las sentencias necesarias para compilar y ejecutar correctamente el programa.

a. Considere el mismo programa presentado en el capítulo 8, sección 8.7, ejercicio 1 que realiza la conversión a pulgadas de un valor leído (desde el teclado) expresado en centímetros, pero ahora incluyendo algunas modificaciones en los prototipos de las funciones.

```
#include <stdio.h>
#define CMT_POR_INCH 25.4

/* Complete el prototipo de la función leer_valor */
       void leer_valor (______);

/* Complete el prototipo de la función cmts_a_pulgadas */
       void cmts_a_pulgadas(________,________);
 main()
 {
  float cm, pul;

   /* Realice la llamada a la función leer_valor para
     leer el valor de los centímetros a convertir */

    ____________________________________

     /* Realice la llamada a la función cmts_a_pulgadas
        para convertir a pulgadas el valor de los
        centímetros leído*/

      ____________________________________

  printf("%.2f cmts son %.2f puldadas\n", cm, pul);
}
void leer_valor(________)
```

```
{
   printf("Introduzca cmts: ");

   /* Complete la llamada a la función scanf */

scanf("%f%*c", _________);
}

 /* Complete la cabecera de la función cmts_a_pulgadas */
void cmts_a_pulgadas(________,________)
{

  /* Complete el cuerpo de la función cmts_a_pulgadas
     sabiendo que 1 pulgada = 1 cm * CMT_POR_INCH */

  ____________________________________
}
```

b. Considere el programa siguiente, que manipula estructuras y vectores. No es necesario conocer qué hace exactamente el programa para responder al problema.

```
#define MAX 10
typedef struct
{
  int dia;
  int mes;
  int anyo;
} tfecha;

void f(int *val, tfecha *f);
void f1(int *val, tfecha v[MAX]);
int f2(tfecha v[MAX]);

main()
{
  int a;
  tfecha fech[MAX];

  /* Realice la llamada a la función f2 */

  ____________________________________
}

void f(int *val, tfecha *f)
{
  ...
}

void f1(int *val, tfecha v[MAX])
{
  int i;

  ...
  /* Guarde el valor leído de teclado en la dirección
     apuntada por val */
```

```
  scanf("%d%*c",_____________);

  for (i=0;i<MAX;i++)
  {
    /* Realice la llamada a la función f para cada
       elemento del vector v */

    ____________________________________
  }
}

int f2(tfecha v[MAX])
{
  int val;

  /* Realice la llamada a la función f1 */

   ____________________________________
}
```

2. Utilizando la función `swap`, que intercambia el valor de dos variables:

```
void swap(int *x, int *y)
{
  int aux;

  aux = *x;
  *x = *y;
  *y = aux;
}
```

Escriba un programa que rote el valor de tres variables `a`, `b`, `c`. Es decir, al final de la ejecución, la variable `a` ha de guardar el valor original de `b`, la variable `b` el valor original de `c` y, finalmente, la variable `c` guardará el valor original de `a`.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca el valor de a, b, c: 5 4 3

Valores rotados: a=4, b=3, c=5
```

3. Dado el prototipo de la función siguiente:

```
int cambio_base (int val, int base, int *res);
/* Esta función devuelve, a través del parámetro res, el valor del parámetro val en la base de numeración base. Suponga que val es estrictamente positivo y está en base 10 y que el parámetro base es un valor entre 2 y 9. Además, la función devuelve, a través del return, el número de dígitos del valor calculado */
```

a. Defina la función `cambio_base`.

b. Escriba un programa en lenguaje C, utilizando la función `cambio_base`, que permita al usuario realizar una serie de cambios de base. Primero, se pide al usuario cuántos cambios de base desea realizar. A partir de aquí, se le pide, uno a uno, el valor en base diez y la nueva base de numeración. El programa muestra en pantalla, para cada uno de los datos, el valor en la nueva base y el número de dígitos del nuevo valor. Además, el programa controla que el valor introducido sea un valor estrictamente positivo y la base esté entre `2` y `9`. Si no es así, aparece un mensaje en pantalla y se vuelven a pedir los datos.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca el numero de cambios de base que desea realizar: 3

Cambio de base 0 - Introduzca el valor en base diez y la nueva
base: 23 0

Error en los datos introducidos
Introduzca de nuevo el valor en base diez (>0) y la nueva base
(2-9): 23 1

Error en los datos introducidos
Introduzca de nuevo el valor en base diez (>0) y la nueva base
(2-9): 23 2

23 en base 10 = 10111 en base 2 (5 dígitos)

Cambio de base 1 - Introduzca el valor en base diez y la nueva
base: 54 9

54 en base 10 = 60 en base 9 (2 dígitos)

Cambio de base 2 - Introduzca el valor en base diez y la nueva
base: 3456 7

3456 en base 10 = 13035 en base 7 (5 dígitos)
```

4. Dado el siguiente tipo de dato `tfecha`, los prototipos de las funciones y las definiciones de las funciones `es_bisiesto` y `mostrar_fecha`:

```
typedef struct
{
  int dia, mes, anyo;
} tfecha;

int es_bisiesto (int anyo);
int leer_fecha(tfecha *f);
void  mostrar_fecha(tfecha f);
void  sumar_dias_fecha(tfecha *f, int nd);

int es_bisiesto (int anyo)
{
  int bisiesto=0;

  if (anyo%400==0 || (anyo%4==0 && anyo%100!=0))
    bisiesto = 1;

  return(bisiesto);
```

```
}

void  mostrar_fecha(tfecha f)
{
  printf("%02d/%02d/%02d", f.dia, f.mes, f.anyo);
}
```

a. Defina la función `leer_fecha` que lee una fecha del teclado y comprueba que la fecha sea válida, es decir, que el número del mes esté entre `1` y `12` y que el número del día sea correcto en función del mes. Téngase en cuenta si el año es bisiesto. La función devuelve la fecha leída a través del parámetro `f`, y retorna `1` si la fecha es válida y `0` en caso contrario.

b. Defina la función `sumar_dias_fecha` que, dada una fecha `f` pasada como parámetro, actualiza dicha fecha sumándole tantos días como indica el parámetro `nd`. Téngase en cuenta el número de días exactos que tiene cada mes, considerando también los años bisiestos. Así, por ejemplo, si se suman `60` días al `28/02/2000`, se obtiene la fecha `28/04/2000` porque el `2000` fue bisiesto. Si se suman `60` días al `27/02/2001`, se obtiene el `28/04/2001` porque el `2001` no fue bisiesto. Finalmente, si se suman `60` días al `30/12/2000`, se obtiene el `28/02/2001`.

c. Escriba el programa principal para que lea del teclado `10` fechas y un número de días concreto y muestre por pantalla las `10` fechas actualizadas con la suma de esos días. Se comprobará que sea válida cada fecha que el usuario introduzca. Si no lo es, se pedirá que la vuelva a introducir. A continuación, se muestra un ejemplo de ejecución.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca 10 fechas validas:
Fecha 0: 31/02/2000
Fecha no valida. Vuelva a introducirla.
Fecha 0: 29/02/2000
Fecha 1: 29/02/2001
Fecha no valida. Vuelva a introducirla.
Fecha 1: 28/02/2000
Fecha 2: 27/02/2001
Fecha 3: 31/03/2003
Fecha 4: 12/09/1999
Fecha 5: 30/12/2000
Fecha 6: 29/11/2999
Fecha 7: 16/03/2111
Fecha 8: 23/03/2000
Fecha 9: 12/02/2000
Introduzca el numero de dias que quiere sumar a las fechas:
60

Fechas:
29/02/2000 + 60 dias = 29/04/2000
28/02/2000 + 60 dias = 28/04/2000
27/02/2001 + 60 dias = 28/04/2001
31/03/2003 + 60 dias = 30/05/2003
```

```
12/09/1999 + 60 dias = 11/11/1999
30/12/2000 + 60 dias = 28/02/2001
29/11/2999 + 60 dias = 28/01/3000
16/03/2111 + 60 dias = 15/05/2111
23/03/2000 + 60 dias = 22/05/2000
12/02/2000 + 60 dias = 12/04/2000
```

5. Dado el siguiente tipo de dato `tvector` y los prototipos de funciones:

```
#define MAX 100

typedef struct
{
  int nelem;         /* Número de elementos en el vector */
  int vector[MAX];   /* Vector de enteros */
} tvector;

void leer_vector (tvector *v);
void mostrar_vector (tvector v);
int insertar_en_vector_ordenado (tvector *v, int elem);
int eliminar_elementos_de_vector (tvector *v, int elem);
```

   a. Defina la función `mostrar_vector` que muestra por pantalla los elementos del vector `v` separados por comas. Por ejemplo, si el vector contiene tres elementos: `33`, `8` y `21`, la función muestra por pantalla los elementos del vector en el formato siguiente:

```
VECTOR: 33,8,21
```

   b. Defina la función `insertar_en_vector_ordenado` que inserta el elemento `elem` en el vector apuntado por el parámetro `v`. Si no puede hacer la inserción del elemento porque el vector está lleno, la función devuelve `0`. En caso contrario, inserta `elem` en el vector de la estructura apuntada por `v` manteniendo el orden (de menor a mayor) y devuelve `1`.

   c. Defina la función `leer_vector` que lee del teclado un vector de enteros desordenado y deja los elementos ordenados de menor a mayor en el vector de la estructura apuntada por el parámetro `v`. La función pide primero al usuario que introduzca el número de elementos del vector y, a continuación, pide los elementos. Hay que comprobar que el número de elementos sea correcto (entre `1` y `MAX`). Mientras no sea correcto, se pedirá al usuario que introduzca el dato de nuevo.

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca el numero de elementos en el vector (MAX. 100):
101
Error en el valor introducido.
Introduzca el numero de elementos en el vector (MAX. 100): -3
Error en el valor introducido.
Introduzca el numero de elementos en el vector (MAX. 100 ): 3
Introduzca los elementos del vector: 33 8 21
```

   d. Defina la función `eliminar_elementos_de_vector` que elimina todos los elementos iguales al parámetro `elem` del vector apuntado por el parámetro

`v`. La función retorna el número de elementos del vector que han sido eliminados.

`e`. Escriba el programa principal para que salga continuamente por pantalla el menú de opciones siguiente:

```
1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir
```

y ejecute la opción escogida por el usuario. La opción `1` pide al usuario que introduzca un vector de enteros. Primero, pide el número de elementos y, a continuación, los elementos del vector. Los elementos se introducen desordenados, pero al leerlos se ordenan de menor a mayor en el vector. La opción `2` muestra el vector (ya ordenado) por pantalla. La opción `3` pide al usuario que introduzca un elemento y lo inserta en el vector manteniendo el orden. En caso de no poder insertar el elemento porque el vector está lleno, ha de aparecer en pantalla un mensaje que lo indique. La opción `4` pide al usuario un elemento y elimina todas las apariciones de ese elemento en el vector. Ha de indicarse en pantalla la cantidad de elementos eliminados. La opción `5` permite finalizar el programa. Si el usuario selecciona esta opción, el programa le pregunta si está seguro de querer salir. Si indica que sí, el programa finalizará. En caso contrario, el programa volverá a mostrar el menú de opciones. Finalmente, si el usuario escoge una opción incorrecta, se mostrará un mensaje por pantalla indicando el error y volverá a mostrar el menú. A continuación, se proporciona un ejemplo de ejecución del programa.

Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 1
Introduzca el numero de elementos en el vector (MAX. 100 ): 101
Error en el valor introducido.
Introduzca el numero de elementos en el vector (MAX. 100 ): 3
Introduzca los elementos del vector: 33 8 21

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 2
```

```
VECTOR: 8,21,33

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 6

Opcion incorrecta. Elija de nuevo.

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 3
Introduzca el elemento a insertar: 21

Elemento insertado.

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 2

VECTOR: 8,21,21,33

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 4
Introduzca el elemento a eliminar: 21

Se han eliminado 2 elementos.

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 4
Introduzca el elemento a eliminar: 15

Se han eliminado 0 elementos.

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
```

```
5- Salir

Elija una opcion: 2

VECTOR: 8,33

1- Leer vector
2- Mostrar vector ordenado
3- Insertar elemento
4- Eliminar elemento
5- Salir

Elija una opcion: 5

Ha elegido la opcion Salir del programa. Esta seguro (s/n): s
```

6. Escriba un programa en C que determine cuántas palabras palíndromas hay en una frase acabada en `'.'`. Téngase en cuenta que entre palabras puede haber más de un espacio o signos de puntuación.

   Para la implementación, defina primero las funciones siguientes y luego utilícelas para construir el programa principal.

```
#define MAXFR 300
#define MAXPAL 20

typedef char tfrase[MAXPAL];

void leer_frase(tfrase f);
/* Lee una frase acabada en punto del teclado y la
   almacena en f */

int extraer_palabra(tfrase f, char p[MAXPAL], int *i);
/* Extrae de la frase f la palabra a partir de la posición
   apuntada por i (índice) y la almacena en p. Actualiza
   el índice para que apunte al inicio de la palabra
   siguiente. La función devuelve la longitud de la
   palabra extraída */

int palindroma(char p[MAXPAL], int lon);
/* Esta función devuelve 1 si la palabra almacenada en p y
   de longitud lon es palíndroma. Devuelve 0 en caso
   contrario. */

void mostrar_palabra(char p[MAXPAL], int lon);
/* Muestra en pantalla la palabra p de longitud lon */
```

   Ejemplo de ejecución (en negrita, los datos que el usuario introduce):

```
Introduzca una frase acabada en punto:
Ellos pueden reconocer a Ana cuando narran la historia.

Palabras palindromas:
reconocer
a
```

```
Ana
narran
En total, hay 4 palabras palindromas.
```

9.8 Respuesta a los ejercicios propuestos

1.

a.

```
#include <stdio.h>
#define CMT_POR_INCH 25.4

/* Complete el prototipo de la función leer_valor */
void leer_valor(float *cm);

/* Complete el prototipo de la función cmts_a_pulgadas */
void cmts_a_pulgadas(float cm, float *pul);

main()
{
  float cm, pul;

  /* Realice la llamada a la función leer_valor para
     leer el valor de los centímetros a convertir */
  leer_valor(&cm);

/* Realice la llamada a la función cmts_a_pulgadas
   para convertir a pulgadas el valor de los
    centímetros leído*/
  cmts_a_pulgadas(cm, &pul);

  printf("%.2f cmts son %.2f pulgadas\n", cm, pul);
}

void leer_valor(float *cm)
{
  printf("Introduzca cmts: ");

  /* Complete la llamada a la función scanf */
  scanf("%f%*c", &(*cm));    /* o  scanf("%f%*c", cm); */
}

/* Complete la cabecera de la función cmts_a_pulgadas */
void cmts_a_pulgadas(float cm, float *pul)
{
  /* Complete el cuerpo de la función cmts_a_pulgadas
     sabiendo que 1 pulgada = 1 cm * CMT_POR_INCH */
  *pul = cm*CMT_POR_INCH;
}
```

b.

```
#include <stdio.h>
#define MAX 10
typedef struct
{
```

```
  int dia;
  int mes;
  int anyo;
} tfecha;

void f(int *val, tfecha *f);
void f1(int *val, tfecha v[MAX]);
int f2(tfecha v[MAX]);

main()
{
  int a;
  tfecha fech[MAX];

  /* Realice la llamada a la función f2 */
  a = f2(fech);
}

void f(int *val, tfecha *f)
{

  ...

}

void f1(int *val, tfecha v[MAX])
{
  int i;

  ...
  /* Guarde el valor leído de teclado en la dirección
     apuntada por val */
  scanf("%d%*c",&(*val));   // scanf("%d%*c",val);

  for (i=0;i<MAX;i++)
  {
    /* Realice la llamada a la función f para cada
       elemento del vector v */
    f(&(*val), &v[i]);        // f(val, &v[i]);
  }
}

int f2(tfecha v[MAX])
{
  int val;

  /* Realice la llamada a la función f1 */
  f1(&val, v);
  return(1);
}
```

2.

```
#include <stdio.h>

void swap (int *x, int *y);

main()
{
  int a, b, c;

  printf("Introduzca el valor de a, b, c: ");
```

```
  scanf("%d%*c%d%*c%d%*c", &a, &b, &c);
  swap(&a, &b);
  swap(&b, &c);
  printf("\nValores rotados: a=%d, b=%d, c=%d\n", a, b, c);
}

void swap(int *x, int *y)
{
  int aux;

  aux = *x;
  *x = *y;
  *y = aux;
}
```

3.

a.

```
int cambio_base (int val, int base, int *res)
{
  int coc, resto, aux, cont;

  coc = val;
  *res = 0;
  aux = 1;
  cont = 0;
       while (coc!=0)
  {
    resto = coc%base;
    *res = *res + aux*resto;
    coc = coc/base;
    aux = aux*10;
    cont = cont+1;
  }

       return(cont);
}
```

b.

```
#include <stdio.h>

int cambio_base (int val, int base, int *res);

main()
{
  int n, i, val, base, ndig, res;

  printf("Introduzca el numero de cambios de base que desea
          realizar: ");
  scanf ("%d%*c", &n);

  for (i=0; i<n; i++)
  {
    printf("Cambio de base %d - ", i);
    printf("Introduzca el valor en base diez y la nueva
            base: ");
```

```
    scanf("%d%*c%d", &val, &base);
    while (val<=0 || base<2 || base>9)
    {
            printf("Error en los datos introducidos\n");
       printf("Introduzca de nuevo el valor en base diez
              (>0) y la nueva base (2-9): ");
       scanf("%d%*c%d", &val, &base);
     }
     ndig = cambio_base(val, base, &res);
     printf ("\n%d en base 10 = %d en base %d
             (%d dígitos)\n\n", val, res, base, ndig);
  }
}

/* Coloque aquí la definición de la función cambio_base */
```

4.

a.

```
int leer_fecha(tfecha *f)
{
  int ok=1, bisiesto;
  int dias_mes[12]={31, 29, 31, 30, 31, 30, 31, 31, 30,
                    31, 30, 31};

  scanf("%d%*c%d%*c%d%*c", &f->dia, &f->mes, &f->anyo);

  if  (f->mes<1 || f->mes>12)
    ok = 0;
  else if ( f->dia<1 || f->dia>dias_mes[f->mes-1])
    ok = 0;
  else
  {
    bisiesto=es_bisiesto(f->anyo);
    if (f->mes==2 && bisiesto==0 &&
        f->dia==dias_mes[f->mes-1])
      ok = 0;
   }

   return(ok);
}
```

b.

```
void  sumar_dias_fecha(tfecha *f, int nd)
{
  int bisiesto, extra;
  int dias_mes[12]={31, 29, 31, 30, 31, 30, 31, 31, 30,
                    31, 30, 31};

  bisiesto = es_bisiesto(f->anyo);
  f->dia = f->dia+nd;
  while (f->dia > dias_mes[f->mes-1] ||
        (f->mes==2 && bisiesto==0 &&
         f->dia==dias_mes[f->mes-1]))
  {
```

```
        f->dia=f->dia - dias_mes[f->mes-1];
        if (f->mes==2 && bisiesto==0)
          f->dia = f->dia+1;
        f->mes = f->mes+1;
        if (f->mes > 12)
        {
          f->mes = 1;
          f->anyo = f->anyo+1;
          bisiesto=es_bisiesto(f->anyo);
        }
      }
    }
```

c.

```
#include <stdio.h>
#define MAXFECHAS 10

typedef struct
{
  int dia, mes, anyo;
} tfecha;

int es_bisiesto (int anyo);
int leer_fecha(tfecha *f);
void  mostrar_fecha(tfecha f);
void  sumar_dias_fecha(tfecha *f, int nd);

main()
{
  tfecha vfechas[MAXFECHAS];
  int i, numdias, ok;

  printf("Introduzca %d fechas validas:\n", MAXFECHAS);
  i = 0;
  while (i<MAXFECHAS)
  {
    printf("Fecha %d: ", i);
    ok = leer_fecha(&vfechas[i]);
    if (ok == 1)
      i = i+1;
    else
      printf("Fecha no valida. Vuelva a introducirla.\n");
  }

  printf("Introduzca el numero de dias que quiere sumar a
          las fechas: ");
  scanf("%d%*c", &numdias);

  printf("Fechas:\n");
  for (i=0; i<MAXFECHAS; i++)
  {
     mostrar_fecha(vfechas[i]);
     sumar_dias_fecha(&vfechas[i], numdias);
     printf(" + %d dias = ", numdias);
     mostrar_fecha(vfechas[i]);
```

```
      printf("\n");
    }
}

/* Coloque aquí la definición de las funciones del enunciado
y de los apartados anteriores */
```

5.

a.

```
void mostrar_vector (tvector v)
{
  int i;

  printf("\nVECTOR: ");
  if (v.nelem==0)
    printf("El vector no tiene elementos\n");
  else
  {
    for (i=0; i<v.nelem-1; i++)
      printf("%d,", v.vector[i]);
    printf("%d\n", v.vector[i]);
  }
}
```

b.

```
int insertar_en_vector_ordenado (tvector *v, int elem)
{
  int i, pos, inser;

  if (v->nelem == MAX)
    inser = 0;
  else
  {
    /* Como el vector esta ordenado, se busca la posición
       a insertar para mantener el orden */
    i = 0;
    while (i<v->nelem && v->vector[i]<elem)
      i = i+1;

    pos = i;  /* Posición donde hay que insertar el
                 elemento */

    /* Se desplazan los elementos una posición
       a la derecha */
    for (i=v->nelem-1; i>=pos; i--)
      v->vector[i+1] = v->vector[i];

    /* Se inserta el elemento */
    v->vector[pos] = elem;
    v->nelem = v->nelem+1;
    inser = 1;
  }
  return (inser);
}
```

c.

```
void leer_vector (tvector *v)
{
  int i, n, pos, ok=1, el;

  v->nelem = 0;
       printf("Introduzca el numero de elementos en el vector
         (MAX. %d): ", MAX);
  scanf("%d%*c", &n);
  while (n<1 || n>MAX)
  {
    printf("Error en el valor introducido.\n");
    printf("Introduzca el numero de elementos en el vector
            (MAX. %d): ", MAX);
    scanf("%d%*c", &n);
  }

  printf("Introduzca los elementos del vector: ");
  /* Leemos e insertamos ordenadamente */
  for (i=0; i<n; i++)
  {
    scanf("%d%*c", &el);
    insertar_en_vector_ordenado (v, el);
  }
}
```

d.

```
int eliminar_elementos_de_vector (tvector *v, int elem)
{
  int i, j, elim=0;

  i = 0;
  while (i<v->nelem)
  {
    if (v->vector[i]==elem)
    {
      /* Se elimina elemento elem del vector */
      for (j=i; j<v->nelem-1; j++)
        v->vector[j] = v->vector[j+1];

      v->nelem = v->nelem-1;
      elim = elim+1;
    }
    else
      i = i+1;
  }
  return (elim);
}
```

e.

```
#include <stdio.h>
#define MAX 10
```

```
typedef struct
{
  int nelem;         /* Número de elementos en el vector */
  int vector[MAX];   /* Vector de enteros */
} tvector;

void leer_vector(tvector *v);
void mostrar_vector(tvector v);
int insertar_en_vector_ordenado(tvector *v, int elem);
int eliminar_elementos_de_vector(tvector *v, int elem);

main()
{
  int opcion=0, el, ok, nelim;
  char opc;
  tvector v;

  v.nelem = 0;
  while (opcion!=5)
  {
    printf("\n1- Leer vector\n2- Mostrar vector ordenado\n
            3- Insertar elemento\n4- Eliminar elemento\n
            5- Salir\n\nElija una opcion: ");
    scanf("%d%*c", &opcion);

    if (opcion==1)
      leer_vector(&v);
    else if (opcion==2)
      mostrar_vector(v);
    else if (opcion==3)
    {
      printf("Introduzca el elemento a insertar: ");
      scanf("%d%*c", &el);
      ok = insertar_en_vector_ordenado(&v, el);
      if (ok == 0)
        printf("\nEl vector esta lleno. No se ha insertado
                el elemento.\n");
      else
        printf("\nElemento insertado.\n");
    }
    else if (opcion==4)
    {
      printf("Introduzca el elemento a eliminar: ");
      scanf("%d%*c", &el);
      nelim = eliminar_elementos_de_vector(&v, el);
      printf("\nSe han eliminado %d elementos.\n", nelim);
    }
    else if (opcion==5)
    {
      printf("\nHa elegido la opcion Salir del
              programa. Esta seguro (s/n): ");
      scanf("%c%*c", &opc);
      while (opc!='s' && opc!='n' && opc!='S' && opc!='N')
      {
```

```
        printf("\nOpcion incorrecta.Esta seguro (s/n): ");
        scanf("%c%*c", &opc);
      }
      if (opc=='n' || opc=='N')
        opcion = 0;
    }
    else
      printf("\nOpcion incorrecta. Elija de nuevo.\n\n");
 }
    }

/* Coloque aquí la definición de las funciones de los
   apartados anteriores */
```

6.

```
#include <stdio.h>
#define MAXFR 300
#define MAXPAL 20

typedef char tfrase[MAXFR];

void leer_frase(tfrase f);
int extraer_palabra(tfrase f, char p[MAXPAL], int *i);
int palindroma(char p[MAXPAL], int lon);
void mostrar_palabra(char p[MAXPAL], int lon);

main()
{
  tfrase f;
  char pal[MAXPAL];
  int i, j, k, lonp, cont_pal;

       /* Lee una frase acabada en '.' */
  leer_frase(f);

  printf("\nPalabras palindromas:\n");
  cont_pal = 0;
  i = 0;
  while (f[i]!='.')
       {
    /* Extrae la palabra */
    lonp = extraer_palabra(f, pal, &i);

    if (palindroma(pal, lonp)==1)   /* Es palíndroma */
    {
      mostrar_palabra(pal, lonp);
      cont_pal = cont_pal+1;
    }
  }
  printf("En total, hay %d palabras palindromas\n\n",
          cont_pal);
}
```

```
void leer_frase(tfrase f)
{
  int i;

  printf("Introduzca una frase acabada en punto: ");
  i = 0;
  scanf("%c", &f[i]);

  while (i<MAXFR-1 && f[i]!='.')
  {
    i = i+1;
    scanf("%c", &f[i]);
  }

  if (i==MAXFR-1)
    f[i] = '.';
}


int extraer_palabra(tfrase f, char p[MAXPAL], int *i)
{
  int j, lon;

  j = 0;
  while ((f[*i]>='a' && f[*i]<='z') ||
         (f[*i]>='A' && f[*i]<='Z'))
  {
    p[j] = f[*i];
    j = j +1;
    *i = *i+1;
   }
   lon = j;  /* Longitud de la palabra extraída */

   /* Se saltan espacios y signos de puntuación y se
      deja el índice al inicio de la palabra siguiente */
   while (f[*i]!='.' && (f[*i]<'a' || f[*i]>'z') &&
         (f[*i]<'A' || f[*i]>'Z'))
     *i = *i+1;

   return(lon);
}

int palindroma(char p[MAXPAL], int lon)
{
  int j, k, espalin=0;

  j = 0;
  k = lon-1;

  while (j<k && (p[j]==p[k] || (p[j]>='a' && p[j]<='z' &&
              p[j]==p[k]+('a'-'A'))    ||    (p[j]>='A'    &&
p[j]<='Z'
              && p[j]==p[k]-('a'-'A'))))
  {
    j = j+1;
    k = k-1;
```

```
  }

  if (j>=k)   /* Es palíndroma */
    espalin = 1;

  return(espalin);
}

void mostrar_palabra(char p[MAXPAL], int lon)
{
  int j;

  for (j=0; j<lon; j++)
    printf("%c", p[j]);

  printf("\n");
}
```

9.9 Anexos

Anexo 9.A. Precedencia de los operadores

Nivel	Operador	Símbolo	Asociatividad
1	Paréntesis Índice de vector Acceso a campo de estructura Acceso a campo de puntero de estructura	() [] . ->	De izquierda a derecha
2	Negación lógica Signo negativo Posincremento Posdecremento Indirección Referencia (también llamado: operador dirección de) Conversión de tipo	! - ++ -- * & (tipo)	De derecha a izquierda
3	Multiplicación División Módulo	* / %	De izquierda a derecha
4	Suma Resta	+ -	De izquierda a derecha
5	Menor que Menor que o igual a Mayor que Mayor que o igual a	< <= > >=	De izquierda a derecha
6	Igual a Distinto que	== !=	De izquierda a derecha
7	Conjunción	&&	De izquierda a derecha
8	Disyunción	||	De izquierda a derecha
9	Asignación	=	De derecha a izquierda

En esta tabla, la prioridad de los operadores va disminuyendo al ir descendiendo en la tabla. De esta forma, cuanto menor es el nivel de un operador, mayor es su prioridad. Los operadores de un mismo nivel tienen el mismo orden de prioridad entre ellos y, por tanto han de resolverse considerando el orden de asociatividad que tenga el nivel del operador.

Anexo 9.B. Normas de estilo

Este anexo resume las normas de estilo que se han seguido en este libro para escribir los programas.

1. Las llaves (`{}`) se abren y cierran en una línea exclusiva para ellas, es decir, no se escribe el código en la misma línea de la llave.
    a. La llave que abre se coloca en la misma columna, justo debajo de la primera letra que aparece en la línea anterior. Por ejemplo, si en la línea anterior aparece la palabra `main`, la llave que abre irá en la misma columna que ocupa la letra `m`. Si en la línea anterior hay una sentencia condicional `if`, la llave que abre irá en la misma columna que la que ocupa la letra `i`. Si en la línea anterior hay una iterativa `for` o `while`, la llave que abre irá en la misma columna que la que ocupan las letras `f` o `w`, según corresponda.
    b. La llave que cierra irá en la misma columna que ocupa la correspondiente llave que abre.
2. Todas las líneas de código (declaración de variables y sentencias), tanto del `main` como de las funciones de usuario, llevan una indentación de entre `2` y `4` espacios. En este libro, se utiliza una indentación a `2` espacios.
3. El nombre de las variables se escribe en letras minúsculas y el nombre de las constantes, en letras mayúsculas. Además, los identificadores utilizados para cada caso han de ser nemotécnicos.
4. Todas las sentencias que forman parte del cuerpo de las sentencias condicionales e iterativas están indentadas entre `2` y `4` espacios. En este libro, se utiliza una indentación a `2` espacios.
5. En los mensajes de textos de la función `printf`, solamente se utilizan los caracteres de la tabla ASCII del anexo 3.A.
6. Al definir los tipos de datos estructurados:
    a. Los campos del tipo estructurado están indentados entre `2` y `4` espacios. En este libro, se utiliza una indentación a `2` espacios.
    b. El `nombre_tipo_estructura` es un identificador que comienza con el carácter `t` o `T`.
7. Siempre que se implementan programas utilizando funciones de usuario, los prototipos de las funciones se colocan al inicio del fichero y las funciones se definen después del `main`.