

FUNDAMENTOS DE PROGRAMACIÓN UTILIZANDO EL LENGUAJE C

José Daniel Muñoz Frías
Rafael Palacios Hielscher

Colección

24

INGENIERÍA



FUNDAMENTOS DE PROGRAMACIÓN UTILIZANDO EL LENGUAJE C

PUBLICACIONES
DE LA UNIVERSIDAD PONTIFICIA COMILLAS
MADRID

COLECCIÓN INGENIERÍA, 24

PEDIDOS:

UPCO DEPARTAMENTO DE PUBLICACIONES
Universidad Pontificia Comillas, 5
28049 MADRID. Teléf. 91 540 61 45

EDISOFER, S. L.
San Vicente Ferrer, 71
28015 MADRID
Teléf. 91 521 09 24
Fax 91 532 28 63

JOSÉ DANIEL MUÑOZ FRÍAS
RAFAEL PALACIOS HIELSCHER

FUNDAMENTOS DE PROGRAMACIÓN UTILIZANDO EL LENGUAJE C



2006

Los autores de este libro han donado sus "derechos de autor" a la Asociación Juan Carlos Lavallo, <http://www.iit.upcomillas.es/ajcl>

ASOCIACIÓN JUAN CARLOS LAVALLE

La asociación Juan Carlos Lavallo es una asociación privada, sin ánimo de lucro y pluralista, dedicada al fomento de las relaciones con el Tercer Mundo en materia de educación y ayuda social.

Fue fundada en recuerdo de Juan Carlos Lavallo, Ingeniero ecuatoriano que realizó sus estudios superiores en Estados Unidos y decidió estudiar su doctorado en España con el objetivo de volver a Ecuador y ayudar al desarrollo de su país. Murió en un accidente en 1990 durante una excursión campestre y un grupo de amigos constituyó una Asociación en su nombre para promover su recuerdo y para contribuir a sus ideales para la consecución de un mundo más justo.

A lo largo de los 15 años de existencia, la Asociación ha cobrado vida propia y la mayoría de sus miembros no conocieron a Juan Carlos pero comparten sus ideales. Se ha colaborado en diversos proyectos educativos y se han financiado becas de estudio en 8 países de América y África.

Una pequeña cantidad del precio de este libro la recibe la Asociación Juan Carlos Lavallo y la dedica íntegramente al desarrollo de estos proyectos educativos y a la financiación de los estudios de niños de varios países en desarrollo.

Fe de erratas: ver la página web del libro: <http://www.iit.upcomillas.es/libroc>

© Universidad Pontificia Comillas
Universidad Comillas, 3
28049 MADRID
© José Daniel Muñoz Frías
© Rafael Palacios Hielscher

Diseño de cubierta: Belén Recio Godoy

EDICIÓN DIGITAL
ISBN: 978-84-8468-333-9

Reservados todos los derechos. Queda totalmente prohibida la reproducción total o parcial de este libro por cualquier procedimiento electrónico o mecánico, incluyendo fotocopia, grabación magnética o cualquier sistema de almacenamiento o recuperación de información, sin permiso escrito de la Universidad Pontificia Comillas.

Índice general

Índice general	VII
Prólogo	XI
1 Descripción del ordenador	1
1.1. Introducción	1
1.2. Arquitectura de un ordenador de propósito general	1
1.3. Codificación de la información	3
1.4. Codificación del programa	8
1.5. Resumen	9
2 El primer programa en C	11
2.1. Introducción	11
2.2. El sistema operativo	11
2.3. Creación de un programa en C	11
2.4. Nuestro primer programa en C	13
2.5. Resumen	17
2.6. Ejercicios	18
3 Tipos de datos	19
3.1. Introducción	19
3.2. Variables	19
3.3. Tipos básicos de variables	19
3.4. Más tipos de variables	21
3.5. Constantes	22
3.6. Tamaño de las variables	23
3.7. Escritura de variables en pantalla	23
3.8. Lectura de variables por teclado	27
3.9. Recomendaciones y advertencias	28
3.10. Resumen	28
3.11. Ejercicios	29
4 Operadores Aritméticos	31
4.1. Introducción	31
4.2. El operador de asignación	31
4.3. Operadores para números reales	31
4.4. Operadores para números enteros	32
4.5. Operador de cambio de signo	33
4.6. De vuelta con el operador de asignación	34
4.7. Conversiones de tipos	34

4.8.	Otras operaciones matemáticas	37
4.9.	Recomendaciones y advertencias	37
4.10.	Resumen	38
4.11.	Ejercicios	38
5	Control de flujo: Bucles	39
5.1.	Introducción	39
5.2.	Sintaxis del bucle for	40
5.3.	Operadores relacionales	41
5.4.	Doble bucle for	42
5.5.	Bucle while	44
5.6.	Bucle do-while.	48
5.7.	¿Es el bucle for igual que el while?	49
5.8.	El índice del bucle for sirve para algo	51
5.9.	Recomendaciones y advertencias	52
5.10.	Resumen	52
5.11.	Ejercicios	53
6	Control de flujo: if y switch-case	55
6.1.	Introducción	55
6.2.	El bloque if. Introducción	55
6.3.	Sintaxis del bloque if	55
6.4.	Formato de las condiciones	57
6.5.	Valores de verdadero y falso	59
6.6.	Bloque if else-if	59
6.7.	La construcción switch-case. Introducción	62
6.8.	La sentencia break	68
6.9.	La sentencia continue	70
6.10.	Recomendaciones y advertencias	71
6.11.	Resumen	71
6.12.	Ejercicios	72
7	Funciones	75
7.1.	Introducción	75
7.2.	Estructura de una función	76
7.3.	Prototipo de una función	77
7.4.	Ejemplo	78
7.5.	Paso de argumentos a las funciones. Variables locales	80
7.6.	Salida de una función y retorno de valores	82
7.7.	Funciones sin argumentos	83
7.8.	La pila	83
7.9.	Funciones recursivas	85
7.10.	Ejemplos	87
7.11.	Recomendaciones y advertencias	90
7.12.	Resumen	91
7.13.	Ejercicios	91

8 Vectores y Matrices	93
8.1. Introducción	93
8.2. Vectores	93
8.3. Cadenas de caracteres	95
8.4. Matrices	103
8.5. Utilización de #define con vectores y matrices	104
8.6. Paso de vectores, cadenas y matrices a funciones	107
8.7. Recomendaciones y advertencias	110
8.8. Resumen	111
8.9. Ejercicios	111
9 Punteros	115
9.1. Introducción	115
9.2. Declaración e inicialización de punteros	115
9.3. Operaciones con punteros	118
9.4. Punteros y funciones	120
9.5. Punteros y vectores	122
9.6. Asignación dinámica de memoria	125
9.7. Recomendaciones y advertencias	130
9.8. Resumen	131
9.9. Ejercicios	131
10 Estructuras de datos	135
10.1. Introducción	135
10.2. Declaración y definición de estructuras	135
10.3. La sentencia typedef	136
10.4. Acceso a los miembros de una estructura	137
10.5. Ejemplo: Suma de números complejos	138
10.6. Estructuras y funciones	140
10.7. Punteros a estructuras	141
10.8. Vectores de estructuras	144
10.9. Recomendaciones y advertencias	147
10.10. Resumen	147
10.11. Ejercicios	148
11 Archivos	149
11.1. Introducción	149
11.2. Apertura de archivos. La función fopen	150
11.3. Cierre de archivos. La función fclose.	152
11.4. Lectura y escritura en archivos de texto. Las funciones fprintf y fscanf.	153
11.5. Otras funciones de entrada y salida para archivos de texto	159
11.6. Almacenamiento de estructuras. Archivos binarios	162
11.7. Recomendaciones y advertencias	168
11.8. Resumen	169
11.9. Ejercicios	169

A	Consejos de Estilo de Programación en C	173
A.1.	Estructura del programa	173
A.2.	Variables	173
A.3.	Funciones	173
A.4.	Claridad del Código	174
A.5.	Documentación	175
A.6.	Ejemplo	175
B	Consejos para Programación Segura y Estable en C	177
B.1.	Controlar Buffer Overflow	177
B.2.	Medidas para evitar bucles infinitos	177
B.3.	Desconfiar del valor de las variables	177
B.4.	Estabilidad del programa	177
C	Contenidos disponibles en la web del libro	179
	Bibliografía	181
	Índice alfabético	182

Prólogo

Este libro es el fruto de varios años de experiencia de los autores en la enseñanza de la programación. Debido a esto, la finalidad principal del libro no es enseñar el lenguaje C, para lo cual existen innumerables obras en el mercado, sino enseñar a programar usando el lenguaje C. La elección de este lenguaje ha sido simplemente porque es muy popular hoy en día, especialmente en el campo de los microprocesadores y microcontroladores, y porque es la base de lenguajes más avanzados como C++ y Java. Algunos profesores argumentan que el lenguaje C no es el más apropiado para enseñar a programar, siendo Pascal o Modula-2 lenguajes más orientados a la docencia. No obstante, debido a la falta de popularidad de estos lenguajes en la actualidad, no parece la elección más sensata. Además, tal como se demostrará a lo largo del libro, el lenguaje C, bien usado, es una buena elección para mostrar los rudimentos de la programación.

Como se acaba de decir, este libro no pretende mostrar todos los detalles del lenguaje C. Para ello puede consultar cualquier referencia, como por ejemplo el clásico [Kernighan and Ritchie, 1991]. De hecho, algunos aspectos del lenguaje ni siquiera se mencionan porque, bajo nuestro humilde punto de vista, lo único que hacen es contribuir a que el programador escriba programas ilegibles sin aportar nada a cambio. Se ha hecho un especial énfasis en el uso de la programación estructurada y en favorecer la legibilidad del código frente a los “trucos” para ahorrar la escritura de unos cuantos caracteres. Esto último es algo que tenía sentido en los años en los que se desarrolló el lenguaje, con ordenadores con 64 kB de memoria; pero que desde luego no tiene ningún sentido hoy en día.

La estructura del libro es la siguiente: en el primer capítulo se presenta una introducción a la arquitectura interna del ordenador y a la forma en la que se codifican sus datos. Aunque parezca que esto no viene a cuento, es crucial conocer cómo son los entresijos del ordenador para comprender los principios de diseño del lenguaje y también para poder escribir programas eficientes. En el segundo capítulo se escribe un primer programa sencillo para ilustrar el lenguaje y el proceso que hay que llevar a cabo para escribir el programa y ejecutarlo en el ordenador. El tercer capítulo muestra cómo crear variables en un programa y cómo su uso es distinto en función del tipo de dato que vayan a contener (números, letras, etc.). En el siguiente capítulo se muestra como realizar operaciones matemáticas con las variables, lo que nos permitirá usar el ordenador como una “super-calculadora”. En el quinto capítulo se verá como repetir la ejecución de un conjunto de instrucciones, lo que en programación se denomina un bucle, lo cual permite al ordenador realizar tareas repetitivas fácilmente. El siguiente capítulo introduce otro concepto fundamental: la toma de decisiones. Esto permite que el programa realice una serie de instrucciones u otras en función de un resultado lógico, lo cual hace que el ordenador parezca “inteligente”. En el capítulo siete se presenta otro de los conceptos fundamentales de programación, la división del programa en partes mediante funciones. Esto permitirá diseñar programas extensos sin morir

en el intento. En el capítulo ocho se estudia el manejo de vectores y matrices en C, los cuales son imprescindibles en los programas de cálculo numérico. Además se estudia también en este capítulo el manejo de cadenas de caracteres. A continuación se estudia el concepto de puntero y sus principales aplicaciones en el lenguaje, como son el manejo de vectores de tamaño variable y el permitir que las funciones puedan “devolver” más de un valor. Para finalizar, los dos últimos capítulos estudian conceptos más avanzados: las estructuras de datos y los archivos.

Agradecimientos

Es claro que un libro nunca es exclusivamente fruto del trabajo de sus autores. Hay numerosas personas que de una manera u otra aportan su granito de arena en una labor tan ingente como es la escritura de un libro. Es por tanto de justicia mostrar nuestro agradecimiento en estas páginas a todas esas personas. En primer lugar tenemos que agradecer los comentarios de muchos de nuestros alumnos que nos han ayudado a convertir unos apuntes en el libro que tiene ahora en sus manos. También ha sido muy valiosa la ayuda de Eduardo Alcalde, que ha tenido la paciencia de leer el libro desde el principio hasta el final corrigiendo numerosas erratas y aportando valiosos comentarios. Tampoco podemos olvidarnos de Santiago Canales que aportó algunas ideas para el apartado sobre recursividad, de Yolanda González que siempre nos animó a escribir este libro y que también aportó numerosos comentarios al mismo; ni tampoco de Romano Giannetti, que siempre ha sabido sacarnos de nuestros atascos con \LaTeX . También tenemos que agradecer la labor de Belén Recio en el diseño e impresión del libro. Por último, aunque no por ello menos importante, nuestras familias siempre han sabido tener paciencia cuando hemos tenido que dedicarle más tiempo de la cuenta al libro.

Convenciones usadas en el texto

Para facilitar la lectura del texto, se han usado una serie de convenciones tipográficas, las cuales se enumeran a continuación:

<i>inglés</i>	Las palabras que se expresan en su idioma original se han escrito en <i>cursiva</i> .
código	Los ejemplos de código en C se muestran con un tipo de letra monoespaciada.
int	Las palabras reservadas del lenguaje se escriben con un tipo de letra monoespaciada y negrita .
<i>/* Ojo */</i>	Los comentarios del código se escriben con un tipo de letra <i>monoespaciada y cursiva</i> .

Tenga en cuenta que estas convenciones tipográficas son sólo para facilitar la lectura. Cuando escriba un programa no intente cambiar el tipo de letra. No obstante, la mayoría de los compiladores actuales disponen de editores de código que colorean automáticamente las palabras clave y los comentarios.

Además se han usado una serie de notas al margen para mostrar material adicional. El tipo de información mostrado en la nota al margen se indica mediante un icono:



Información sobre C99



Documentos adicionales disponibles en la página web del libro. La dirección de la página es: www.iit.upcomillas.es/libroc



Ejercicios propuestos.

CAPÍTULO 1

Descripción del ordenador

1.1. Introducción

Un ordenador es un equipo de procesamiento de datos. Su tarea consiste en, a partir de unos datos de entrada, generar otros datos de salida, tal como se muestra en la figura 1.1.a. La principal ventaja de un ordenador radica en que el procesamiento que realiza con los datos no es fijo, sino que responde a un programa previamente introducido en él, tal como se muestra en la figura 1.1.b. Esta propiedad dota a los ordenadores de una gran flexibilidad, permitiendo que una misma máquina sirva para fines tan dispares como diseñar circuitos electrónicos, resolver problemas matemáticos, navegar por Internet o, por qué no, jugar a los marcianitos. Nótese también que debido a esto un ordenador sin un programa que lo controle es un cacharro totalmente inútil.

El objetivo de este libro es enseñar al lector los fundamentos del proceso de programación de un ordenador usando el lenguaje C.

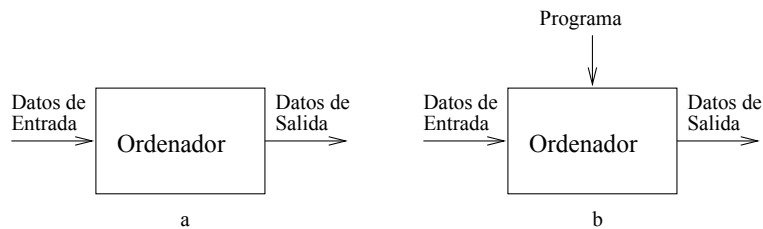


Figura 1.1: Proceso de información en un ordenador

1.2. Arquitectura de un ordenador de propósito general

En la figura 1.2 se muestra un diagrama de bloques de un ordenador de propósito general, en el que se pueden observar tres bloques principales: la **unidad central de proceso** o CPU,¹ la **memoria** y el **sistema de entrada/salida**. Todos estos elementos se comunican mediante un conjunto de conexiones eléctricas denominadas **buses**.

¹Del inglés *Central Processing Unit*.

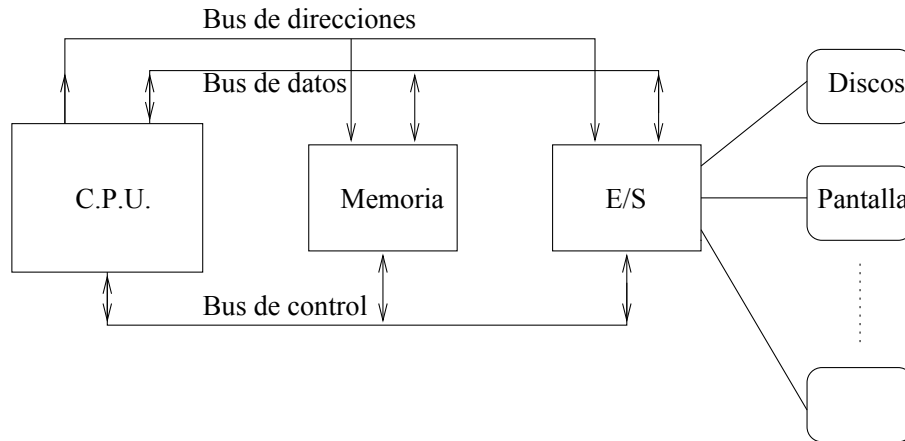


Figura 1.2: Diagrama de bloques de un ordenador

1.2.1. La unidad central de proceso

La unidad central de proceso, también llamada CPU o simplemente procesador, es la unidad responsable de realizar todo el procesamiento de información. Para ello lee un **programa** de la memoria y actúa según las instrucciones de dicho programa. Dichas instrucciones son muy simples: leer datos de la memoria, realizar operaciones matemáticas y lógicas simples (sumas, comparaciones, etc.), escribir resultados en la memoria, etc.

1.2.2. Memoria

Es la unidad encargada de almacenar tanto el programa que le dice a la CPU lo que tiene que hacer, como los datos con los que tiene que trabajar. Desde el punto de vista del programador consiste en un vector de **posiciones de memoria** en cada una de las cuales se puede almacenar un solo objeto (una instrucción o un dato), tal como se muestra en la figura 1.3. A cada una de estas posiciones de memoria se puede acceder (leer o escribir) sin más que especificar su **dirección**. Nótese que a la primera posición se le asigna la dirección 0, a la siguiente la 1, y así sucesivamente hasta llegar a la última posición de memoria.

...	...
3	15
2	254
1	37
0	9
0	27
Dirección	Memoria

Figura 1.3: Estructura conceptual de la memoria

1.2.3. Unidad de entrada/salida

Esta unidad se encarga de comunicar al ordenador con el mundo exterior y con los dispositivos de almacenamiento secundario (discos). En algunas arquitecturas aparece como una serie de posiciones de memoria más, indistinguibles por el programador de la memoria normal, y en otras está en un **espacio de direcciones** separado.

1.2.4. Buses

La interconexión entre los elementos del ordenador se realiza mediante un bus de datos, gracias al cual el procesador lee o escribe datos en el resto de dispositivos,² un bus de direcciones por el cual el procesador indica a los dispositivos qué posición quiere leer o escribir,³ un bus de control mediante el cual el procesador les indica el momento en el que va a realizar el acceso, si éste va a ser de lectura o escritura, etc. Este bus de control también permite a los dispositivos pedir la atención del procesador ante un suceso mediante un mecanismo llamado **interrupción**.

1.3. Codificación de la información

Todas las unidades estudiadas en la Sección 1.2 están formadas por circuitos electrónicos digitales que se comunican entre sí. Si ha estudiado algo de electrónica digital, sabrá que estos circuitos digitales trabajan con señales que sólo toman dos valores: encendido-apagado, cargado-descargado, tensión alta-tensión baja.⁴ A estos dos estados diferenciados se les asignan los valores binarios 0 y 1. Por tanto, dentro de un ordenador, todo discurre en forma de dígitos binarios o **bits**.⁵ Por el contrario en la vida real casi ningún problema está basado en números binarios, y la mayoría ni siquiera en números. Por tanto es necesario establecer una correspondencia entre las magnitudes binarias con las que trabaja el ordenador y las magnitudes que existen en el mundo real. A esta correspondencia se le denomina **codificación**.

Si las operaciones a realizar son funciones lógicas (AND, OR, NOT, etc.) la codificación es muy simple: al valor falso se le asigna el dígito binario 0 y a la condición cierto se le asigna el 1.

Si en cambio hemos de realizar operaciones matemáticas, la codificación de números es un poco más compleja. Antes de estudiar cómo se codifican los números en un ordenador, vamos a estudiar un poco de matemáticas (¡pero que no cunda el pánico!).

1.3.1. Sistemas de numeración posicionales

Al sistema de numeración que usamos día a día se le denomina posicional porque cada número está formado por una serie de dígitos de forma que cada dígito tiene un peso que depende de su posición. El valor del número se forma por la suma del producto de cada dígito por su peso, así por ejemplo cuando se escribe el número 1327, en realidad se está diciendo:

²Se dice entonces que este bus es **bidireccional** pues permite que la información viaje desde el procesador a los dispositivos o viceversa.

³Este bus en cambio es **unidireccional** pues la dirección viaja siempre del procesador a los dispositivos.

⁴Si no ha estudiado nada de electrónica digital tampoco se preocupe; lo único que necesita saber es lo que se acaba de comentar.

⁵La palabra bit viene del inglés **binary digit**.

$$1327 = 1 \times 1000 + 3 \times 100 + 2 \times 10 + 7 \times 1 = 1 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 7 \times 10^0 \quad (1.1)$$

Como se puede apreciar, los pesos por los que se multiplica cada dígito se forman elevando un número, al que se denomina **base** a la potencia indicada por la posición del dígito. Al dígito que está más a la derecha se le denomina **dígito menos significativo** y se le asigna la posición 0; al que está más a la izquierda se le denomina **dígito más significativo**.

Como en un sistema digital sólo existen dos dígitos binarios, 0 y 1, los números dentro de un ordenador han de representarse en base 2,⁶ así por ejemplo:⁷

$$\begin{aligned} 11011_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = \\ &= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 27 \end{aligned} \quad (1.2)$$

Esta base es muy adecuada para las máquinas, pero bastante desagradable para los humanos, por ejemplo el número 1327 en binario es 10100101111_2 ; bastante más difícil de recordar, escribir e interpretar. Por ello los humanos, que para eso somos más listos que las máquinas, usamos sistemas de numeración más aptos a nuestras facultades. La base 10 es la más cómoda para operar con ella debido a que las potencias de la base son fáciles de calcular y a que la suma expresada en (1.1) se calcula sin ninguna dificultad. Por el contrario en base 2 ni las potencias de la base ni la suma final (1.2) son tan fáciles de calcular.

Uno de los inconvenientes de la representación de números en binario es su excesiva longitud. Para simplificar las cosas se pueden convertir los números binarios a decimales y trabajar con ellos, pero eso no es tarea fácil, pues hay que realizar una operación como la indicada en (1.2). Para solucionar este problema, se pueden usar bases que son también potencias de 2, de forma que la notación sea compacta y fácil de manejar por los humanos y que además las conversiones a binario sean fáciles de realizar. Las dos bases más usadas en el mundo de la computación son la octal (base 8) y la hexadecimal (base 16). En el cuadro 1.1 se muestran las equivalencias entre binario, decimal, octal y hexadecimal.

Como se puede apreciar, los números octales sólo usan 8 símbolos (del 0 al 7) y la base hexadecimal precisa 16 (del 0 al 9 y se añaden las letras de la A a la F).

La conversión entre binario y octal se realiza agrupando los bits en grupos de tres y realizando la transformación de cada grupo por separado, según el cuadro 1.1:

$$10100101111_2 = 10|100|101|111 = 2457_8$$

y entre binario y hexadecimal se hace igual, pero agrupando los bits de cuatro en cuatro:

$$10100101111_2 = 101|0010|1111 = 52F_{16}$$

La notación octal se usó en los ordenadores primitivos, en los que el ordenador comunicaba los resultados con lamparitas (como salen en las películas antiguas) que se agrupaban en grupos de 3 para su interpretación en base octal. Sin embargo, con

⁶Para representar números en una base n hacen falta n símbolos distintos.

⁷Para indicar números representados en una base distinta a la base 10, se pondrá en este libro la base como subíndice.

Binario	Decimal	Octal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F
10000	16	20	10
10001	17	21	11

Cuadro 1.1: Conversión entre bases

el avance de la técnica, se vio que era conveniente usar números binarios de 8 bits, a los que se les denominó **bytes**. Estos bytes se representan mucho mejor como dos dígitos hexadecimales, uno para cada grupo de 4 bits.⁸ El caso es que hoy en día la notación octal está francamente en desuso, siendo lo más corriente expresar un número binario mediante su equivalente hexadecimal cuando ha de ser interpretado por un ser humano (dentro del ordenador por supuesto todo esta en binario).

1.3.2. Codificación de números en un ordenador

Una de las aplicaciones principales de los ordenadores es la realización de cálculos matemáticos, cálculos que pueden ser tan simples como restar los 20 Euros de una cuenta bancaria cuando se sacan de un cajero o tan complejos como una simulación de la combustión en el cilindro de un motor de explosión. Sin embargo se ha visto que los ordenadores sólo saben usar el 0 y el 1. Por tanto es necesario buscar métodos para codificar los números en un ordenador. La tarea no es tan fácil como pueda parecer a primera vista. En primer lugar ¿cuántos tipos de números existen? Pues tenemos en primer lugar los números naturales, que son los más simples. Si usamos restas nos hacen falta los números negativos. Si queremos dividir, salvo que tengamos mucha suerte y la división sea exacta, debemos de usar los números racionales. Si además tenemos que hacer raíces cuadradas o usar números tan apasionantes como π , nos las tendremos que ver con los irracionales. Incluso si se nos ocurre hacer una raíz cuadrada de un número negativo tenemos que introducirnos en las maldades de los números imaginarios.

Los lenguajes de programación de propósito general como el C o el Pascal, permiten trabajar tanto con números enteros como con números reales. Otros lenguajes como el

⁸Además a cada grupo de 4 bits, representable directamente por un dígito hexadecimal, se le denomina **nibble**.

nº de bits	8	16	32
Naturales	$0 \leftrightarrow 255$	$0 \leftrightarrow 65535$	$0 \leftrightarrow 4294967296$
Enteros	$-128 \leftrightarrow 127$	$-32768 \leftrightarrow 32767$	$-2147483648 \leftrightarrow 2147483647$

Cuadro 1.2: Rangos de los números naturales y enteros.

FORTRAN, más orientados al cálculo científico, también pueden trabajar con números complejos.

Codificación de números enteros

Si se desea codificar un número natural, la manera más directa es usar su representación en binario, tal como se ha visto en la Sección 1.3.1. No obstante, los procesadores trabajan con magnitudes de un número de bits predeterminados, típicamente múltiplos pares de un byte. Así por ejemplo los procesadores Intel Pentium pueden trabajar con números de 1, 2 y 4 bytes, o lo que es lo mismo, de 8, 16 y 32 bits. Por tanto, antes de introducir un número natural en el ordenador, hemos de averiguar el número de bits que hacen falta para codificarlo, y luego decirle al ordenador que use un **tipo** de almacenamiento adecuado.

Para averiguar cuantos bits hacen falta para almacenar un número natural no hace falta convertirlo a binario. Si nos fijamos en que con n bits se pueden representar números naturales que van desde 0 a $2^n - 1$, sólo hace falta comparar el número a codificar con el rango del tipo en el que lo queremos codificar. Así por ejemplo si queremos almacenar el número 327 en 8 bits, como el rango de este tipo es de 0 a $2^8 - 1 = 255$, vemos que en 8 bits no cabe, por tanto hemos de almacenarlo en 16 bits, que al tener un rango de 0 a $2^{16} - 1 = 65535$, nos vale perfectamente.

Para la codificación de números negativos existen varios métodos, aunque el más usado es la codificación en complemento a dos, que posee la ventaja de que la operación suma es la misma que para los números naturales, lo que simplifica la circuitería del ordenador. En esta codificación el bit más significativo vale cero para los números positivos y uno para los negativos. En el resto de bits está codificada la magnitud, aunque el método de codificación se escapa de los fines de esta introducción. Lo que si ha de quedar claro es que como ahora tenemos en n bits números positivos y negativos, el rango total de 0 a $2^n - 1$ se divide ahora en -2^{n-1} a $2^{n-1} - 1$. Así pues si por ejemplo queremos codificar el número -227 necesitaremos 16 bits, pues el rango de un número codificado en complemento a dos de 8 bits es de -2^7 a $2^7 - 1$, es decir, de -128 a 127. En cambio el rango disponible en 16 bits es de -32768 a 32767. En el cuadro 1.2 se muestran los rangos disponibles para números con y sin signo codificados con 8, 16 y 32 bits.

Codificación de números reales

Para la mayoría de las aplicaciones de cálculo científico se necesita trabajar con números reales. Estos números reales se representan según una **mantisa** de parte entera 0 y un **exponente** (por ejemplo 3,72 se almacena como $0,372 \times 10^1$). Nuevamente tanto la mantisa como el exponente se almacenan en binario, según una codificación que escapa de los fines de esta breve introducción. Antiguamente cada ordenador usaba un tipo de codificación en punto flotante distinta. Por ello surgió la necesidad de buscar un estándar de almacenamiento común para todas las plataformas. De

ahí surgió el estándar IEEE 854⁹ que es el usado actualmente por la mayoría de los procesadores para la realización de las operaciones en coma flotante. Este estándar define tanto la codificación de los números como la manera de realizar las operaciones aritméticas con ellos (incluidos los redondeos).

1.3.3. Codificación de caracteres alfanuméricos

Aunque el proceso matemático es una parte importante dentro de un ordenador, la mayoría de los problemas en el mundo real requieren la posibilidad de trabajar con texto. Por tanto es necesario buscar un método para codificar los caracteres alfanuméricos dentro de un ordenador. La manera de realizar esta codificación es asignar a cada carácter un valor numérico, totalmente arbitrario, almacenando las correspondencias carácter-número en una tabla.

Al principio existieron varias codificaciones de caracteres, pues los fabricantes, como siempre, no se pusieron de acuerdo. Por ello nació el estándar ASCII que son las siglas de “American Standard Code for Information Interchange”. Como su propio nombre indica este estándar es americano, y como su idioma no contiene letras con tildes, usaron sólo números de 7 bits (0 a 127) para almacenar letras, números y algunos signos especiales (\$, -, %, etc.) pero no codificaron caracteres tan estupendos como la ñ o la á. Esto ha dado lugar a que, incluso hoy en día, los que tenemos la “desgracia” de llamarnos Muñoz, nos vemos con frecuencia rebautizados a Mu\oz. Para solucionar tan dramática situación, se extendieron los 7 bits originales a 8, usando los 128 nuevos caracteres para codificar las letras especiales de las lenguas no inglesas. El problema fue que nuevamente no se pusieron de acuerdo los fabricantes y cada uno lo extendía a su manera, con el consiguiente problema. Para aliviar esto, surgió el estándar ISO 8859,¹⁰ que en los 128 primeros caracteres coincide con el ASCII y en los restantes 128 contiene extensiones para la mayoría de los lenguajes. Como no caben todos los caracteres de todas las lenguas en sólo 128 posiciones, se han creado varios alfabetos ISO 8859, de los cuales el primero (ISO 8859-1 Latin-1) contiene los caracteres necesarios para codificar los caracteres de las lenguas usadas en Europa occidental. No obstante, con la llegada de nuestro querido Euro, ha sido necesario modificar este juego de caracteres, lo cual se ha aprovechado también para añadir unos cuantos caracteres que faltaban para soportar completamente el Francés y el Finlandés. La nueva tabla se denomina ISO 8859-15 (También denominado Latin-9). Para no tener que disponer de varios alfabetos como ocurre con el estándar ISO 8859, se ha creado el ISO 10646, también llamado **Unicode**, el cual usa 16 bits para codificar cada carácter, con lo que la tabla contiene 65536 posibles caracteres, que son más que suficientes para codificar todos los caracteres del mundo. Para facilitar la compatibilidad con los sistemas actuales, los primeros 256 caracteres de la tabla Unicode coinciden con el juego de caracteres ISO 8859-1.

En el cuadro 1.3 se muestra una tabla del juego de caracteres ISO 8859-1. En dicha tabla la posición de cada carácter indica el número con el que se codifica. Para obtener dicho número, la fila en la que está el carácter indica el dígito hexadecimal menos significativo, y la columna del carácter se corresponde con el dígito más significativo. Para facilitar las cosas la primera fila y la primera columna contienen el número de orden de las columnas y las filas de la tabla en hexadecimal. Así por ejemplo, para

⁹IEEE son las siglas del Institute of Electrical and Electronics Engineers, que es una asociación americana de ingenieros que se dedica, entre otras tareas, a la definición de estándares relacionados con la ingeniería eléctrica y electrónica.

¹⁰ISO son las siglas de International Standards Organization.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL \0	DLE	␣	0	@	P	‘	p	Ă	Ř	ă	ř	À	Đ	à	đ
1	SOH	DC1	!	1	A	Q	a	q	Ą	Ś	ą	ś	Á	Ñ	á	ñ
2	STX	DC2	"	2	B	R	b	r	Č	Š	č	š	Â	Ô	â	ô
3	ETX	DC3	#	3	C	S	c	s	Č	Š	č	š	Ã	Ó	ã	ó
4	EOT	DC4	\$	4	D	T	d	t	Ď	Ť	ď	ť	Ä	Ö	ä	ö
5	ENQ	NAK	%	5	E	U	e	u	Ě	Ť	ě	ť	Å	Õ	å	õ
6	ACK	SYN	&	6	F	V	f	v	Ě	Ť	ě	ť	Æ	Ö	æ	ö
7	BEL \a	ETB	'	7	G	W	g	w	Ĝ	Ŭ	ĝ	ŭ	Ç	Œ	ç	œ
8	BS \b	CAN	(8	H	X	h	x	Ĺ	Ÿ	ĺ	ÿ	È	Ø	è	ø
9	HT \t	EM)	9	I	Y	i	y	Ľ	Ž	ľ	ž	É	Ũ	é	ù
A	LF \n	SUB	*	:	J	Z	j	z	Ł	Ż	ł	ż	Ê	Ū	ê	ú
B	VT \v	ESC	+	;	K	[k	{	Ń	Ż	ń	ż	Ë	Ū	ë	û
C	FF \f	FS	,	<	L	\	l		Ń	IJ	ñ	ij	Ì	Ū	ì	ü
D	CR \r	GS	-	=	M		m	}	■	Í	■	í	Í	Ý	í	ý
E	SO	RS	.	>	N	^	n	~	Ō	đ	ó	đ	Î	Þ	î	þ
F	SI	US	/	?	O	_	o	DEL	Ř	š	ř	š	Ī	SS	ī	ß

Cuadro 1.3: Juego de caracteres ISO 8859-1.

saber el código asignado a la letra 'a', basta con mirar en la tabla la columna en la que está situada dicha letra (6) y la fila (1). Por tanto la letra 'a' se codifica con el número 61_{16} . La tabla sirve también para realizar el proceso contrario. Si por ejemplo queremos saber qué letra se corresponde con el número $4D_{16}$, basta con mirar qué letra está en la columna 4 fila D, que en este caso es la 'M'.

Los primeros caracteres de la tabla (del 00_{16} al $1F_{16}$) son caracteres de control y por tanto carecen de símbolo. Los caracteres que aparecen en la tabla son una abreviatura de su función. Por ejemplo, el carácter $0A_{16}$ (LF) es el carácter de retorno de carro (*Line Feed* en inglés). Algunos de estos caracteres se usan frecuentemente en los programas en C, por lo que el lenguaje ha previsto unas secuencias de caracteres para generarlos, denominadas **secuencias de escape**. Dichas secuencias aparecen al lado de las abreviaturas. Por ejemplo, para generar el carácter LF en C se puede escribir `\n`, tal como se verá en el siguiente tema.

Por último, conviene destacar que el orden numérico dentro de la tabla coincide con el orden alfabético de las letras, lo cual es muy útil cuando se desean ordenar caracteres por orden alfabético.

1.4. Codificación del programa

En la sección 1.1 se dijo que para que un ordenador realice cualquier tarea, es necesario decirle cómo debe realizarla. Esto se hace mediante un programa, el cual está compuesto por instrucciones muy simples. Estas instrucciones también están codificadas (en binario por supuesto), según el denominado **juego de instrucciones** del procesador. Cada procesador tiene su propio juego de instrucciones y su propia manera de codificarlas, que dependen de la arquitectura interna del procesador. Así por ejemplo en un microprocesador 68000 de Motorola la instrucción para sumar 7 a

una posición de memoria indicada por el registro interno¹¹ A2 es 0101111010010010.

En los primeros ordenadores los programas se codificaban directamente en binario, lo cual era muy penoso y propenso a errores. Por suerte los ordenadores de entonces no eran muy potentes y los programas tampoco eran demasiado sofisticados. Aun así, como al hombre nunca le han gustado las tareas repetitivas, pronto se vio la necesidad de automatizar un poco el proceso. Para ello a cada instrucción se le asigna un **mnemónico** fácil de recordar por el hombre y se deja a un programa la tarea de convertir estos mnemónicos a sus equivalentes en binario legibles por el procesador. El ejemplo de antes sobre sumar (add) el número 7 al registro A2 se escribiría ahora como ADDQ.B #7, (A2), lo cual es mucho más legible. A este lenguaje se le denomina lenguaje ensamblador, aunque a veces también se le denomina código máquina.

El problema de la programación en ensamblador es que este lenguaje es distinto para cada procesador, pues está muy unido a su arquitectura interna. Por ello si tenemos un programa escrito en ensamblador para un 68000 y decidimos ejecutarlo en un Pentium, tendremos que reescribirlo de nuevo. Para solucionar este problema (o al menos paliarlo bastante), se desarrollaron los lenguajes de alto nivel. Estos lenguajes parten de una arquitectura de procesador genérica, como la que se ha introducido en este capítulo, y definen un lenguaje independiente del procesador, por lo que una vez escrito el programa, éste se puede ejecutar prácticamente sin cambios en cualquier procesador.

1.4.1. *Compiladores e intérpretes*

En un lenguaje de alto nivel es necesario traducir el programa que introduce el usuario al código máquina del ordenador. Esta traducción se puede realizar al mismo tiempo de la ejecución, de forma que se traduce cada línea del programa de alto nivel y después se ejecuta. Como ejemplos de lenguajes que son normalmente interpretados tenemos el BASIC y el Java. El principal inconveniente de los lenguajes interpretados es que el proceso de traducción lleva tiempo, y hay que realizarlo una y otra vez.

Si se desea eliminar el tiempo de traducción del programa de alto nivel, podemos traducirlo todo de una vez y generar un programa en código máquina que será ejecutable directamente por el procesador. Con ello la ejecución del programa será mucho más rápida, pero tiene como inconveniente que cada vez que se cambia algo en el programa hay que volver a traducirlo a código máquina. Al proceso de traducción del programa a código máquina se le denomina **compilación**, y al programa encargado de ello **compilador**. Ejemplos de lenguajes normalmente compilados son el C, el Pascal o el FORTRAN.

1.5. Resumen

En este capítulo se ha estudiado de una manera muy somera la estructura interna de un ordenador. La idea principal desde el punto de vista del programador es el modelo de memoria, que puede entenderse como un vector de posiciones de memoria en cada una de las cuales se almacena un dato (normalmente un byte), de forma que cada posición tiene una dirección única con la cual podemos acceder al dato que se almacena allí. Además se ha visto que dentro del ordenador sólo pueden almacenarse datos binarios, es decir, “ristras” de ceros y unos. Es por tanto necesario establecer

¹¹Un registro interno no es más que una posición de memoria interna a la CPU que es usada por ésta para almacenar temporalmente los datos sobre los que opera.

una codificación entre las magnitudes del mundo real y las magnitudes binarias almacenadas dentro del ordenador. Es más, el ordenador es incapaz de saber por si solo si una “ristra” de unos y ceros es un número entero, un número real, un carácter o una instrucción de un programa, siendo responsabilidad del programador el saber cómo interpretar cada una de estas “ristras”. Esto último quedará más claro en el tema 3.

CAPÍTULO 2

El primer programa en C

2.1. Introducción

Una vez descrito el funcionamiento básico de un ordenador, vamos a realizar nuestro primer programa en lenguaje C. Veremos en este capítulo las herramientas necesarias para crear programas, almacenarlos, compilarlos y ejecutarlos.

2.2. El sistema operativo

En el capítulo anterior se ha descrito muy someramente el funcionamiento del ordenador. Se vio que existían unidades de entrada salida como el teclado o la pantalla y unidades de almacenamiento secundario como discos o CDROM. El manejo de estos dispositivos es altamente complejo, sobre todo para los programadores noveles, y además está estrechamente ligado al funcionamiento físico de los dispositivos, por lo que si se cambia el dispositivo, varía la forma de usarlo. Para facilitar la vida al programador, todas las tareas “sucias” del ordenador como son entre otras la gestión de la pantalla, teclado o accesos a discos las realiza el sistema operativo. Para ello, los sistemas operativos poseen una serie de funciones que hacen de interfaz entre el programador y el sistema, que se denominan **interfaz del programador de aplicaciones**, y comúnmente se conoce con las siglas inglesas API.¹

El sistema operativo también se encarga de interpretar las órdenes que el usuario le da, bien mediante una interfaz de comandos como en MS-DOS o UNIX o bien mediante una interfaz gráfica como en Microsoft Windows o en X Window System. Esto permite al usuario decirle al sistema que ejecute un programa, que borre un archivo, que lo copie, que se conecte a Internet, etc.

2.3. Creación de un programa en C

El proceso de creación de un programa en C, ilustrado en la figura 2.1, consta de los siguientes pasos: escribir el programa fuente, compilarlo y enlazarlo.

2.3.1. Primer paso: Edición del programa fuente

El primer paso a realizar para la creación de un programa es escribirlo. Para ello se necesita una herramienta llamada editor de texto, como por ejemplo el edit de MSDOS, el notepad de Windows o el vi de UNIX. Todos estos programas permiten al usuario introducir un texto en el ordenador, modificarlo y luego guardarlo en forma

¹De *Application Programmer Interface*

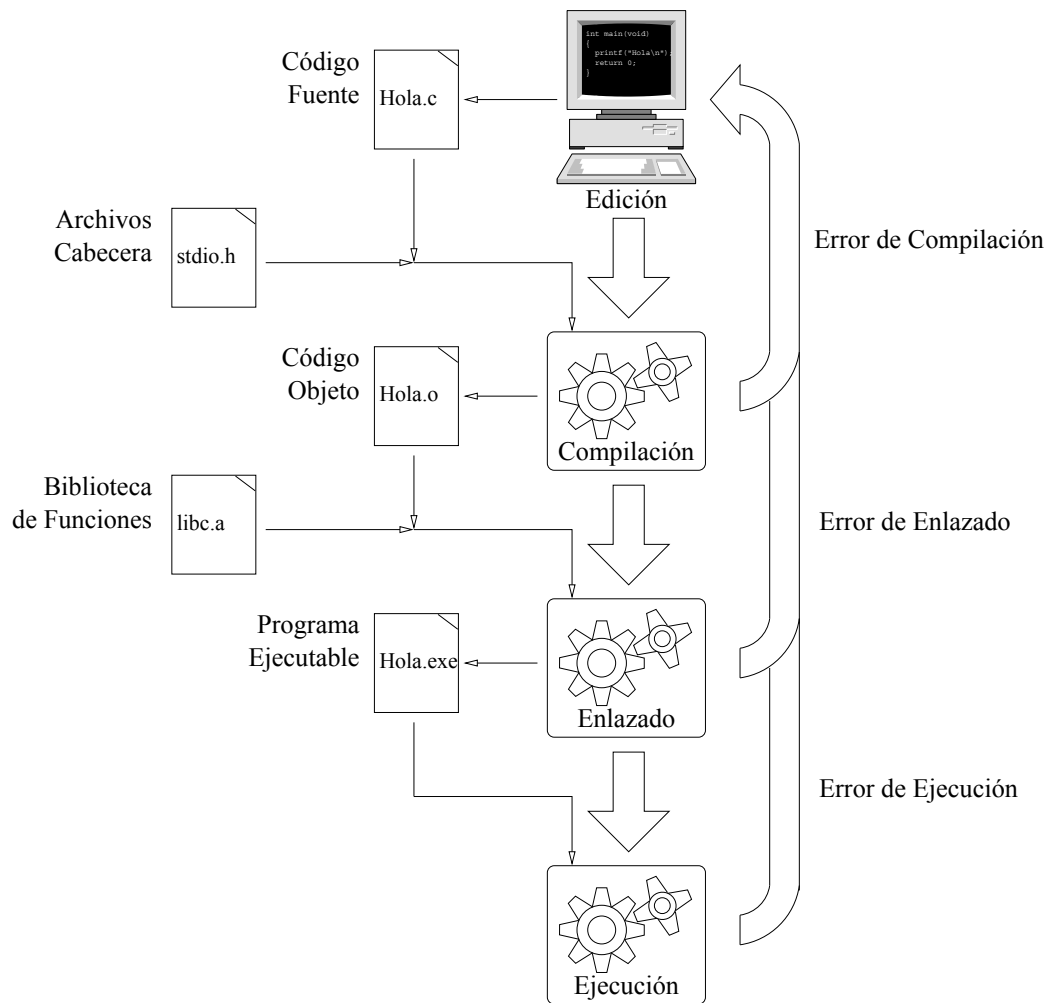


Figura 2.1: Compilación de un programa.

de archivo en el disco duro, para su posterior recuperación o para que sea usado por otros programas (como por ejemplo el compilador).

Al archivo creado se le denomina **programa fuente** o también **código fuente**. Típicamente se emplea una extensión al nombre del archivo para indicar su contenido. En el caso de archivos que contienen código fuente en C, el nombre ha de tener extensión `.c`. Por ejemplo en la figura 2.1 el código fuente que se compila se llama `Hola.c`.

2.3.2. Segundo paso: Compilación

Una vez creado el programa fuente es necesario traducirlo. De ello se encarga un programa llamado **compilador**, el cual tomando como entrada el programa fuente (con extensión `.c`) y los **archivos cabecera** (que tienen extensión `.h`), los traduce a código máquina creando lo que se denomina **código objeto** (que tienen extensión `.o` o bien `.obj`). Si existe algún error sintáctico en el código fuente el compilador generará un

mensaje de error indicando la línea en la que encontró el problema y diciéndonos la causa del error. En la mayoría de los casos el error estará en la línea indicada, aunque puede estar en líneas anteriores.

Si el programa compila sin errores, podemos pasar a la siguiente fase. Si no, habrá que volver a editar el programa fuente para corregir los errores y repetir el proceso hasta que el compilador termine su tarea con éxito.

2.3.3. Tercer paso: Enlazado

Los programas en C usan siempre funciones de propósito general que están almacenadas en una biblioteca. Ejemplos de estas funciones son las de impresión en pantalla, lectura del teclado, matemáticas, etc. Ahora bien, si hacemos uso de una de estas funciones es necesario incluirla en nuestro programa final. De esto se encarga un tercer programa llamado enlazador (*linker* en inglés), que busca en el código objeto las referencias a funciones que usa el programa y las localiza en el propio programa o en las bibliotecas de funciones (que tienen extensión `.lib` o `.a`) y las enlaza con nuestro programa. El resultado final es un **programa ejecutable** (con extensión `.exe`, `.com` o sin extensión) que contiene todo el código necesario para que el procesador realice lo que le hemos indicado en nuestro programa fuente.

Una vez realizado este paso podemos ejecutar el programa y comprobar si lo que hace es lo que realmente queremos. Si no es así, se habrá producido lo que se denomina un **error de ejecución** y habrá que volver al primer paso para corregir nuestro programa fuente y repetir el proceso: edición, compilación, enlace y ejecución, hasta que el programa haga lo que realmente queremos.

2.4. Nuestro primer programa en C

Una vez descrito todo el proceso vamos a realizar nuestro primer programa en C. El programa es muy simple: se limita a escribir un mensaje en la pantalla del ordenador. A pesar de esto contiene la mayoría de los elementos del lenguaje. El programa es:

```
/* Programa: Hola
 *
 * Descripción: Escribe un mensaje en la pantalla del ordenador
 *
 * Revisión 1.0: 16/02/2005
 *
 * Autor: José Daniel Muñoz Frías
 */

#include <stdio.h> /* Declara las funciones de entrada-salida
                  estándar */

int main(void)
{
    printf("Hola!\n"); /* Imprimo el mensaje */
    return 0; /* Indica al S.O. que el programa ha terminado
              sin error */
}
```

Veamos a continuación cada una de las partes que componen el programa:

2.4.1. Comentarios

En primer lugar vemos las líneas:

```
/* Programa: Hola
 *
 * Descripción: Escribe un mensaje en la pantalla del ordenador
 *
 * Revisión 1.0: 16/02/2005
 *
 * Autor: José Daniel Muñoz Frías
 */
```

que forman la **ficha** del programa. La finalidad de esta ficha es la documentación del programa, de forma que cualquier persona sepa el nombre, la finalidad, la revisión y el autor del programa sin más que leer el principio del archivo del código fuente.

Si observamos más detenidamente las líneas anteriores veremos que comienzan por `/*` y terminan por `*/` (los demás `*` se han colocado con fines decorativos). En C todo el texto encerrado entre `/*` y `*/` se denomina **comentario** y es ignorado por el compilador, de forma que el programador pueda escribir lo que quiera con fines de documentación del código. Esto, que puede parecer una tontería en un programa tan simple como el mostrado en esta sección, es fundamental cuando se abordan programas más grandes.

Existen dos tipos de comentarios:² los que acabamos de ver, tipo ficha del programa, y los que se insertan en el código para aclarar operaciones que no sean obvias. Ejemplos de este tipo de comentarios son:

```
#include <stdio.h> /* Declara las funciones de entrada-salida
                    estándar */

printf("Hola!\n"); /* Imprimo el mensaje */
```

En este caso, al ser el primer programa que realizamos, los comentarios incluidos son obvios, habiéndose añadido simplemente con fines ilustrativos.

2.4.2. Directivas del preprocesador

Todas las líneas que comienzan por el carácter `#` son **directivas del preprocesador** de C. Este preprocesador es una parte del compilador que se encarga de realizar varias tareas para preparar nuestro archivo de código fuente antes de realizar el proceso de compilación. Las directivas del preprocesador le dan instrucciones a éste para que realice algún tipo de proceso. Así en la línea:

```
#include <stdio.h> /* Declara las funciones de entrada-salida
                    estándar */
```

se le dice al preprocesador que **incluya** el archivo cabecera `stdio.h`. Este archivo ya ha sido creado por el desarrollador del compilador, aunque ya veremos más adelante que también nosotros podemos incluir nuestros propios archivos cabecera. El proceso de inclusión de archivos realizado por el preprocesador consiste en sustituir la línea:

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

²Desde el punto de vista del programador, pues para el compilador todos son iguales.



C99 admite, al igual que C++, comentarios que empiezan por `/**` y terminan con el final de la línea.

por el contenido del archivo `stdio.h`. Así, si suponemos que el contenido del archivo `stdio.h` es:

```
/* Este
es
el
archivo
stdio.h*/
```

el código fuente después de pasar por el preprocesador queda como:

```
/* Programa: Hola
*
* Descripción: Escribe un mensaje en la pantalla del ordenador
*
* Revisión 1.0: 16/02/2005
*
* Autor: José Daniel Muñoz Frías
*/

/* Este
es
el
archivo
stdio.h*/ /* Declara las funciones de entrada-salida
estándar */

int main(void)
{
    printf("Hola!\n"); /* Imprimo el mensaje */
    return 0; /* Indica al S. O. que el programa ha terminado
sin error */
}
```

La utilidad de incluir archivos es la de poder escribir en un archivo declaraciones de funciones, de estructuras de datos, etc. que sean usadas repetidamente por nuestros programas, de forma que no tengamos que reescribir dichas declaraciones cada vez que realizamos un nuevo programa. En este ejemplo, antes de poder usar la función `printf`, es necesario decirle al compilador que esa función existe en una biblioteca, y que no debe preocuparse si no está en nuestro archivo de código fuente, pues ya se encargará el enlazador de añadirla al programa ejecutable.

Existen más directivas del preprocesador que se irán estudiando a lo largo del libro.

2.4.3. La función principal

Todo programa en C está constituido por una o más **funciones**. Cuando se ejecuta un programa, éste ha de empezar siempre por un lugar predeterminado. En BASIC por ejemplo el programa comienza a ejecutarse por la línea 1. En C en cambio, para dotarlo de mayor flexibilidad, la ejecución arranca desde el comienzo de la función `main`. Por tanto, en nuestro programa vemos que después de incluir los archivos de cabecera necesarios, aparece la línea:

```
int main(void)
```

que indica que el **bloque** que sigue a continuación es la **definición** de la función principal. Este bloque está entre { y } y dentro están las instrucciones de nuestro programa, que en este ejemplo sencillo es sólo una llamada a una función del sistema para imprimir por pantalla un mensaje.

2.4.4. Las funciones

La realización de un programa complejo requiere la división del problema a resolver en partes más pequeñas hasta que se llega a un nivel de complejidad que puede programarse en unas pocas líneas de código. Esta metodología de diseño se denomina **Arriba-Abajo**, refinamientos sucesivos o *Top-Down* en inglés. Para permitir este tipo de desarrollo el lenguaje C permite la descomposición del programa en módulos a los que se denominan **funciones**. Estas funciones permiten agrupar las instrucciones necesarias para la resolución de un problema determinado. El uso de funciones tiene dos ventajas. La primera es que permite que unos programadores usen funciones desarrolladas por otros (trabajo en equipo). La segunda es que mejora la legibilidad del código al dividirse un programa grande en funciones de pequeño tamaño que permiten concentrarse sólo en una parte del código total.

En esta primera parte del libro sólo se van a usar funciones ya escritas por otros programadores, dejándose para la parte final el manejo y creación de funciones.

Para usar una función ya creada sólo hacen falta dos cosas:

- Incluir el archivo cabecera donde se declara la función a usar.
- Realizar la llamada a la función con los argumentos apropiados.

La primera parte ya se ha visto como se realiza, con la instrucción **#include**. La segunda requiere simplemente escribir el nombre de la función, seguida por sus argumentos encerrados entre paréntesis. Así para llamar a la función `printf` en el programa se hace:

```
printf("Hola!\n");
```

En el que se ha pasado el argumento "Hola!\n" a la función. Este argumento es una **cadena de caracteres**, que es lo que escribirá la función `printf` en la pantalla. En realidad lo que se imprime es Hola!, pues los caracteres `\n` forman lo que se denomina una **secuencia de escape**. Las secuencias de escape son grupos de dos o más caracteres que representan acciones especiales. En este caso la secuencia `\n` significa que se avance una línea. Existen más secuencias de escape que se estudiarán más adelante.

Otro detalle que se aprecia en la línea de código anterior es que al final se ha puesto un `;`. Este carácter indica el final de cada instrucción de C y ha de ponerse obligatoriamente. Esto presenta como inconveniente que a los programadores noveles se les olvida con frecuencia, con el consiguiente lío del compilador y ristra de mensajes de error, a menudo sin sentido para el pobre novatillo. La principal ventaja es que así una instrucción puede ocupar más de una línea sin ningún problema, como por ejemplo:

```
resultado_de_la_operacion = variable_1 + variable_2
                           + variable_muy_chula
                           + la_ultima_variable_que_hay_que_sumar;
```

2.4.5. Finalización del programa

La instrucción:

```
return 0; /* Indica al S. O. que el programa ha terminado
           sin error */
```

hace que la función `main`, y por tanto el programa, termine su ejecución. Además se devuelve el número 0 al Sistema Operativo. Este número devuelto por el programa es usado por el Sistema Operativo para saber si el programa se ha ejecutado satisfactoriamente (cuando devuelve un cero) o por el contrario ha fallado (cuando se devuelve un código de error distinto de cero). Esto es útil cuando se ejecutan los programas de forma automática.



2.4.6. La importancia de la estructura

El mismo programa anterior se podía haber escrito de la siguiente manera:

```
#include <stdio.h>
main(void){printf("Hola!\n");}
```

Este programa desde el punto de vista del compilador es idéntico al programa original, es decir, el compilador generará el mismo programa ejecutable que antes. El inconveniente de este “estilo” de programación es que el código fuente es bastante más difícil de leer y entender.

Por tanto, aunque al compilador le da lo mismo la estructura que posea nuestro código fuente, a los pobres lectores de nuestra obra sí que les interesa. Tenga en cuenta que un programa no es algo estático, que se escribe una sola vez y se abandona a su suerte. Por el contrario, los programas tienen un ciclo de vida en el cual es necesario volver al código para corregir errores que hemos cometido y que se manifiestan a lo largo de la vida del programa, o bien para añadir funcionalidades nuevas requeridas por el cliente. Por tanto es importante seguir unas normas de estilo para convertirse en un buen programador en C, que se resumen en:

- Escribir al principio de cada programa un comentario que incluya el nombre del programa, describa la tarea que realiza, indique la revisión, fecha y el nombre del programador.
- Cada instrucción ha de estar en una línea separada.
- Las instrucciones pertenecientes a un bloque han de estar sangradas respecto a las llaves que lo delimitan:


```
{
  printf("Hola!\n");
}
```
- Hay que comentar todos los aspectos importantes del código escrito.

2.5. Resumen

En este capítulo se han explicado todos los pasos necesarios para escribir un programa y hacerlo funcionar en el ordenador. También se ha visto el primer programa, escrito en lenguaje C, que se muestra en el libro. Se trata de un ejemplo muy sencillo

La estructura general de un programa en C está en la página web. En este ejemplo sólo se han mostrado los elementos más básicos.

que no realiza ningún cálculo, sino que se limita a mostrar un saludo por la pantalla. Sin embargo este ejemplo sirve para ir perdiendo el miedo a la programación y para ver algo funcionando en el ordenador.



Puede descargar un compilador de dominio público desde la página web del libro.

2.6. Ejercicios

1. Instale en su ordenador un entorno de desarrollo (editor y compilador) y realice los pasos descritos en el apartado 2.3 para compilar y ejecutar el programa descrito en este tema.

CAPÍTULO 3

Tipos de datos

3.1. Introducción

Este capítulo introduce las variables y los tipos de datos básicos de C (**int**, **float**, **double** y **char**). También se explican las funciones básicas de entrada y salida que van a permitir leer desde teclado y escribir por pantalla variables de cualquier tipo. Tanto las variables como las funciones de entrada y salida se utilizarán en todos los programas.

3.2. Variables

Todos los datos que maneja un programa se almacenan en variables. Estas variables tienen un nombre arbitrario definido por el programador e invisible para el usuario del programa. Ni el resultado, ni la velocidad de cálculo ni la cantidad de memoria utilizada dependen del nombre de las variables. Algunos ejemplos son `x`, `i`, `variable_temporal` y `resistencia2`.

Sin embargo hay que tener en cuenta que existen algunas restricciones en el nombre de las variables:

- Los nombres pueden contener letras 'a'..'z', 'A'..'Z', dígitos '0'..'9' y el carácter '_' ; sin embargo siempre deben comenzar por una letra. Otros caracteres de la tabla ASCII como por ejemplo el guión '-', el punto '.', el espacio ' ', el dólar '\$', etc. no son válidos.
- Algunos nombres de variables no son válidos por ser palabras reservadas del lenguaje, por ejemplo **if**, **else**, **for**, **while**, etc.

El compilador dará un error si se intentan utilizar nombres de variables no válidos.

Es importante recordar que C es un lenguaje de programación que distingue minúsculas y mayúsculas y por lo tanto la variable `X1` es diferente a la variable `x1`. Generalmente en C todos los nombres de variables se definen en minúsculas y en los nombres formados por varias palabras se utiliza el carácter '_' para separarlas. Por ejemplo suele escribirse `variable_temporal` en lugar de `variabletemporal`, que resulta más confuso de leer para el propio programador y para otros programadores que revisen su código.

3.3. Tipos básicos de variables

Según se vio en el apartado 1.3, cada tipo de dato básico (natural, entero, real y carácter) se codifica de distinta forma dentro del ordenador. Por tanto, antes de usar



Además de los tipos de datos presentados en este capítulo, C99 también admite variables lógicas tipo `_Bool` y define las constantes `true` y `false`. Además se incluyen los tipos de dato `_Imaginary` y `_Complex`.



C99 permite definir variables en cualquier parte de la función, aunque obviamente antes de ser usadas. No obstante es más estructurado definir las al principio de la función.

una variable, hay que decirle al compilador qué tipo de dato va a contener para que así sepa cuanta memoria necesita y cómo ha de codificar el dato en binario. Este proceso se conoce como **definición** de la variable, el cual ha de realizarse al principio de la función, antes de escribir las instrucciones ejecutables.¹

El lenguaje C proporciona cuatro tipos básicos de variables para manejar números enteros, números reales y caracteres. Otros tipos de variables se definen a partir de estos tipos básicos.

3.3.1. Números enteros

Los números enteros, positivos o negativos, se almacenan en variables de tipo **int**. La manera de definir una variable llamada `mi_primer_entero` como entera es escribir la siguiente línea:

```
int mi_primer_entero;
```

Una vez definida la variable se puede almacenar en ella cualquier valor entero que se encuentre dentro del rango del tipo **int**. Así, en la variable `mi_primer_entero` se puede almacenar el valor 123, pero no el valor 54196412753817103 por ser demasiado grande ni 7.4 por ser un número real. El rango de valores que pueden almacenarse en variables de tipo **int** está relacionado con el número de bytes que el ordenador utilice para este tipo de variables, según se ha visto en la sección 1.3. Al definir una variable de tipo **int**, el compilador se encarga de reservar unas posiciones de la memoria del ordenador donde se almacenará el valor de la variable. Inicialmente el contenido de estas posiciones de memoria es desconocido,² hasta que se asigna valor a la variable, por ejemplo:

```
int i;
```

```
i=7;
```

Al hacer la asignación, el valor 7 se convierte a binario (de acuerdo a la codificación de los números enteros) y el resultado se almacena en las posiciones de memoria reservadas para la variable `i`.

3.3.2. Números reales

Los números reales se almacenan en variables de tipo **float** o de tipo **double**. La manera de definir variables de estos tipos es análoga al caso de variables enteras:

```
float x;
```

```
double z;
```

La única diferencia entre **float** y **double** es que la primera utiliza menos cantidad de memoria; como consecuencia tiene menor precisión y menor rango.³ Como se vio en

¹Un ejemplo de instrucción ejecutable es el `printf("Hola!\n");` mostrado en el ejemplo del capítulo anterior.

²Un error muy común en los programadores novatos es suponer que las variables no inicializadas valen cero.

³El rango de una variable está relacionado con los valores máximo y mínimo que puede almacenar mientras que la precisión está relacionada con el número de cifras significativas (o número de decimales). Por ejemplo el rango de las variables **float** suele ser del orden de $1E+38$ y la precisión de unas 6 cifras significativas.

el capítulo de introducción, las variables reales se codifican en forma de una mantisa, donde se almacenan las cifras significativas, y un exponente. Ambos pueden ser positivos o negativos.

3.3.3. Variables de carácter

Un carácter es una letra, un dígito, un signo de puntuación o en general cualquier símbolo que pueda escribirse en pantalla o en una impresora. El lenguaje C sólo tiene un tipo básico para manejar caracteres, el tipo **char**, que permite definir variables que almacenan **un** carácter. Ya se verá más adelante que para almacenar cadenas de caracteres, por ejemplo nombres y direcciones, hay que trabajar con conjuntos de variables tipo **char** (vectores de **char**). La manera de definir la variable *c* como tipo **char** es la siguiente:

```
char c;
```

Hay que tener en cuenta que los caracteres se codifican en memoria de una manera un poco especial. Como se vio en el capítulo 1, la tabla ASCII asocia un número de orden a cada carácter y cuando se guarda una letra en una variable tipo **char**, el compilador almacena el número de orden en la posición de memoria reservada para la variable. Por ejemplo si se escribe *c*='z'; se almacenará el valor 122 ($7A_{16}$) en la posición de memoria reservada para la variable *c*; es decir se almacenará 01111010.

Teniendo en cuenta lo anterior, una variable de tipo **char** almacena un número entero, al igual que una variable de tipo **int**. La única diferencia entre ambos tipos es el tamaño, que en el caso de un **char** será el número de bits necesarios para codificar todos los caracteres de la tabla. Así, si un ordenador codifica los caracteres según la tabla ISO 8859-1, serán necesarios 8 bits (256 caracteres distintos). Si en cambio se usa el juego de caracteres Unicode, entonces se necesitan 16 bits para almacenar cada carácter.



3.4. Más tipos de variables

Existen unos modificadores que permiten definir más tipos de variables a partir de los tipos básicos. Estos modificadores son **short** (corto), **long** (largo) y **unsigned** (sin signo), que pueden combinarse con los tipos básicos de distintas maneras. Los nuevos tipos de variables a los que dan lugar son los siguientes (algunos pueden abreviarse):

```
short int           = short
long int           = long
long double
unsigned int       = unsigned
unsigned short int = unsigned short
unsigned long int  = unsigned long
unsigned char
```

En general los modificadores **short** y **long** cambian el tamaño de la variable básica y por lo tanto cambian el rango de valores que pueden almacenar. Por otro lado el modificador **unsigned**, que no puede aplicarse a los tipos reales, modifica el rango de las variables sin cambiar el espacio que ocupan en memoria. Por ejemplo, en un ordenador donde el tipo **int** puede almacenar valores en el rango desde -32768 hasta 32767, una variable **unsigned int** tendría un rango desde 0 hasta 65535, ya que no utiliza números negativos. En ambos casos el número de valores diferentes que pueden tomar las variables es el mismo (65536 valores posibles = 2^{16}).

Realice el ejercicio 1.



Para soportar mejor la potencia de los nuevos procesadores de 64 bits, C99 incluye nuevos tipos de dato llamados **long long** y **unsigned long long** que deben tener al menos 64 bits.

3.5. Constantes

Las constantes son, en general, cualquier número que se escribe en el código y que por lo tanto no puede modificarse una vez que el programa ha sido compilado y enlazado. Si escribimos por ejemplo `i=7`, la variable `i` tomará siempre el valor 7, ya que este número es fijo en el programa: es una constante.

El compilador considera como constantes de tipo **int** todos los **números enteros**, positivos o negativos, que se escriben en un programa. Para que estas constantes se consideren de tipo **long** deben llevar al final una `l` o una `L`,⁴ para que se consideren de tipo **unsigned** deben llevar una `u` o una `U` y los enteros largos sin signo deben llevar `ul` o bien `UL`. Véase el siguiente ejemplo:



En C99 las constantes numéricas **long long** deben llevar sufijos `LL` y las **unsigned long long** deben llevar `ULL`.

```
int entero;
unsigned sin_signo;
long largo;
unsigned long sin_signo_largo;

entero=7;
sin_signo=55789u;
largo=-300123456L; /* Es mejor L, porque l se confunde con un 1 */
sin_signo_largo=2000000000UL;
```

Las constantes de tipo entero también pueden escribirse en octal y en hexadecimal, lo que nos ahorra tener que hacer las conversiones a decimal mientras escribimos algunos programas. Los números en base 8 se escriben empezando con cero y en base 16 empezando por `0x` o por `0X`. Las tres líneas siguientes son equivalentes:

```
i=31; /* Decimal */
i=037; /* Octal */
i=0x1F; /* Hexadecimal, también vale poner 0x1f */
```

Los **números reales** se identifican porque llevan un punto (123.4) o un exponente (1e-2) o ambas cosas. En estos casos la constante se considera de tipo **double**. Por ejemplo:

```
double x;

x = 3.7; /* Constante real */
```

Las **constantes de caracteres** se escriben entre apóstrofes (o comillas simples) como en el siguiente ejemplo:

```
char c;

c='z'; /* Constante de carácter */
```

En realidad las constantes de caracteres se convierten automáticamente a tipo **int** de acuerdo a la tabla ASCII, o sea que la instrucción anterior es equivalente a escribir:

```
char c;

c=0x7A; /* Esta asignación es correcta en C,
         aunque c es de tipo char, no int */
```

⁴Dado que en las tipografías usadas para representar el código la `l` (ele) es muy similar al `1` (uno), es mejor usar la `L` mayúscula para identificar las constantes de tipo **long**.

Determinados caracteres especiales se pueden escribir utilizando secuencias de escape que comienzan por el carácter \, como \n en el programa del capítulo 2. Los ejemplos más típicos son los siguientes:

```
'\n'   cambio de línea (new line)
'\r'   retorno de carro (Return)
'\0'   carácter nulo (NULL). No confundir con el carácter '0'.
'\t'   TAB
'\f'   cambio de página (form feed)
'\'    apóstrofe
'\"    comillas
'\\'   la barra \
```

3.6. Tamaño de las variables

El tamaño que cada tipo de variable ocupa en la memoria no está fijado por el lenguaje, sino que depende del tipo del ordenador y a veces del tipo de compilador que se utilice. Todo programa que necesite conocer el tamaño de una determinada variable puede utilizar el operador **sizeof** para obtenerlo, pero es una mala técnica de programación suponer un tamaño fijo. El operador **sizeof** puede utilizarse tanto con tipos concretos (**sizeof(int)**) como con variables (**sizeof(i)**). El operador **sizeof** devuelve un **entero** que es el número de bytes que ocupa la variable en memoria.

Como regla general, el tipo **char** ocupa 1 byte (2 si se usa Unicode), el tipo **short** ocupa al menos 2 bytes en todos los compiladores y **long** ocupa un mínimo de 4, mientras que **int** suele ser de tamaño 2 ó 4 dependiendo del tipo de ordenador (mínimo 2). En definitiva, lo que ocurrirá en cualquier implantación del lenguaje es que:

$$\text{sizeof(char)} \leq \text{sizeof(short int)} \leq \text{sizeof(int)} \leq \text{sizeof(long int)}$$

3.7. Escritura de variables en pantalla

La función más utilizada para mostrar el valor de una variable por pantalla es la función **printf**, que forma parte de la biblioteca estándar de funciones de entrada y salida en C. Esta función está declarada en el archivo de cabeceras (*header file*) **stdio.h**, por lo tanto es necesario incluir este archivo en el programa para que el compilador pueda hacer las comprobaciones oportunas sobre la sintaxis de la llamada a **printf**. Sin embargo, al ser una función de la biblioteca estándar, no es necesario incluir ninguna biblioteca (*library*) especial para generar el ejecutable, ya que dicha librería se enlaza por defecto.

La función **printf** permite escribir cualquier texto por pantalla, como se mostró en el programa **hola.c** del capítulo anterior. Pero también se puede utilizar para mostrar el valor de cualquier variable y además nos permite especificar el formato de salida. Los valores de las variables pueden aparecer mezclados con el texto, por lo tanto es necesario definir una cadena de caracteres especial donde quede clara la posición en la que deben mostrarse los valores de las variables. Las posiciones donde aparecen las variables se definen utilizando el carácter % seguido de una letra que indica el tipo de la variable. Veamos un ejemplo:

```
/******
Programa: Entero.c
Descripción: Escribe un número entero
```

Revisión 0.1: 16/FEB/1999

Autor: Rafael Palacios

******/*

#include <stdio.h>

int main(void)

{

int dia;

dia=47;

printf("Hoy es el día %d del año\n", dia);

return 0;

}



Tanto C99 como C++ permiten declarar variables en cualquier parte del código. Sin embargo esto no funcionaría en un compilador de C puro y el código es menos claro.

En el programa anterior la función `printf` sustituye el texto `%d` por el valor de la variable `dia` expresada como un entero. Este programa escribe por pantalla la siguiente línea:

Hoy es el día 47 del año

Cada tipo de variable requiere utilizar una letra diferente después del carácter `%`, los formatos más normales para los tipos básicos de variables se muestran en el siguiente cuadro:

Formato	Tipo de variable
<code>%d</code>	int
<code>%f</code>	float y double
<code>%c</code>	char

La función `printf` puede escribir más de una variable a la vez, del mismo tipo o de tipo distinto, pero hay que tener cuidado para incluir en la cadena de caracteres tantos `%` como número de variables. Además hay que tener cuidado para introducir correctamente los formatos de cada tipo de variable. Ejemplo:

*/******

Programa: Tipos.c

Descripción: Escribe variables de distintos tipos

Revisión 0.1: 16/FEB/1999

Autor: Rafael Palacios

******/*

#include <stdio.h>

int main(void)

{

int dia;

double temp;

char unidad;

dia=47;

temp=11.4;

unidad='C';

```
printf("Día %d, Temperatura %f %c\n", dia, temp, unidad);
return 0;
}
```

```
----- Salida -----
Día 47, Temperatura 11.400000 C
```

3.7.1. Escritura de variables de tipo entero

Las variables de tipo entero pueden escribirse en varios formatos: decimal, octal o hexadecimal. Sin embargo, la función `printf` no proporciona un formato para escribir el valor en binario.

Los formatos para variables de tipo entero son `%d` para mostrar el valor en decimal, `%o` para mostrarlo en octal y `%X` o `%x` para mostrarlo en hexadecimal.⁵ En formato octal y hexadecimal se muestra el valor sin signo.

```
printf("Decimal %d, en octal %o, en hexadecimal %X\n", dia, dia, dia);
```

escribe la siguiente línea en pantalla:

```
Decimal 47, en octal 57, en hexadecimal 2F
```

Los tipos de variables derivados de **int**, como **short**, **long**, **unsigned**, etc. utilizan los caracteres de formato que se muestran en el siguiente cuadro:

Formato	Tipo de variable
<code>%d</code>	int
<code>%hd</code>	short
<code>%ld</code>	long
<code>%u</code>	unsigned int
<code>%hu</code>	unsigned short
<code>%lu</code>	unsigned long

3.7.2. Escritura de variables de tipo real

Las variables de tipo real, tanto **float** como **double** se escriben con formato `%f`, `%e`, `%E`, `%g` o `%G`, dependiendo del aspecto que se quiera obtener. Con `%f` siempre se escribe el punto decimal y no se escribe exponente, mientras que con `%e` siempre se escribe una e para el exponente y con `%E` se escribe una E. Por ejemplo:

```
/******
```

```
Programa: Formatos.c
```

```
Descripción: Escribe números reales en distintos formatos
```

```
Revisión 0.0: 16/FEB/1999
```

```
Autor: Rafael Palacios
```

```
*****/
```

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

⁵La única diferencia entre `%X` y `%x` es que con el primer formato los símbolos de la A a la F del número en base hexadecimal se escriben en mayúsculas mientras que con el segundo se escriben en minúsculas.


```
int main(void)
{
    printf("%f\n", 12345.6789);
    printf("%e\n", 12345.6789);
    printf("%E\n", 12345.6789);
    return 0;
}
```

```
----- Salida -----
12345.678900
1.234568e+04
1.234568E+04
```

El formato %g es el más seguro cuando no se conoce de qué orden es el valor a escribir ya que elige automáticamente entre %f (que es más bonito) y %e (donde cabe cualquier número por grande que sea).

3.7.3. Escritura de variables de tipo carácter

Las variables tipo **char** se escriben con formato %c lo que hace que aparezca el carácter en pantalla. También es cierto que una variable tipo **char** puede escribirse en formato decimal, es decir con %d, ya que los tipos **char** e **int** son compatibles en C. En caso de escribir una variable tipo **char** con formato %d se obtiene el número de orden correspondiente en la tabla ASCII. Comprobad el siguiente programa viendo la tabla ASCII.

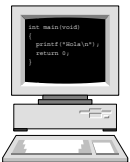
```
/* *****
 * Programa: Caracter.c
 * Descripción: Escribe variables de tipo char
 * Revisión 0.0: 16/FEB/1999
 * Autor: Rafael Palacios
 * *****/
```

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

```
int main(void)
{
    char c;

    c='z';
    printf("Carácter %c\n", c);
    printf("Valor      %x\n", c);
    return 0;
}
```

```
----- Salida -----
Carácter z
Valor      7a
```



Realice el ejercicio 2.

Es importante recordar que las variables de tipo **char** sólo almacenan un carácter, es decir, una letra. Para trabajar con palabras o frases es necesario declarar vectores

de **char** y para mostrarlos con `printf` se utiliza `%s` en lugar de `%c`. Esto se verá más adelante en el capítulo 8.

3.7.4. Escritura de variables con formato

La función `printf` también permite definir el tamaño con el que aparece cada variable en la pantalla y la alineación, derecha o izquierda, que debe tener. Esto se hace escribiendo números entre el carácter `%` y el carácter que define el tipo de variable. En lugar de enumerar las reglas que se utilizan para definir estos formatos, lo más práctico es mostrar una lista de ejemplos que se puede utilizar como referencia rápida con mayor comodidad.

Variables enteras:

```
printf(":%5d: \n",i);    --> : 123: Hay espacio suficiente
printf(":%-5d: \n",i);   --> :123 : Alineación izquierda
printf(":%05d: \n",i);   --> :00123: Llena con ceros
printf(":%2d: \n",i);    --> :123: No cabe, formato por defecto
```

Variables reales:

```
printf(":%12f: \n",x);   --> : 1024.251000: Por defecto 6 dec.
printf(":%12.4f: \n",x); --> : 1024.2510: .4 indica 4 decimales
printf(":%-12.4f: \n",x); --> :1024.2510 : Alineación izquierda
printf(":%12.1f: \n",x); --> : 1024.3: Redondea correctamente
printf(":%3f: \n",x);    --> :1024.251000: No cabe-> por defecto
printf(":%.3f: \n",x);   --> :1024.251: Por defecto con 3 dec.

printf(":%12e: \n",x);   --> :1.024251e+03: 12 caracteres
printf(":%12.4e: \n",x); --> : 1.0243e+03: Idem. pero con 4 dec.
printf(":%12.1e: \n",x); --> : 1.0e+03: Idem. pero con 1 dec.
printf(":%3e: \n",x);    --> :1.024251e+03: No cabe. Por defecto
printf(":%.3e: \n",x);   --> :1.024e+03: 3 decimales.
```

3.8. Lectura de variables por teclado

La función más cómoda para lectura de variables desde teclado es la función `scanf`, que tiene un formato similar a `printf`. También se trata de una función de la biblioteca estándar de entrada y salida y por lo tanto funciona en cualquier compilador de C. En este apartado sólo se verá la manera de leer variables de tipo básico aunque `scanf` es una función potente que admite muchas opciones.

El siguiente programa lee una variable del teclado y escribe el valor en pantalla:

```
/******
Programa: Leer.c
Descripción: Lee una variable con scanf
Revisión 0.0: 16/MAR/1998
Autor: Rafael Palacios
*****/

#include <stdio.h>
```

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

    printf("Dame un valor ");
    scanf("%d", &a);
    printf("Has escrito %d\n", a);
    return 0;
}
```

Puede observarse que la sintaxis de `scanf` y de `printf` es similar. La función `scanf` recibe una cadena de caracteres donde se define el formato de la variable a leer y luego el nombre de la variable precedido de un signo `&`. Este signo `&` es fundamental para que `scanf` funcione correctamente. En el capítulo de funciones se explicará por qué es necesario; de momento haga un pequeño acto de Fe. Los formatos de los distintos tipos de variables son los mismos que para `printf` salvo que las variables **float** se leen con formato `%f` y las **double** se leen con `%lf`. Para escritura ambas utilizan el formato `%f`, si bien la mayoría de los compiladores también admiten especificar el formato como `%lf` para escritura y se evita la pequeña inconsistencia que existe entre `printf` y `scanf`.

3.9. Recomendaciones y advertencias

- Antes de empezar a desarrollar un programa, hay que pensar qué variables se necesitan y de qué tipo han de ser.
- Las definiciones de variables han de realizarse al principio de la función, antes de las instrucciones ejecutables.
- El uso de variables enteras es más eficiente que las reales y no se acumulan errores de redondeo. No se deben definir todas las variables como reales pensando que “así funciona todo”.
- Las advertencias *warnings* que da el compilador sobre el uso de variables no se deben ignorar. Generalmente nos dan pistas sobre problemas importantes en el programa. Es decir, aunque el compilador haya sido capaz de generar un programa ejecutable no quiere decir que éste sea correcto.
- Los nombres de las variables se han de escribir con letras minúsculas, usando el carácter `'_'` para separar las palabras en los nombres compuestos, como por ejemplo en: `variable_interesante`.
- Los nombres de las variables han de indicar su función dentro del programa. Salvo los nombres `'n'`, `'m'`, `'i'` o `'j'` que suelen usarse como contadores en los bucles, el resto han de ser autoexplicativos. Es muy frecuente encontrar en programas escritos por programadores novatos un cúmulo de variables denominadas `aux`, `aux1`, `aux2`, `aux3`, etc. que pasados unos días ni siquiera el autor es capaz de recordar para qué servía cada una de ellas.

3.10. Resumen

En este capítulo se ha expuesto cómo se usan las variables en C. Se ha visto que es fundamental indicar el **tipo** de dato que va a almacenar una variable y se han

estudiado los distintos tipos básicos que admite el lenguaje C. Por último se ha visto cómo se pueden imprimir los distintos tipos de variables usando la función `printf` y cómo pueden leerse desde datos desde el teclado y almacenarlos en variables mediante la función `scanf`.

3.11. Ejercicios

1. ¿Es igual hacer `c='9'`; que `c=9`? Suponiendo que el ordenador codifica los caracteres según la tabla ISO 8859-1, ¿Qué valor se almacenará en la posición de memoria reservada para la variable `c` en cada caso?
2. Modifique el programa `Caracter.c` de la página 26 para que en lugar de imprimir el valor hexadecimal como `7a` lo imprima como `0x7a`. ¿Qué habrá que hacer para que se imprima `0x7A`.
3. Escriba un cuadro con todos los tipos de datos que incluya los formatos que utiliza cada uno con `printf` y `scanf`.
4. Escriba un programa que imprima el código ASCII de cada una de las letras de su nombre de pila.
5. Escriba un programa que imprima los tamaños en bytes de los tipos básicos del lenguaje C (**char**, **short**, **int**, **long**, **float** y **double**).
6. Escriba un programa que declare una variable de tipo **unsigned long**, otra de tipo **int** y otra **double**. El programa debe leer las tres variables con `scanf` y luego escribir los valores con `printf`.

CAPÍTULO 4

Operadores Aritméticos

4.1. Introducción

En el capítulo anterior se han visto los tipos de datos básicos del lenguaje C, cómo leer estos datos desde teclado y cómo imprimirlos en la pantalla. Puesto que los ordenadores son equipos cuya misión es el proceso de datos, los lenguajes de programación proveen al programador con una serie de operadores que permiten realizar operaciones matemáticas básicas sobre los datos. En este capítulo vamos a estudiar los distintos operadores aritméticos soportados por el lenguaje C. Existen además otros operadores que permiten trabajar con expresiones lógicas. Dichos operadores se estudiarán más adelante junto con las instrucciones de control de flujo, que es donde se usan principalmente.

4.2. El operador de asignación

El operador de asignación en C se representa mediante el signo `=`. Este operador asigna el valor de la expresión situada a su derecha a la variable situada a su izquierda. Así por ejemplo, para asignar a la variable `n` el valor 2 se escribe:

```
n = 2;
```

No ha de confundirse este operador con la igualdad en sentido matemático, aunque se use el mismo símbolo. En matemáticas se puede escribir $4 = 2 + 2$, cosa que daría un soberano error en C al intentar compilar, pues 4 no es una variable. Tampoco es legal en C la instrucción `2 = n`, aunque matemáticamente sea una expresión correcta.

En resumen el operador de asignación en C evalúa la expresión situada a su derecha y asigna su valor a la variable situada a su izquierda:

variable = expresión;

4.3. Operadores para números reales

En la mayoría de las aplicaciones es necesario operar con números reales. Para hacer esto posible en el lenguaje C, éste soporta las operaciones matemáticas básicas, es decir, la suma, resta, multiplicación y división. Estas operaciones se representan en C mediante los operadores `+`, `-`, `*` y `/` respectivamente.

Supongamos que estamos desarrollando un programa de facturación y que tenemos que calcular el importe total a partir de la base imponible para un IVA del 16%. La forma de realizarlo sería:

```
total = base * 1.16;
```

En donde se realiza el producto de la variable base por la constante 1.16 para después asignar el resultado a la variable total.

En una misma expresión se pueden combinar varios operadores, aplicándose en este caso las mismas reglas de **precedencia** usadas en matemáticas, es decir, en primer lugar se realizan las multiplicaciones y las divisiones, seguidas de las sumas y restas. Cuando las operaciones tienen la misma precedencia, por ejemplo $2*\pi*r$, éstas se realizan de izquierda a derecha.

Siguiendo con el ejemplo anterior, si tenemos dos artículos de los cuales el primero es un libro, al que se le aplica un 4% de IVA y el segundo una tableta de chocolate, a la que se le aplica (injustamente) un IVA del 16%, el cálculo del importe total de la factura se realizaría en C mediante la siguiente línea de código:

```
total = 1.04*valor_libro + 1.16*valor_chocolate;
```

En donde en primer lugar se evalúa la expresión $1.04*\text{valor_libro}$, seguidamente $1.16*\text{valor_chocolate}$ y por último se suman los resultados de ambas operaciones, asignándose el valor final a la variable total.

Si se desea alterar el orden de precedencia preestablecido, se pueden usar paréntesis al igual que en matemáticas, eso sí, sólo paréntesis; las llaves y los corchetes tienen otros significados en el lenguaje. Siguiendo con el ejemplo anterior, si los dos artículos tienen el mismo tipo de IVA, el cálculo del total de la factura se escribiría:

```
total = 1.16*(valor_chocolate + valor_helado);
```

En este caso se realiza en primer lugar la suma de la variable `valor_chocolate` a la variable `valor_helado` y el resultado se multiplicará por la constante 1.16, asignándose el resultado a la variable total.

Si hubiésemos escrito la línea anterior sin los paréntesis, es decir:

```
total = 1.16*valor_chocolate + valor_helado; /* Factura mal
                                              calculada*/
```

El programa multiplicaría 1.16 por `valor_chocolate` y al resultado le sumaría el `valor_helado`, por lo que seríamos perseguidos duramente por el fisco por facturar helados sin IVA.

4.4. Operadores para números enteros

El lenguaje C también soporta las operaciones básicas para los números enteros. La funcionalidad y los símbolos usados son los mismos que los usados para los números reales. La única diferencia está en el operador de división, que en el caso de números enteros da como resultado la parte entera de la división. Así por ejemplo $3/2$ da como resultado 1, lo que puede parecer malo, pero aún hay cosas peores como $1/2$, que da como resultado 0, lo cual tiene un gran peligro en expresiones como:

```
resultado = 1/2*4;
```

Como los dos operadores tienen la misma precedencia, la expresión se evalúa de izquierda a derecha, con lo cual en primer lugar se efectúa $1/2$ con el resultado antes anunciado de 0 patatero, que al multiplicarlo después por el 4 vuelve a dar 0, como todo el mundo habrá ya adivinado. Si en cambio se escribe:

```
resultado = 4*1/2;
```

El resultado ahora es de 2, con lo que se aprecia que en lenguaje C cuando se trabaja con números enteros lo de la propiedad conmutativa no funciona nada bien, y deja bien claro que, en contra de la creencia popular, los ordenadores no son infalibles realizando cálculos matemáticos, sobre todo cuando el programador no sabe muy bien lo que está haciendo y descuida temas como los redondeos de las variables o los rangos máximos de los tipos.

4.4.1. Operador resto

Siguiendo con la división de números enteros, existe un operador que nos da el **resto** de la división (no es tan bueno como tener los decimales, pero al menos es un consuelo). El operador **resto** se representa mediante el símbolo %. Así por ejemplo $4\%2$ da como resultado 0 y $1\%2$ da como resultado 1.

Una utilidad de este operador es la de averiguar si un determinado número es múltiplo de otro; por ejemplo el número 4580169 es múltiplo de 33 porque $4580169\%33$ da cero.

Por supuesto este operador no tiene sentido con números reales, por lo que el compilador se quejará si lo intentamos usar en ese caso.



Realice el ejercicio 1.

4.4.2. Operadores de incremento y decremento

Dado que una de las aplicaciones principales de los números enteros en los programas es la realización de contadores, que usualmente se incrementan o decrementan de uno en uno, los diseñadores de C vieron aconsejable definir unos operadores para este tipo de operaciones.¹ El operador incremento se representa añadiendo a la variable que se desee incrementar dos símbolos +. La sintaxis es por tanto: *NombreVariable++*. Así por ejemplo la línea:

```
Contador++;
```

Sumaría uno a la variable Contador.

El operador decremento es idéntico al de incremento, sin mas que sustituir los + por -. Siguiendo el ejemplo anterior:

```
Contador--;
```

Le restará uno a la variable Contador.

4.5. Operador de cambio de signo

Hasta ahora, todos los operadores que se han discutido eran binarios, es decir, operaban con dos números: el situado a su izquierda con el situado a su derecha. El operador de cambio de signo, también denominado “operador unario -” por algunos autores, toma sólo un valor situado a su derecha y le cambia el signo. Así en:

```
i = -c;
```

el operador - toma el valor de la variable c y le cambia el signo. Por tanto si c vale 17, al finalizar la ejecución de la instrucción, la variable i contendrá el valor -17.

¹Además la mayoría de los procesadores tienen instrucciones especiales de incremento y decremento, usualmente más rápidas que la de suma normal, con lo que estos operadores le permiten al compilador usar ese tipo de instrucciones, consiguiendo un programa más eficiente.

4.6. De vuelta con el operador de asignación

Ya se dijo al principio de este capítulo que el operador de asignación en C no debía confundirse con la igualdad en sentido matemático. Un uso muy frecuente del operador de asignación en C es el ilustrado en la instrucción:

```
i = i + 7;
```

Esto, que matemáticamente no tiene ningún sentido, en los lenguajes de programación es una práctica muy común. El funcionamiento de la instrucción no tiene nada de especial. Tal como se explicó en la sección 4.2 el operador de asignación evalúa en primer lugar la expresión que hay situada a su derecha para después asignársela a la variable situada a su izquierda. Obviamente da igual que la variable a la que le asignamos el valor intervenga en la expresión, tal como ocurre en este caso. Así pues en el ejemplo anterior, si la variable *i* vale 3, al evaluar la expresión se le sumará un 7, con lo que el resultado de la evaluación será 10, que finalmente se almacenará en la variable *i*.

Como este tipo de instrucciones son muy comunes en C, existe una construcción específica para realizarlas: preceder el operador = por la operación a realizar (+ en este caso). Así la instrucción anterior se puede escribir:

```
i += 7;
```

Con los demás operadores la situación es similar: *t /= 2* asigna a *t* el resultado de la expresión *t/2*.

La ventaja de este tipo de construcción queda manifiesta cuando se usa un nombre de variable largo:

```
variable_de_cuyo_nombre_no_quiero_acordarme += 2;
```

En donde, aparte del ahorro en el desgaste del teclado, nos ahorramos la posibilidad de escribir mal la variable la segunda vez, y facilitamos la lectura a otros programadores, que no tienen que comprobar que las variables a ambos lados del = son la misma.

4.7. Conversiones de tipos

En las expresiones se pueden mezclar variables de distintos tipos, es decir, un operador binario puede tener a su izquierda un operando de un tipo, como por ejemplo **int** y a su derecha un operando de otro tipo, como por ejemplo **double**. En estos casos el compilador automáticamente se encargará de realizar las **conversiones de tipos** adecuadas. Estas conversiones trabajan siempre **promoviendo** el tipo “inferior” al tipo “superior”, obteniéndose un resultado que es del tipo “superior”. De esta forma las operaciones se realizan siempre con la precisión adecuada. Por ejemplo si *i* es una variable de tipo **int** y *d* es de tipo **double**, la expresión *i*d* convierte en primer lugar el valor de *i* a tipo **double** y luego multiplica dicho valor por *d*. El resultado obtenido de la operación es también de tipo **double**.

Las reglas de conversión se resumen en:

Si cualquier operando es de tipo **long double**

Se convierte el otro operando a **long double**

Si no: Si cualquier operando es de tipo **double**

Se convierte el otro operando a **double**

Si no: Si cualquier operando es de tipo **float**

Se convierte el otro operando a **float**
 Si no: Si cualquier operando es de tipo **long**
 Se convierte el otro operando a **long**
 Si no:
 Se convierten **char** y **short** a **int**

La última línea quiere decir que aunque los operandos de una operación sean los dos de tipo **char** o **short**, el compilador los promociona automáticamente a **int**, para realizar la operación con mayor precisión.

Cuidado con la división y los enteros

Como se ha visto anteriormente, la división de valores **int** puede producir resultados inesperados para programadores novatos. En una expresión como la siguiente:

```
resultado = 1/2*d;
```

como todos los operadores tiene la misma precedencia, la expresión se evalúa de izquierda a derecha, realizándose en primer lugar la operación $1/2$, en la que como los dos operandos son constantes de tipo **int**, se realiza la división entera, obteniéndose un 0 como resultado. La siguiente operación a realizar será la multiplicación que, si d es de tipo **double**, convertirá el resultado anterior (0) a **double** y luego lo multiplicará por d , obteniéndose cero como resultado final. Si se realiza la operación al revés, es decir, $d*1/2$ el resultado será ahora correcto ¿por qué?

Para evitar este tipo de situaciones conviene evitar usar constantes enteras cuando trabajemos en expresiones con valores reales. Así la mejor manera de evitar el problema anterior es escribir:

```
resultado = 1.0/2.0*d
```

o bien:

```
resultado = 0.5*d
```

Ahora bien, este método no es válido cuando tenemos que usar en una expresión variables enteras junto con variables reales. En este caso puede ser necesario forzar una conversión que el compilador no realizaría de modo automático. Por ejemplo, si las variables $i1$ e $i2$ son de tipo **int** y la variable d es de tipo **double**, en la expresión $i1/i2*d$ se realizará la división de números enteros, con el consabido peligro y falta de precisión. Para conseguir que la división se realice con números de tipo **double** podemos reordenar las operaciones como se hizo antes, o mejor, forzar al compilador a que realice la conversión. Para forzar la conversión se utiliza un operador **molde**, llamado **cast** en inglés, que tiene el formato:

```
(tipo_al_que_se_desea_convertir) variable_a_convertir
```

El ejemplo anterior forzando las conversiones quedaría como:

```
resultado = (double) i1 / (double) i2 * d;
```

Como el lector aventajado habrá notado, sólo sería necesario poner un molde en cualquiera de las dos variables enteras ¿Por qué?



4.7.1. El operador asignación y las conversiones

Si el resultado de una expresión no es del tipo de la variable a la que es asignado, el compilador realizará automáticamente la conversión necesaria. En caso de que el tipo de la variable sea “superior” al resultado de la expresión no existirá ningún problema, pero si es inferior, se pueden producir pérdidas de precisión (por ejemplo al convertir de **double** a **int** se pierde la parte decimal). Es habitual que el compilador de un aviso en estos casos, ya que la pérdida de precisión puede dar lugar a resultados erróneos (por ejemplo al asignar un valor grande de tipo **long** a un **int**).

Como resumen a esta sección cabe decir que se ha de ser extremadamente cuidadoso/a cuando en una instrucción se mezclen variables y constantes de tipos distintos para evitar resultados erróneos. Estos errores de programación a menudo sólo se manifiestan en ciertas situaciones, funcionando el programa correctamente en las demás, lo que dificulta su detección.

4.7.2. Ejemplos de conversión de variables

Para afianzar los conceptos vamos a estudiar a continuación algunos ejemplos de expresiones matemáticas en C.

```
main(void)
{
    int ia, ib;
    int ires;
    double da, db;
    double dres;

    ia = 1;
    ib = 3;
    da = 2.3;
    db = 3.7;

    ires = ia + (ib*2 + ia*3); /* Ej 1 */
    dres = ia + (ib*2 + ia*3); /* Ej 2 */
    dres = ia + da*(ib/2.0 + ia/2.0); /* Ej 3 */
    ires = ia + da*(ib/2.0 + ia/2.0); /* Ej 4 */
    dres = -da*(ia + ib*(da + db/3.0)); /* Ej 5 */
}
```

En el primer ejemplo, se evalúa en primer lugar la expresión que está entre paréntesis. Como esta expresión contiene multiplicaciones y sumas, en primer lugar se realizan las multiplicaciones, sumándose seguidamente los resultados. Por último al resultado se le suma *ia* y se almacena el número obtenido en la variable *ires*. Como todas las variables involucradas son de tipo **int** no se realiza ninguna conversión de tipos. Nótese que en este caso los paréntesis son innecesarios.

El segundo ejemplo es idéntico al primero, salvo que el resultado, que recordemos es de tipo **int**, se almacena en una variable de tipo **double**. Se realizará por tanto una conversión de tipos del resultado de **int** a **double** antes de realizar la asignación, lo que no debe dar ningún problema.

En el tercer ejemplo se realizan en primer lugar las divisiones de dentro del paréntesis, en las que debido a que el segundo operando es una constante real, los operandos enteros se convierten a **double**, obteniéndose por tanto un resultado de tipo **double**

que se multiplica por *da* para sumarlo después a *ia*, que al ser un **int** ha de convertirse previamente a **double**. El resultado se almacena en una variable de tipo **double**, por lo que no se realizan más conversiones de tipos.

En el cuarto ejemplo el proceso es el descrito para el ejemplo anterior, salvo que al asignar el número de tipo **double** que resulta a uno de tipo **int**, se realiza una conversión de **double** a **int**, por lo que se almacena en *ires* la parte entera del resultado de la expresión, siempre y cuando esta esté dentro el rango del tipo **int**, perdiéndose para siempre la parte decimal. En este caso el compilador suele dar un aviso.

Por último en el ejemplo quinto se muestra la posibilidad de anidar varias expresiones entre paréntesis. En este caso se evalúa en primer lugar el paréntesis interno, el resultado se multiplica por *ib* y después se suma *ia* al resultado anterior. Todo ello se multiplica por *da* cambiando de signo. ¿Qué conversiones de tipos se han realizado en este ejemplo?



Realice el ejercicio 3.

4.8. Otras operaciones matemáticas

El lenguaje C sólo proporciona operadores para cálculos muy básicos (+, -, * y /). El resto de operaciones (potenciación,² logaritmos, funciones trigonométricas, etc.) están implantadas en una biblioteca matemática, debiendo incluirse el archivo `math.h` si se desean usar.³ Por ejemplo para calcular la raíz cuadrada del valor contenido en la variable *x* y almacenar el resultado en la variable *raiz_x* se utiliza la función `sqrt` pasándole *x* como argumento:

```
raiz_x = sqrt(x);
```

El uso de funciones y como desarrollar nuestras propias funciones se verá más adelante en el capítulo 7. Sin embargo es bueno saber que la mayoría de las operaciones matemáticas están implementadas dentro de la biblioteca matemática y pueden consultarse en la referencia de C.



En la página web del libro hay un enlace a la referencia de C.

4.9. Recomendaciones y advertencias

- Como recomendación general, la variable que se encuentra a la izquierda de un signo '=' debe ser del mismo tipo que el resultado de la expresión que hay a la derecha.
- Cuando en una expresión hay una mezcla de variables o constantes de distintos tipos, es mejor hacer las conversiones explícitas para ahorrarse sorpresas desagradables.
- Hay que tener cuidado con las divisiones, tanto enteras como reales, porque hacen fallar al programa si el divisor es cero.
- Tenga cuidado con la división entera. Recuerde que por ejemplo `1/2` se evalúa como `0`. Esto se soluciona poniendo `1.0/2.0` o bien `0.5`.

²Al contrario que en otros lenguajes el operador ^ no es el de elevar un número a una potencia.

³Además es necesario decirle al enlazador que añada la biblioteca matemática, lo cual depende del sistema de desarrollo usado.

4.10. Resumen

En este capítulo se han visto los operadores aritméticos, los cuales permiten escribir operaciones matemáticas. Utilizando estos operadores es posible escribir programas que realicen cálculos sencillos como los que normalmente se realizan con una calculadora de bolsillo.

Se ha remarcado la diferencia que existe entre calcular un cociente de variables enteras y reales, así como los problemas de conversión de tipos de variables.

4.11. Ejercicios

1. Escriba un programa que pida al usuario dos números enteros e imprima por pantalla su cociente y su resto.
2. Indique, razonando las respuestas, el valor de las siguientes expresiones en C:
$$\begin{array}{l} 2/3 \\ 2.0/3 \\ 2/3.0 \end{array}$$
3. Obtener los valores que resultan de cada una de las expresiones del programa de la sección 4.7.2. Comprobarlos ejecutando el programa en el ordenador.
4. Escriba un programa que pida al usuario dos valores enteros e imprima por pantalla el resultado de la suma, resta, multiplicación, división y resto.
5. Modifique el programa anterior para que los dos valores sean reales. ¿Hay alguna operación que ahora no tiene sentido realizar?

CAPÍTULO 5

Control de flujo: Bucles

5.1. Introducción

Es habitual que los programas realicen tareas repetitivas o iteraciones (repetir las mismas operaciones pero cambiando ligeramente los datos). Esto no supone ningún problema para el programador novato, que después de aprender a cortar y pegar puede repetir varias veces el mismo código, pero dentro de un límite. Por ejemplo, un programa que escriba 3 veces la dirección de la universidad para imprimir unos sobres de cartas sería el siguiente:

```
/******  
Programa: Sobres.c  
Descripción: Escribe la dirección de la Universidad en tres sobres  
Revisión 0.0: 10/MAR/1998  
Autor: Rafael Palacios  
*****/  
  
#include <stdio.h>  
  
int main(void)  
{  
    printf("Universidad Pontificia Comillas\n");  
    printf("c/Alberto Aguilera, 23\n");  
    printf("E-28015 Madrid\n");  
  
    printf("Universidad Pontificia Comillas\n");  
    printf("c/Alberto Aguilera, 23\n");  
    printf("E-28015 Madrid\n");  
  
    printf("Universidad Pontificia Comillas\n");  
    printf("c/Alberto Aguilera, 23\n");  
    printf("E-28015 Madrid\n");  
  
    return 0;  
}
```

Este programa es fácil de escribir, a base de *copy* y *paste*, cuando sólo se quieren imprimir tres sobres, pero ¿qué ocurre si queremos imprimir sobres para todos los alumnos de la universidad?

Todos los programas que se han visto hasta ahora tienen un flujo continuo desde arriba hasta abajo, es decir las instrucciones se ejecutan empezando en la primera línea y descendiendo hasta la última. Esto obliga a copiar parte del código cuando se quiere que el programa repita una determinada tarea. Sin embargo el lenguaje C, como cualquier otro lenguaje de programación, proporciona métodos para modificar el flujo del programa permitiendo pasar varias veces por un conjunto de instrucciones.

Se llama **bucle** de un programa a un conjunto de instrucciones que se repite varias veces. Es decir, es una zona del programa en la que el flujo deja de ser descendente y vuelve hacia atrás para repetir la ejecución de una serie de instrucciones.

El bucle más sencillo es el bucle **for**, que generalmente se utiliza para repetir parte del código un número predeterminado de veces. Además existen otros dos tipos de bucles más: el bucle **while** y el bucle **do-while** que se usan para repetir una serie de instrucciones mientras una condición es cierta. La elección entre un tipo u otro de bucle puede parecer complicada en un principio, pero como verá más adelante, con un poco de experiencia no tendrá ningún problema en elegir el tipo de bucle más adecuado al problema a resolver.

5.2. Sintaxis del bucle for

El bucle **for** queda definido por tres argumentos: *sentencia_inicial*, *condición* de repetición y *sentencia_final* de bucle. Estos argumentos se escriben separados por punto y coma y no por coma como en las funciones. Así, la sintaxis del bucle **for** es:

```
for(sentencia_inicial; condición; sentencia_final ){
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
}
```

Lo mejor es ver un ejemplo:

```
#include <stdio.h>
2
int main(void)
4 {
    int i;
6
    for(i=0; i<3; i++) {
8        printf("Universidad Pontificia Comillas\n");
        printf("c/Alberto Aguilera, 23\n");
10       printf("E-28015 Madrid\n");
    }
12    return 0;
}
```

Este programa hace lo mismo que el programa anterior, pero podemos escribir el número de sobres que queramos simplemente cambiando el 3 de la línea 7 por otro número. En este ejemplo la instrucción `i=0` es la sentencia inicial, que se ejecuta antes de entrar en el bucle. La condición es `i<3`, que indica que el bucle se va a repetir mientras sea cierto que la variable `i` tiene un valor menor que 3. Como se acaba de

asignar el valor 0 a la variable `i`, inicialmente se cumple la condición del bucle, por lo tanto el flujo del programa entra en el bucle y se ejecutan todas las sentencias comprendidas entre las dos llaves (`{` en la línea 7, y `}` en la línea 11). Al alcanzar la línea 11 se ejecuta la sentencia final de bucle, en este caso `i++` y luego se vuelve a comprobar la condición. En la primera iteración la variable `i` vale 0, pero al llegar al final del bucle la instrucción `i++` hace que pase a valer 1, entonces se vuelve a comprobar la condición `i<3` que se cumple y por lo tanto el flujo vuelve a la línea 8.

Este bucle se ejecuta 3 veces en total, durante la primera pasada `i` vale 0, al llegar a la línea 11 pasa a valer 1 y hace la segunda pasada, entonces pasa a valer 2 y hace la tercera. Al terminar la tercera pasada, la instrucción `i++` hace que `i` valga 3 y por lo tanto deja de cumplirse la condición `i<3` ya que 3 no es menor que 3 (es igual). Como puede apreciarse, la variable `i` controla el número de veces que se ejecuta el bucle. Por ello a este tipo de variables se les denomina **variables de control**.

Es aconsejable empezar los bucles **for** con instrucciones del tipo `i=0` (en lugar de `i=1`) y poner condiciones del tipo `i<3` (en lugar de `i<=3`). La razón se expondrá en el capítulo 8.

Es muy importante hacer un buen uso del sangrado para facilitar la lectura del programa. En este ejemplo puede observarse que todas las instrucciones del programa tienen un sangrado de tres espacios, pero las instrucciones del bucle (líneas 8 a 10) tienen un sangrado adicional de otros tres espacios. De esta manera se ve claramente que la llave de la línea 11 cierra el bucle **for** de la línea 7 mientras que la llave de la línea 13 cierra la función `main`, haciendo pareja con la llave de la línea 4. Puesto que el compilador ignora todos los espacios y los cambios de línea, cada programador puede elegir su estilo propio, lo importante es ser coherente en todo el código. Por ejemplo en lugar de 3 espacios pueden escribirse **siempre 2**, lo que no vale es poner unas veces 3 y otras 2 porque entonces las columnas quedan hechas un churro. Tampoco es correcto imprimir los programas utilizando un tipo de letra proporcionado¹ como “Times” o “Helvética” sino que los programas deben imprimirse en tipos de letra monoespaciados como por ejemplo “Courier”.

Por último destacar las siguientes nociones:

- la sentencia inicial siempre se ejecuta
- la condición siempre se comprueba **antes** de cada iteración. Por lo tanto puede ocurrir que nunca se llegue a entrar en el bucle si desde el principio no se cumple la condición. Por ejemplo `for(i=0; i<-3; i++)` nunca entra en el bucle.
- la sentencia final se ejecuta al terminar cada iteración. Por lo tanto si no se entra en el bucle esta sentencia no se ejecuta nunca.
- Si el bucle está mal construido puede ocurrir que no termine nunca. La sensación es que el ordenador se paraliza. Un ejemplo de este tipo de bucle es:

```
for(i=0; i<3; i--);
```

5.3. Operadores relacionales

Veamos ahora una breve introducción a los operadores relacionales que pueden utilizarse para escribir la condición del bucle.


¹Los tipos de letra proporcionados son aquellos en los que cada letra tiene una anchura diferente, por ejemplo la letra ‘m’ es más ancha que la letra ‘l’, mientras los tipos monoespaciados son aquellos en los que todas las letras y signos tienen la misma anchura.



C99 permite definir variables locales a los bucles. Estas variables sólo tienen efecto dentro del bucle y no se pueden utilizar al salir del mismo. Su utilidad principal es para definir el índice del bucle: `for(int i=0; i<3; i++)`.

Los operadores relaciones son:

- mayor que: >
- menor que: <
- mayor o igual a: >=
- menor o igual a: <=
- igual a: ==
- distinto de: !=



```

int main(void)
{
    printf("Hola\n");
    return 0;
}

```

5.4. Doble bucle for

```

/*****
Programa: CortaCesped.c
Descripción: Programa para manejar un robot que corta el césped.
Revisión 0.0: 12/MAR/1998
Autor: Rafael Palacios
*****/

```

```
#include <stdio.h>
#include <robot.h> /* Declara las funciones para el manejo
                    del robot*/

int main(void)
{
    int i,j;

    /*Inicialmente el robot está en la esquina
       R*****
       *****
       *****
       *****
       *****
```

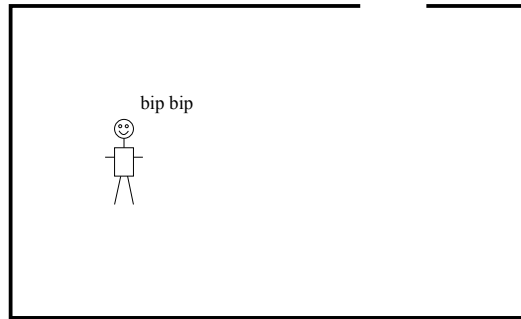


Figura 5.1: Robot

El robot debe ir avanzando hacia la derecha mientras corta el césped y al llegar al final retrocede y baja un metro.

```

.....
...R*****
*****
*****
*****
*/

for(i=0; i<5; i++) {
    ColocarEnPosicion(); /*Se coloca mirando a la derecha*/
    for(j=0; j<20; j++) {
        AvanzaDerecha();
        printf("He avanzado un metro\n");
    }
    printf("He llegado al final, vuelvo\n");
    RegresarIzquierda();
    BajarUnMetro();
}
printf("Trabajo terminado. Esto es fácil.\n");
return 0;
}

```

En caso de bucles anidados es más importante todavía tener cuidado con utilizar un sangrado correcto, de manera que se identifique claramente el comienzo y el final de cada bucle. En este ejemplo también se puede observar que no es lo mismo escribir la instrucción `ColocarEnPosicion()` antes del **for** interno que después de la llamada a `BajarUnMetro()`. Aunque parezca lo mismo, sólo se garantiza que el robot empieza a cortar el césped en la dirección adecuada cuando el programa está escrito como en el ejemplo. Es importante tener en cuenta que las variables de control de dos bucles anidados deben ser distintas.

5.5. Bucle while

Hasta ahora, siempre que hemos tenido que realizar un bucle sabíamos de antemano el número de veces que teníamos que repetirlo. Lamentablemente, la vida no es siempre tan fácil. Imaginemos que tenemos un robot en una habitación como la mostrada en la figura 5.1. El robot sólo obedece a dos instrucciones sencillas: avanzar un centímetro hacia adelante y girar g grados a la derecha. Nuestra tarea es conseguir que el robot salga por la puerta, pero no sabemos a priori donde está, por lo que una solución a base de un bucle **for** que avance el robot hasta la pared, una instrucción de giro de 90 grados, otro bucle para que avance hasta la puerta y un último giro de -90 grados para salir no nos sirve, ya que al no conocer a priori la posición del robot, no sabemos el número de iteraciones que tenemos que realizar en cada bucle para sacar al robot de la habitación sin que peligre su integridad física.

Como futuros ingenieros que somos no nos vamos a rendir fácilmente ante semejante problema. Hablando con el fabricante, nos sugiere que le instalemos al robot su flamante sensor de detección de paredes, como siempre a cambio de un módico precio. Además, el fabricante, en un gesto de amabilidad insuperable, nos ha regalado una serie de funciones en C para que el manejo del robot sea mucho más fácil. Estas funciones son las siguientes:

- `avanza_1_cm_adelante()` que hace avanzar al robot un centímetro hacia adelante.
- `gira(double angulo)` que es una función que admite un **double** como parámetro y que hace que el robot gire a derechas el número de grados indicado en dicho parámetro.
- `toca_pared()` que es una función que vale uno si el robot toca la pared y cero si no la toca.

Por tanto, una vez instalado el sensor, la solución al problema es muy simple: basta con hacer avanzar al robot hasta que toque la pared, girar 90 grados, y seguir avanzando pegado a la pared, lo que seguirá siendo detectado por nuestro estupendo sensor, hasta que lleguemos a la puerta, momento en el cual giraremos -90 grados para salir, cumpliendo con éxito nuestro objetivo. El algoritmo descrito se puede expresar usando **pseudocódigo**² como:

```
mientras que el robot no toque la pared:
    avanza_1_cm_adelante(); /* Avanza hacia la pared */

gira(90);
mientras que el robot toque la pared:
    avanza_1_cm_adelante(); /* Avanza pegado a la pared hacia
                             la puerta*/

girar(-90); /* Se pone delante de la puerta */
avanza_1_cm_adelante(); /* Sale por la puerta! */
```

Como podemos apreciar, el algoritmo consiste en repetir una serie de instrucciones **mientras** una condición sea cierta. Para ello todos los lenguajes estructurados

²Un pseudocódigo es una manera de expresar un algoritmo usando lenguaje natural y a un nivel de detalle muy alto.

poseen unas sentencias de control específicas, que el caso del C son los bucles **while** y **do-while**.

5.5.1. Sintaxis del bucle *while*

Este tipo de bucles repiten una serie de instrucciones mientras una condición sea cierta. La sintaxis es:

```
while(condición){
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
}
```

El funcionamiento del bucle es como sigue: en primer lugar se evalúa la expresión *condición*. Si el resultado es falso no se ejecutará ninguna de las instrucciones del bucle, el cual está delimitado, al igual que en el caso del bucle **for** por dos llaves ({ y }). Si por el contrario la condición es cierta, se ejecutarán todas las instrucciones del bucle. Después de ejecutar la última instrucción del bucle (*instrucción_n*;) se vuelve a comprobar la *condición* y se terminará el bucle si es falsa o se realizará otra iteración si es cierta, y así sucesivamente. Cuando el bucle termina, la ejecución continuará después de la llave }.

5.5.2. Ejemplos

Veamos a continuación algunos ejemplos del uso del bucle **while**.

Sacar al robot de la habitación

Se desea resolver en C el problema propuesto en la introducción, para lo cual disponemos de la librería de funciones proporcionadas por el fabricante del robot discutidas en sección 5.5. Como ya hemos realizado un pseudocódigo, la escritura del programa en C es algo inmediato:

```
/* Programa: SacarRob
 *
 * Descripción: Sacar al robot de una habitación como la mostrada
 *              en la figura 5.1.
 *
 * Revisión 0.0: 10/03/1998
 *
 * Autor: El robotijero loco.
 */

#include <stdio.h>
#include <robot.h> /* Declara las funciones del robot */

int main(void)
{
    while(toca_pared() != 1){
```

```

    avanza_1_cm_adelante(); /* Avanza hacia la pared */
}

girar(90);

while(toca_pared() == 1){
    avanza_1_cm_adelante(); /* Avanza pegado a la pared hacia
                             la puerta*/
}

girar(-90);    /* Se pone delante de la puerta */

avanza_1_cm_adelante(); /* Sale por la puerta! */
return 0;
}

```

Suma de series.

A nuestro hermanito pequeño le han mandado en la escuela la ardua tarea realizar las sumas siguientes:

- $1 + 27 + 34 + 54 =$
- $2 + 34 + 356 + 1234 + 34 + 2378 + 3 =$
- $23 + 2345 =$

Como nuestro hermanito pequeño no tiene aún calculadora por haberse gastado el dinero en chicles, nos pide que le dejemos la nuestra. El problema es que se acaban de gastar las pilas y es domingo por la tarde, con lo que sólo tenemos dos opciones: hacer las sumas a mano o realizar un pequeño programa en C, lo cual parece ser lo más razonable, dado que existen fundadas expectativas de que el próximo día de clase el profesor de matemáticas vuelva a atormentar a nuestro hermanito con más sumas enormes.

Como buenos programadores que somos ya a estas alturas, antes de encender el ordenador tomamos papel y lápiz para analizar el problema, el cual no es más que sumar una serie de números. Lo primero que se nos ocurre es realizar un programa que realice las sumas propuestas: solución fácil, pero poco apropiada, pues sólo es válida para este ejercicio y no para los que se avecinan. Lo siguiente que se nos ocurre es realizar un bucle, que vaya pidiendo los números por el teclado y los vaya sumando. El problema ahora es que el número de sumandos varía de una suma a otra. Una solución consiste en pedir al usuario, al principio del programa, el número de sumandos y con esta información usar un bucle **for**; pero el inconveniente de esta solución es que hay que contar antes todos los números. La segunda solución es usar un bucle **while** y decirle al usuario que después del último número introduzca un cero para indicar al programa el final de la serie. Un posible pseudocódigo sería:

```

pedir número;
mientras número no sea cero {
    sumar el número a la suma anterior;
    pedir número;
}
imprimir total;

```

De este pseudocódigo lo único que nos queda por resolver es cómo hacer para sumar el número a la suma anterior. Este tipo de operaciones son muy comunes en C y se resuelven creando una variable, **inicializándola a cero antes de entrar en el bucle** y dentro del bucle se incrementa con el valor deseado, es decir:

```
int suma;

suma = 0; /* Inicialización */

bucle{ /* bucle for, while... */
    suma += lo_que_sea;
}
continuar el programa;
```

En este pseudocódigo primero se define la variable suma como entera (obviamente el tipo en cada caso será el que haga falta). Después hay que inicializarla, pues dentro del bucle se hace suma += lo_que_sea que como recordará es una manera abreviada de expresar suma = suma + lo_que_sea y por tanto si la variable suma no se inicializa antes de entrar en el bucle, el valor de dicha variable puede ser cualquier cosa, con lo que el valor de la suma final será erróneo y el suspenso de nuestro hermanito cargará para siempre sobre nuestras conciencias.

Una vez resueltos todos los problemas de nuestro programa podemos pasar a escribirlo en C, quedando como:

```
/* Programa: SumaSer
2  *
   * Descripción: Suma una serie de números introducidos por teclado.
4  *               La serie se termina introduciendo un cero, momento
   *               en el que se imprime el total de la suma.
6  *
   * Revisión 0.0: 10/03/1998
8  *
   * Autor: El hermano del hermanito sin calculadora.
10 */

12 #include <stdio.h>

14 int main(void)
   {
16     int suma; /* Valor de la suma */
       int numero; /* número introducido por teclado */
18
       printf("Introduzca los números a sumar. Para terminar"
20           " teclee un 0\n");

22     suma = 0; /* inicializo el total de la suma */

24     scanf("%d", &numero); /* pido el primer número */
       while(numero != 0){
26         suma += numero;
           scanf("%d", &numero);
28     }
```

```

30  printf("El valor de la suma es: %d\n", suma);
    return 0;
32 }

```

5.6. Bucle do-while.

En la sección anterior se ha visto que el bucle **while** comprueba su condición antes de ejecutar el bucle. Esto significa que si la condición no se cumple inicialmente, entonces el bucle no se ejecuta ninguna vez. Para los casos en que queremos garantizar que el bucle se ejecute al menos una vez existe el bucle **do-while**, el cual hace una primera pasada por el bucle y comprueba su condición al final, terminando el bucle si ésta es falsa, o volviéndolo a ejecutar si es cierta. La sintaxis es:

```

do{
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
}while(condición);

```

En este caso se ejecutarán las instrucciones *instrucción_1...instrucción_n* y después se evaluará la *condición*. Si es falsa se continúa con la instrucción que sigue al bucle y si es cierta se vuelve a repetir el bucle (*instrucción_1...instrucción_n*), se evalúa la *condición* y así sucesivamente.

5.6.1. Ejemplos

Como el movimiento se demuestra andando, vamos a ver un ejemplo del bucle **do-while**.

Suma de series. Versión 1.0

Vamos a resolver el problema de la suma de series anterior usando un bucle **do-while**. En este caso, después de la fase de análisis que se realizó en dicha sección llegamos a que otro posible pseudocódigo que soluciona el mismo problema es:

```

hacer{
    pedir número;
    sumar el número a la suma anterior;
}mientras número no sea cero;
imprimir total;

```

Que expresado en C resulta:

```

1  /* Programa: SumaSer
   *
3  * Descripción: Suma una serie de números introducidos por teclado.
   *               La serie se termina introduciendo un cero, momento
5  *               en el que se imprime el total de la suma.
   *
7  * Revisión 1.0: 10/03/1998

```

```

*
9  * Autor: El hermano del hermanito sin calculadora.
  */
11
12 #include <stdio.h>
13
14 int main(void)
15 {
16     int suma;    /* Valor de la suma */
17     int numero; /* número introducido por teclado */
18
19     printf("Introduzca los números a sumar. Para terminar"
20           " teclee un 0\n");
21
22     suma = 0; /* inicializo el total de la suma */
23
24     do{
25         scanf("%d", &numero);
26         suma += numero;
27     }while(numero != 0);
28
29     printf("El valor de la suma es: %d\n", suma);
30     return 0;
31 }

```

Comparando esta solución con la anterior usando un bucle **while**, vemos que la única diferencia es que ahora no hace falta pedir el primer número antes de comenzar el bucle, pues éste se ejecuta al menos una vez, comprobándose la condición al final. Por tanto, es posible leer el número dentro del bucle y comprobar si es cero al final de éste.

Otra cosa que cambia es el orden de las dos instrucciones que hay dentro del bucle. Se deja como ejercicio la explicación del porqué de semejante cambio.

5.7. ¿Es el bucle **for** igual que el **while**?

Algún lector aventajado se habrá dado cuenta de que el bucle **while** y el **for** son muy parecidos. En realidad cualquier cosa que se realice con uno de ellos se puede realizar con el otro. Así un bucle **while** genérico como:

```

while(condición){
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
}

```

Se realizaría con un **for** de la siguiente manera:

```

for( ; condición; ){
    instrucción_1;
    instrucción_2;
}

```



```

    ...
    instrucción_n;
}

```

En donde se aprecia que no se ha puesto nada en la condición inicial que como recordarán iba entre el primer paréntesis del **for** y el primer **;** ni en la condición final que iba después del último **;** y antes del paréntesis que cierra el **for**. El compilador en este caso no da ningún error, sino que simplemente no hace nada antes de empezar el bucle ni tampoco hace nada al final de cada iteración. Por tanto el funcionamiento de este bucle **for** es el siguiente: al principio evalúa la *condición*, si es falsa se continúa después del bucle y si es cierta se ejecuta el bucle, volviéndose a comprobar la *condición* y repitiéndose el ciclo. Dicho comportamiento como podrá apreciar mi querido lector es el explicado no hace muchas líneas atrás cuando se intentaba exponer el funcionamiento del bucle **while**.

Del mismo modo un bucle **for** se puede expresar mediante un **while**, aunque no tan elegantemente. Así el bucle **for**:

```

for(sentencia_inicial; condición; sentencia_final ){
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
}

```

mediante un bucle **while** se expresaría como:

```

sentencia_inicial;
while(condición){
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
    sentencia_final;
}

```

5.7.1. Entonces ¿cuál elijo?

La elección entre un tipo de bucle u otro debe hacerse intentando conseguir la mayor claridad posible del código escrito:

- Si se desea repetir un conjunto de instrucciones **mientras** una condición sea cierta, entonces lo más natural es usar un bucle **while** o **do-while**,³ pues muestra mucho más claramente lo que se desea hacer.
- Si en cambio **sabemos de antemano el número de veces** que ha de repetirse una parte del código, digamos *x* veces, entonces es más cómodo usar un bucle **for**. Normalmente dicho bucle será del tipo:

³La elección entre **while** o **do-while** dependerá si la condición del bucle ha de evaluarse al principio o al final del bucle.

```

for(i=0; i<x; i++){
    /* Código a repetir */
}

```

5.8. El índice del bucle for sirve para algo

En los ejemplos dados hasta ahora, la variable que permitía llevar la cuenta de las veces que ejecutábamos el bucle **for** se usaba única y exclusivamente para eso. Sin embargo, al ser una variable normal y corriente, es posible usarla dentro del bucle o incluso a la salida del bucle. Eso sí, no es muy recomendable cambiar su valor dentro del bucle, pues se generan programas confusos y por tanto difíciles de mantener. Veamos un ejemplo para aclarar este tema: supongamos que acabamos de llegar a casa después de una clase de matemáticas en la que nos han explicado las series aritméticas y nos han dado la fórmula:

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

El problema es que como estábamos hablando con el compañero (como de costumbre), justo cuando terminábamos de copiar la fórmula el profesor la borró de la pizarra, con lo que no estamos muy seguros de ella. Como ya estamos hechos unos artistas del C, decidimos hacer un programilla que calcule el valor de una serie de n números y comprobar así la fidelidad de nuestros apuntes. El pseudocódigo del programa es muy simple:

```

pedir valor final n;
inicializar suma total;
for(i=1; i<=n; i++){
    sumar i a la suma total;
}
imprimir suma total;
calcular el valor de la serie e imprimirla;

```

Una vez comprobado que el pseudocódigo realiza lo que queremos pasamos a la acción, codificando el siguiente programa:

```

1  /* Programa: SerAri
   *
3  * Descripción: Suma una serie aritmética de 1 a n, siendo n un
   *               dato introducido por el usuario, imprimiendo el
5  *               valor de la suma y el de la formula n*(n+1)/2.
   *
7  * Revisión 0.0: 10/03/1998
   *
9  * Autor: El alumno charlatán.
   */
11
12 #include <stdio.h>
13
14 int main(void)
15 {
    int suma;    /* Valor de la suma */

```

```
17  int numero; /* Valor final de la serie */
    int i;      /* índice del for */
19
    printf("Introduzca el número final de la serie.\n");
21  scanf("%d", &numero);

23  suma = 0;   /* inicializo el total de la suma */

25  for(i=1; i<=numero; i++){
    suma += i;
27  }

29  printf("El valor de la suma es: %d\n", suma);
    printf("Y el de la fórmula es: %d\n", numero*(numero+1)/2 );
31  return 0;
}
```

En este programa se han introducido dos novedades: la primera ha sido la anunciada previamente de usar la variable de control del bucle `i` dentro del bucle (línea 26). La segunda es la de usar una expresión dentro del `printf` en la línea 30. El funcionamiento de esta instrucción es el siguiente: se evalúa en primer lugar la expresión `numero*(numero+1)/2` y el valor resultante es el que se imprime en pantalla.

5.9. Recomendaciones y advertencias

- Es importante acostumbrarse desde el principio a utilizar el tipo de bucle adecuado. Normalmente se utiliza **for** cuando se conoce a priori el número de veces que se debe repetir el bucle. Cuando el número de iteraciones es desconocido se utiliza **while** o **do-while** dependiendo de si la condición se verifica antes de empezar o después de la primera iteración.
- Aunque en este tema sólo se han anidado dos bucles **for**, nada impide realizar anidaciones de distintos tipos de bucle, por ejemplo un **while** dentro de un **for**; o anidar más de dos bucles.
- Para garantizar que los bucles **while** puedan terminar, hay que verificar que las variables que intervienen en la condición puedan cambiar de valor dentro del bucle.
- No hay que confundir el operador relacional de igualdad `'=='` con el operador de asignación `'='`.
- Siempre que se realicen operaciones del tipo acumulador (como por ejemplo: `suma += algo;`) es necesario inicializar la variable que hace de acumulador. Esta inicialización ha de realizarse justo antes de empezar el bucle, para así evitar errores y conseguir una mayor claridad.

5.10. Resumen

En este capítulo se ha mostrado la manera de repetir una serie de instrucciones en un programa, o como llaman los entendidos, la manera de escribir **bucles**. Se ha visto que existen dos tipos de bucles: el bucle **for** que permite repetir un bucle un

numero predeterminado de veces y el bucle **while** que repite una serie de instrucciones mientras una condición sea cierta. Además el bucle **while** tiene dos variantes: el **while** que evalúa la condición al principio del bucle y el **do-while**, que la evalúa al final. Por tanto el bucle **do-while** se ejecuta al menos una vez mientras que el **while** puede no ejecutarse nunca si su condición es falsa antes de comenzar el bucle.

5.11. Ejercicios

1. Comprobar que otra forma de escribir el bucle del ejemplo mostrado en la sección 5.2 sería poner **for**(i=0; i!=3; i++).
2. Escribir un programa que presente en pantalla con **printf** todos los números del 0 al 20.
3. Modificar el programa anterior para que escriba sólo los números pares del 2 al 20.
4. Realizar un programa para calcular multiplicaciones del tipo:
 - $2 * 3 * 35$
 - $345 * 34 * 3.5 * 23 * 2.1$
 - $2 * 2.234$

Hacer dos versiones: una en la que se pida al usuario al principio del programa la cantidad de números que desea multiplicar para pasar seguidamente a pedirle cada uno de los números a multiplicar. En la otra versión el usuario introducirá uno a uno los números, indicándole al programa que ya no hay más mediante la introducción de un 1.

5. Realizar un programa para comprobar la veracidad de la siguiente fórmula para calcular la suma de una serie geométrica:

$$1 + 3 + 5 + \dots + (2 * n - 1) = n^2$$

Pistas: La instrucción final de los bucles **for** puede ser también **i+=2**. Para calcular n^2 se puede hacer **n*n**.

6. Escribir un programa que calcule la media aritmética de una serie de números positivos. El programa pedirá los números al usuario hasta que éste introduzca un número negativo. En ese momento se imprimirá la media aritmética y se terminará la ejecución.
7. Escriba un programa que pida un número **n** al usuario y calcule su factorial.

CAPÍTULO 6

Control de flujo: if y switch-case

6.1. Introducción

Es habitual que los programas tengan que realizar tareas diferentes en función del valor de determinadas variables. El lenguaje C dispone de dos construcciones para ello: **if** y **switch-case**. La primera permite ejecutar bloques de código en función del valor de una condición lógica, mientras que la segunda permite ejecutar una serie de bloques de código en función del valor que tome una variable entera.

6.2. El bloque if. Introducción

La instrucción **if** permite definir un bloque de código que sólo se ejecutará si una condición lógica es cierta o definir dos bloques, uno que se ejecutará cuando dicha condición lógica sea cierta y otro cuando sea falsa.

6.3. Sintaxis del bloque if

Existen dos formatos básicos para definir instrucciones que se ejecutan de manera condicional. Un bloque que se puede ejecutar o no (**if** normal), y dos bloques que se ejecutan uno u otro (bloque **if-else**). El formato en cada caso es el siguiente:

- **if** normal:

```
if (condición) {  
    instrucción_1;  
    instrucción_2;  
    ...  
    instrucción_n;  
}
```

- bloque **if-else**:

```
if (condición) {  
    instrucción_1.1;  
    instrucción_1.2;  
    ...  
    instrucción_1.n;  
} else {  
    instrucción_2.1;  
    instrucción_2.2;  
    ...  
}
```

```

        instrucción_2.m;
    }

```

Veamos como ejemplo un programa que pregunta el valor de dos números enteros y que escribe en pantalla cuál es el mayor de los dos. Este programa debe utilizar `scanf` para preguntar los valores y luego un `if` para decidir si se escribe el primero o el segundo.

```

/*****
Programa: Compara.c
Descripción: Lee dos números del teclado y los compara.
Revisión 0.0: 16/FEB/1999
Autor: Rafael Palacios
*****/

```

```

#include <stdio.h>

int main(void)
{
    int a,b;

    printf("Dame un valor entero ");
    scanf("%d",&a);
    printf("Dame otro valor ");
    scanf("%d",&b);

    if (a>b){
        printf("El mayor es %d\n",a);
    } else {
        printf("El mayor es %d\n",b);
    }
    return 0;
}

```

Otra aplicación muy interesante es evitar errores de cálculo en los programas, como por ejemplo las divisiones por cero o las raíces cuadradas de números negativos. El siguiente ejemplo es un programa que incluye un control para evitar las divisiones por cero. En realidad el ejemplo parece algo tonto porque en caso de error simplemente termina con un `exit(1)`, pero un programa más completo puede tomar otras medidas como por ejemplo volver a preguntar el valor del denominador.

La función `exit` hace que el programa termine inmediatamente y le devuelve al sistema operativo el valor de su argumento.¹ Esta manera de terminar un programa antes de que llegue la ejecución al final de la función `main`, **sólo** está permitida en caso de detectarse un error grave y realmente excepcional, que impida continuar con la ejecución normal del programa.

```

/*****
Programa: Division.c
Descripción: Calcula el cociente de dos números.

```

¹Por convenio, un programa que termina a causa de un error debe devolver al sistema operativo un valor distinto de cero.

El programa incluye un control que evita dividir por cero.
 Revisión 0.0: 16/FEB/1999
 Autor: Rafael Palacios
 *****/

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

int main(void)
{
    int numerador, denominador;
    double division;

    printf("Numerador? ");
    scanf("%d",&numerador);
    printf("Denominador? ");
    scanf("%d",&denominador);

    if (denominador==0) {
        printf("Error, el resultado es infinito\n");
        exit(1);
    }
    division=(double)numerador/denominador;
    printf("La división vale: %f\n",division);
    return 0;
}
```

6.4. Formato de las condiciones

Las condiciones que aparecen en los bloques **if** son las mismas que pueden aparecer en los bucles **while**, **do-while** y en el segundo término de los bucles **for**. Son expresiones que normalmente incluyen operadores lógicos y relacionales. Este apartado resume estos operadores, aunque muchos de ellos ya se han visto en capítulos anteriores.

6.4.1. Operadores relacionales

Son operadores que permite comparar variables y/o constantes entre si.
 Los operadores relaciones son:

- mayor que: >
- menor que: <
- mayor o igual a: >=
- menor o igual a: <=
- igual a: ==
- distinto de: !=

Las reglas de precedencia establecen que los operadores $>$, $>=$, $<$, $<=$ se evalúan en primer lugar y posteriormente se evalúan los operadores $==$ y $!=$. Esta regla no es muy importante ya que normalmente no se mezclan estos operadores en la misma condición. Más interesante es saber que la precedencia de los operadores matemáticos es mayor que la de los operadores lógicos y por lo tanto condiciones como la siguiente se evalúan realizando primero los cálculos y luego las comparaciones:

```
(a > 3*b)      /* Esto hace ( a > (3*b) ) */
(a+b == 0)    /* Esto hace ( (a+b) == 0 ) */
```

6.4.2. Operadores lógicos

Son operadores que permiten relacionar varias expresiones. Las operaciones básicas son AND (la condición es cierta sólo si las dos expresiones son ciertas), OR (la condición es cierta si una o las dos expresiones son ciertas) y la negación (invierte el resultado lógico de la expresión que le sigue).

Los operadores lógicos son:

- AND lógico: `&&`
- OR lógico: `||`
- negación: `!`

Estos operadores permiten definir condiciones complicadas en todos los bucles y en los `if`. Hay que tener en cuenta que la precedencia del operador `&&` es mayor que la del operador `||`.

```
(a>3 && b==4 || c!=7) /* Esto hace ( (a>3 && b==4) || c!=7 ) */
```

En caso de duda lo mejor es poner los paréntesis necesarios antes que perder el tiempo haciendo pruebas en el programa. Es decir, o se mira la documentación y se hacen las cosas bien a la primera o se ponen paréntesis hasta que la operación quede clara. Esto último tiene la ventaja de que estará claro para todo el que lea el programa, aunque no se acuerde muy bien de las precedencias de los operadores.

Por último conviene destacar que en C los operadores relacionales son operadores binarios, es decir, realizan la comparación entre los operandos que están a su derecha y a su izquierda; siendo el resultado de dicha comparación un 1 si la condición es cierta o un 0 si es falsa. Si, acostumbrados a la notación matemática, se escribe:

```
0 <= a <= 10
```

para expresar la condición $0 \leq a \leq 10$, el resultado de dicha expresión será siempre 1, independientemente del valor de la variable a . ¿Puede adivinar por qué? Si se tiene en cuenta que en C las expresiones se evalúan de izquierda a derecha, en primer lugar se evaluará el primer operador `<=`, el cual compara el operando que está a su izquierda (0) con el que está a su derecha (a). El resultado de esta expresión será 0 si a es menor que 0 o 1 en caso contrario. Dicho resultado será el operando izquierdo del siguiente operador relacional, con lo que se evaluará la expresión `0 <= 10` o bien `1 <= 10` según haya sido el resultado de la comparación anterior. En cualquiera de los dos casos, el resultado final de la expresión será cierto (1).

Para expresar en C la condición $0 \leq a \leq 10$ correctamente, es necesario dividir la relación en dos, es decir, en primer lugar comprobar si a es mayor o igual que 0 y en segundo lugar si a es menor o igual que 10. Por último es necesario usar un operador

lógico para relacionar ambas expresiones, de forma que el resultado sea cierto cuando **ambas** condiciones sean ciertas. ¿Se atreve a escribir la condición? Por si acaso no se ha atrevido, ésta se muestra a continuación:

```
(0 <= a && a <= 10)
```

6.5. Valores de verdadero y falso

Todas las operaciones lógicas y relacionales devuelven un valor tipo **int** que puede ser 1 para **verdadero** o 0 para **falso**. Por lo tanto las condiciones que se escriben en los bucles o en los **if** suelen tener el valor 1 ó 0. Sin embargo las condiciones se consideran falsas sólo cuando valen 0 y verdaderas cuando valen cualquier otra cosa. En este sentido cualquier variable de tipo entero puede utilizarse como condición y hará que la condición sea falsa sólo si su valor es cero. Esta es la razón fundamental para que en C no sea necesario un tipo de variable para operaciones lógicas (que sólo pueda valer 1 ó 0) como ocurre en otros lenguajes de programación. En C la siguiente línea es válida:

```
i=a>3;
```

En esta instrucción la variable *i* toma el valor 1 ó 0 en función del valor que tenga la variable *a*. Además las siguientes condiciones son equivalentes para comprobar si una variable entera vale **verdadero**:

```
if (a!=0) {
    printf("Verdadero!\n");
}
if (a) {
    printf("Verdadero!\n");
}
```

Lo mismo ocurre para comprobar si una variable es cero o **falso**, se puede preguntar de dos maneras:

```
if (a==0) {
    printf("Falso\n");
}
if (!a) {
    printf("Falso\n");
}
```

En definitiva, la única definición categórica es que 0 es **falso**.

En ambos casos queda más clara la primera expresión lógica (**if(a!=0)**) que la segunda (**if(a)**).

6.6. Bloque if else-if

Es muy habitual en los programas tener que realizar varias preguntas en cadena, de manera que la estructura del bloque **if-else** se puede complicar bastante. Veamos el siguiente ejemplo:

```
/******
Programa: Alturas1.c
Descripción: Pregunta la altura de una persona.
```



Aunque en C se trabaja normalmente con variables **int**, en C99 existen también variables lógicas **_Bool** que sólo pueden tomar los valores 1 ó 0. Además en C99 existen los valores **true** (1) y **false** (0).

Revisión 0.0: 16/FEB/1999

Autor: Rafael Palacios

******/*

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

```
int main(void)
```

```
{
```

```
    double altura;
```

```
    printf("¿Cuál es su altura? ");
```

```
    scanf("%lf",&altura);
```

```
    if (altura<1.5) {
```

```
        printf("Usted es bajito\n");
```

```
    } else {
```

```
        if (altura<1.7) {
```

```
            printf("Usted no es alto\n");
```

```
        } else {
```

```
            if (altura<1.9) {
```

```
                printf("Usted es alto\n");
```

```
            } else {
```

```
                printf("¿Juega al baloncesto?\n");
```

```
            }
```

```
        }
```

```
    }
```

```
    printf("Fin del programa.\n");
```

```
    return 0;
```

```
}
```

En este tipo de estructuras de programa hay que ser muy cuidadoso para colocar correctamente las llaves y es imprescindible hacer un buen uso del sangrado para facilitar la lectura del programa. Únicamente remarcar que el siguiente programa aunque es mucho más compacto **no** hace lo mismo que el anterior:

```
/******
```

```
Programa: Alturas2.c
```

```
Descripción: Pregunta la altura de una persona.
```

```
Revisión 1.0: 16/FEB/1999
```

```
Autor: Rafael Palacios
```

```
*****/
```

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

```
int main(void)
```

```
{
```

```
    double altura;
```

```
    printf("¿Cuál es su altura? ");
```

```
    scanf("%lf",&altura);
```

```

if (altura<1.5){
    printf("Usted es bajito\n");
}
if (altura<1.7){
    printf("Usted no es alto\n");
}
if (altura<1.9){
    printf("Usted es alto\n");
} else {
    printf("¿Juega al baloncesto?\n");
}
printf("Fin del programa.\n");
return 0;
}

```

Este programa es incorrecto porque por ejemplo para alturas de 1.4 m se escriben tres mensajes en lugar de uno, ya que este valor cumple las tres condiciones de los tres **if**. En el caso del primer programa, al ser cierto el primer **if**, se escribe Es usted bajito y se salta directamente al último **printf** del programa porque el resto de las instrucciones forman parte del **else** del primer **if**.

Para este tipo de situaciones se puede usar el bloque **if else-if**, cuya sintaxis es:

```

if (condición) {
    instrucción_1.1;
    instrucción_1.2;
    ...
    instrucción_1.m;
} else if (condición) {
    instrucción_2.1;
    instrucción_2.2;
    ...
    instrucción_2.n;
...
} else {
    instrucción_3.1;
    instrucción_3.2;
    ...
    instrucción_3.p;
}

```

El funcionamiento de esta construcción es el siguiente: Se evalúa la *condición* del primer **if**. Si es cierta se ejecuta el bloque perteneciente a este **if**. Si es falsa se evalúa la *condición* del **else if** que le sigue, ejecutándose su bloque si dicha *condición* es cierta o pasando al siguiente **else if** si es falsa. Si ninguna de las condiciones es cierta se ejecutará el bloque del último **else** en caso de que exista, pues es opcional su uso.

Usando el bloque **if else-if** el programa de las alturas se puede escribir, de una manera mucho más clara como:

```

/*****
Programa: Alturas3.c

```

Descripción: Pregunta la altura de una persona.

Revisión 0.0: 16/FEB/1999

Autor: Rafael Palacios

******/*

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

```
int main(void)
```

```
{
```

```
    double altura;
```

```
    printf("¿Cuál es su altura? ");
```

```
    scanf("%lf",&altura);
```

```
    if (altura<1.5){
```

```
        printf("Usted es bajito\n");
```

```
    }else if (altura<1.7){
```

```
        printf("Usted no es alto\n");
```

```
    }else if (altura<1.9){
```

```
        printf("Usted es alto\n");
```

```
    }else{
```

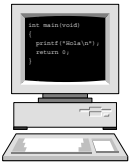
```
        printf("¿Juega al baloncesto?\n");
```

```
    }
```

```
    printf("Fin del programa.\n");
```

```
    return 0;
```

```
}
```



Realice el ejercicio 1.

6.7. La construcción switch-case. Introducción

En muchos programas es necesario realizar una tarea u otra en función de una variable, ya sea ésta el resultado de un cálculo o un valor introducido por el usuario. Este último caso es muy frecuente en los interfaces con el usuario basados en un menú. Tres ejemplos cotidianos de este tipo de interfaz son los cajeros automáticos, los teléfonos móviles y los menús de inicio de los DVDs o de los juegos de ordenador. En función del tipo de dispositivo el usuario puede contar con un botón por cada opción, como ocurre en los cajeros, o con un mando de movimiento y un botón de selección, como ocurre en los móviles o bien con un teclado donde se selecciona la opción debidamente identificada en el menú. En cualquiera de los tres casos, el programa que gestiona el dispositivo recibe la opción del usuario mediante un valor almacenado en una determinada variable. El programa entonces realizará una tarea u otra en función del valor de dicha variable.

Por ejemplo, un menú para seleccionar el idioma de un videojuego podría ser:

IDIOMA

1 English

2 Español

3 Française

Con lo que ya sabemos de C se podría implantar este menú de la siguiente manera:

```

if(idioma == 1){
    juego_ingles();
} else if(idioma == 2){
    juego_espanol();
} else if(idioma == 3){
    juego_frances();
}

```

No obstante, dado que este tipo de situaciones son muy comunes en todos los programas, casi todos los lenguajes de programación poseen estructuras de control específicas para resolver más fácilmente este tipo de decisiones. En el caso de C esta estructura de control es la construcción **switch-case**.

6.7.1. Sintaxis de la construcción *switch-case*

La sintaxis de esta construcción es:

```

switch(expresión){
case constante_1:
    instrucción_1_1;
    ...
    instrucción_1_n1;
    break;
...
case constante_n:
    instrucción_n_1;
    ...
    instrucción_n_nn;
    break;
default:
    instrucción_d_1;
    ...
    instrucción_d_nd;
    break;
}

```

Antes de comenzar a explicar los entresijos del funcionamiento de esta estructura de control conviene destacar algunos aspectos de su sintaxis:

- La expresión que aparece en la línea de **switch** debe producir un resultado **int**. Típicamente dicha expresión es el nombre de una variable **int** o **char**.
- Después de la palabra clave **case** y de su *constante* el símbolo que sigue son dos puntos (:) y no un punto y coma (;).
- Al final de cada bloque de instrucciones de un **case** se escribe la sentencia **break**.
- Sólo hay dos llaves: la de después del **switch** y la del final. Los bloques de instrucciones pertenecientes a cada uno de los **case** vienen delimitados por el **case** y por el **break** del final.

Una vez advertido esto, pasemos a describir el funcionamiento de esta construcción. En primer lugar se evalúa la *expresión* que sigue al **switch**, con lo que se obtendrá

un valor que ha de ser **entero**. Este valor se compara entonces con la constante entera que sigue al primer **case** (la *constante_1*). Si el resultado de la comparación es falso se prueba suerte con el siguiente **case** y así sucesivamente hasta encontrar algún **case** cuya constante sea igual al resultado de la *expresión*. Si esto ocurre se ejecutarán las instrucciones situadas entre el afortunado **case** y el **break** que finaliza su bloque de instrucciones, continuando después con las instrucciones que sigan a la llave que cierra el **switch**. Si ningún **case** tiene a su lado una constante que coincida con el valor de la *expresión*, entonces se ejecutarán las instrucciones situadas entre el **default** y el último **break**, y luego se continuará con las instrucciones que siguen después del **switch**.

Para aclarar ideas, veamos el siguiente ejemplo de un **switch-case**:

```
switch (a){
case 1:
    printf("a vale uno\n");
    break;
case 2:
    printf("a vale dos\n");
    break;
default:
    printf("a no vale ni uno ni dos\n");
    break;
}
```

Supongamos que *a* vale 1. Como acabamos de decir, en primer lugar se evalúa la expresión que hay entre los paréntesis situados después del **switch**. En este caso la expresión es simplemente una variable, por lo que el resultado de la expresión será el valor de la variable, es decir un 1. El programa compara entonces el valor así obtenido con el que hay a continuación del primer **case**, que como hemos tenido mucha suerte, es también un 1, con lo que se ejecutará el conjunto de instrucciones situadas entre este **case** y el siguiente **break** que en este ejemplo tan simple consta tan solo de la instrucción `printf("a vale uno\n")`.

Conviene hacer notar que se puede escribir un bloque **switch-case** sin el apartado de **default**. Si esto ocurre y se da el desafortunado caso de que ninguno de los **case** contenga el valor de la constante resultante de la evaluación de la *expresión*, entonces la ejecución del programa continúa tranquilamente después de la llave que cierra el **switch-case**. En general conviene poner siempre un **default** para controlar errores inesperados.

6.7.2. Ejemplos

Veamos a continuación algunos ejemplos para ilustrar el uso de la construcción **switch-case**.

El robot con control manual

Imaginemos que tenemos un robot como el descrito en el capítulo 5 y que deseamos realizar un controlador interactivo, es decir, un programa mediante el cual el usuario introduzca una serie de comandos al programa y éste mueva al robot según dichos comandos. Como se recordará, el robot sólo obedecía a dos instrucciones básicas: “avanzar un centímetro hacia adelante” y “girar un determinado número de grados a la derecha”. Además disponía de un avanzado sensor que detectaba si el robot estaba

tocando algún obstáculo. Como también recordará, disponíamos de tres funciones que nos había suministrado muy gentilmente el fabricante del robot para gobernarlo (al robot, no al fabricante, claro está). Estas rutinas también se describieron en el capítulo 5, pero las volvemos a poner a continuación para mayor comodidad del lector:

- `avanza_1_cm_adelante()` hace avanzar al robot un centímetro hacia adelante.
- `gira(double angulo)` es una función que admite un **double** como parámetro y que hace que el robot gire a derechas el número de grados indicado en dicho parámetro.
- `toca_pared()` es una función que vale uno si el robot toca la pared y cero si no la toca.

Para que el robot pueda ser gobernado de forma interactiva, el programa debería pedir al usuario un comando y actuar en consecuencia. Dicho comando podría ser cualquier cosa, aunque lo más cómodo es un número o una letra: por ejemplo 1 significaría avanzar hacia adelante y 2 girar un número de grados (que habrá que preguntarle previamente al usuario). Un posible pseudocódigo del programa sería:

```
pedir comando al usuario;
si es 1:
    avanza_1_cm_adelante();
si es 2:
    pedir número de grados;
    gira(número de grados);
```

El pseudocódigo anterior se puede traducir a C usando un **switch-case** de la siguiente manera:

```
/* Programa: RobMan
2  *
   * Descripción: Controla el robot de forma manual. Para ello espera
4  *               a que el usuario introduzca su elección y ejecuta
   *               entonces la opción solicitada por éste.
6  *
   * Revisión 0.0: 15/03/1998
8  *
   * Autor: El robotijero loco.
10 */

12 #include <stdio.h>
   #include <robot.h>
14
16 int main(void)
17 {
   int comando; /* Comando introducido por el usuario */
18   float grados; /* Grados a girar por el robot */

20   printf("Introduzca la opción: ");
   scanf("%d", &comando);

22   switch(comando){
```



```

24  case 1:
      avanza_1_cm_adelante();
26  break;
      case 2:
28      printf("Introduzca el número de grados que desea girar: ");
      scanf("%f", &grados);
30      gira(grados);
      break;
32  default:
      printf("Error: opción no existente\n");
34      break;
  }
36  printf("Adiós.\n");
  return 0;
38 }

```

Ahora bien, salvo que el usuario tenga muy buena memoria, no se acordará para siempre de los comandos, por lo que cada vez que tenga que ejecutar el programa tendrá que consultar el manual de usuario para ver qué comandos tiene disponibles. Para evitar semejante engorro la solución es bien sencilla: imprimamos al comienzo del programa una lista de los comandos disponibles, junto con una breve descripción de lo que hacen. Haciendo esto, el programa al arrancar podría mostrar por pantalla lo siguiente:

Control manual del robot.

Menú de opciones:

- 1.- Avanzar 1 cm hacia adelante.
- 2.- Girar.

Introduzca opción:

A esto se le conoce con el nombre de **menú de opciones**, dado su parecido con los menús de los restaurantes (aunque presentan el grave inconveniente de que no se pueden comer).

Para conseguir esto, basta con añadir al programa anterior entre sus líneas 19 y 20 las instrucciones:

```

printf(" Control manual del robot.\n\n");
printf("     Menú de opciones:\n");
printf("         1.- Avanzar 1 cm hacia adelante.\n");
printf("         2.- Girar.\n\n");

```

Un ejemplo de sesión con el programa, suponiendo que dicho programa se llama `robman` y se ejecuta desde la línea de comandos, sería:

```

c:\temp\grupo00\robman> robman
Control manual del robot.

```

Menú de opciones:

- 1.- Avanzar 1 cm hacia adelante.
- 2.- Girar.

```

    Introduzca opción: 1
Adiós.
c:\temp\grupo00\robman> robman
    Control manual del robot.

    Menú de opciones:
        1.- Avanzar 1 cm hacia adelante.
        2.- Girar.

```

```

    Introduzca opción: 2
    Cuantos grados desea girar: 27
Adiós.
c:\temp\grupo00\robman>

```

Como podemos apreciar el manejo del robot así es un poco pesado, pues para cada instrucción que deseemos darle hemos de volver a llamar al programa. Para hacer un poco más fácil la vida del conductor del robot, podemos realizar un bucle que englobe a todo el programa anterior. Un posible pseudocódigo de esta solución sería:

```

repetir{
    imprimir menú
    pedir comando al usuario;
    si es 1:
        avanza_1_cm_adelante();
    si es 2:
        pedir número de grados;
        gira(número de grados);
}mientras el usuario quiera;

```

El problema ahora está en cómo descubrir cuando el usuario quiere abandonar el programa. Obviamente la única manera de hacerlo (hasta que no se inventen los ordenadores que lean el pensamiento) es preguntárselo directamente. Para ello nada más fácil que incluir una nueva opción en el menú de forma que cuando se elija se salga del bucle. También es posible hacer preguntas dentro del bucle del tipo ¿Quiere continuar (S/N)? pero entonces el uso del programa se vuelve tan pesado como en el caso anterior.

```

/* Programa: RobMan
*
* Descripción: Controla el robot de forma manual. Para ello espera
*              a que el usuario introduzca su elección y ejecuta
*              entonces la opción solicitada por el éste.
*
* Revisión 0.2: 16/03/1998
*              Se añade la opción de salir del programa.
*
* Revisión 0.1: 16/03/1998
*              Se imprime un menú de opciones y se introduce un
*              bucle sin fin para repetir el programa.
*
* Revisión 0.0: 15/03/1998
*

```

```

* Autor: El robotijero loco.
*/

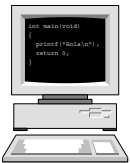
#include <stdio.h>
#include <stdlib.h>
#include "robot.h"

int main(void)
{
    int comando; /* Comando introducido por el usuario */
    float grados; /* Grados a girar por el robot */

    do{
        printf("  Control manual del robot.\n\n");
        printf("  Menú de opciones:\n");
        printf("      1.- Avanzar 1 cm hacia adelante.\n");
        printf("      2.- Girar a derechas.\n");
        printf("      3.- Salir del programa.\n\n");
        printf("Introduzca la opción: ");
        scanf("%d", &comando);

        switch(comando){
            case 1:
                avanza_1_cm_adelante();
                break;
            case 2:
                printf("Introduzca el número de grados que desea girar: ");
                scanf("%f", &grados);
                gira(grados);
                break;
            case 3:
                printf("Que tenga un buen día.\n");
                break;
            default:
                printf("Error: opción no existente\n");
                break;
        }
    }while(comando != 3);
    return 0;
}

```



Realice los ejercicios
2 al 5.

6.8. La sentencia break

Esta sentencia hace que el flujo de programa salga del bloque en el que se encuentre. Este bloque puede ser una construcción **switch-case** tal como acabamos de ver, o cualquiera de los bucles que hemos visto en capítulos anteriores: **for**, **while** o **do-while**. Sin embargo su uso normal se restringe al bloque **switch**.

6.8.1. La sentencia *break* y el *switch-case*

En el caso de la construcción **switch-case** hemos visto que ha de usarse la sentencia **break** explícitamente para indicarle al programa que salga de la construcción. Esto que puede parecer sin sentido, tiene su utilidad en ciertos casos en los que hay que ejecutar el mismo conjunto de instrucciones para varios valores de la expresión que controla el **switch-case**. Imaginemos que tenemos que realizar un programa con un menú para un usuario que es alérgico a los números. Si se da esta situación la única opción que nos queda es usar un menú con letras. En este caso sería deseable que el programa respondiese igual si se introduce la misma letra en minúscula o en mayúscula, lo cual se resuelve fácilmente en C de la siguiente manera:

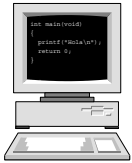
```
switch(letra){
case 'A':
case 'a':
    printf("Opción a del menú\n");
    break;
case 'B':
case 'b':
    printf("Opción b del menú\n");
    break;
default:
    printf("Opción no contemplada\n");
    break;
}
```

En este ejemplo si letra vale 'A' el programa salta a la instrucción que sigue a **case 'A':** y continuará ejecutándose hasta que encuentre un **break** o hasta que llegue el final del **switch**. Por tanto si se da este caso (letra vale 'A') se ejecutará `printf("Opción a del menú\n")` y después se saldrá del **switch**. Conviene destacar que la sentencia **case 'a':** es una etiqueta del **switch-case**, no una instrucción ejecutable.

6.8.2. La sentencia *break* y los bucles

La sentencia **break** puede utilizarse también dentro de los bucles **for**, **while** y **do-while** y tiene como resultado la interrupción inmediata del bucle, continuando la ejecución después de la llave que cierra dicho bucle. Su uso normal es dentro de un **if** que detecte alguna condición especial. En general se considera una mala técnica de programación el utilizar un **break** para salir de los bucles, ya que lo correcto sería incluir la condición de salida que hace ejecutarse el **break** dentro de la condición de control del bucle. A pesar de todo a veces puede ocurrir que tengamos varios **if** anidados, con lo que la condición de salida es realmente complicada y en este caso un **break** nos saca del apuro. Por ejemplo:

```
for(n=0; n<27; n++){
    inst_1;
    inst_2;
    ...
    if(pasa algo){
        inst_3;
        inst_4;
```



Realice el ejercicio 6.

```

    if(pasa otra cosa){
        inst_5;
        if(la cosa se complica){
            inst_6;
            if(la cosa está que arde){
                inst_7;
                if(no podemos continuar){
                    break;
                }
                inst_8;
            }
        }
        inst_9;
    }
    inst_10;
    ...
}

```

En este ejemplo el bucle se repetirá 27 veces, salvo que “pase algo”, luego “pase otra cosa”, entonces “la cosa se complique” para luego descubrir que “la cosa está que arde” y al final nos demos cuenta que “no podemos continuar”, en cuyo caso se ejecutará la sentencia **break** y se saldrá **inmediatamente** del bucle, es decir, no se ejecutarán en esta iteración las instrucciones `inst_8`, `inst_9`, `inst_10` y siguientes. Conviene recalcar que en caso de que se salga del bucle con la instrucción **break** el valor de la variable de control no será 27 como en una ejecución normal del bucle, sino menor.

6.9. La sentencia continue

En ciertos casos no es necesario ejecutar la parte final de una iteración dentro de un bucle cuando se da una situación extraordinaria, situación que se descubre normalmente mediante un **if**. En estos casos se puede utilizar la sentencia **continue** para volver al principio del bucle y comenzar la siguiente iteración, saltándonos lo que nos quedaba de la iteración anterior. Un esquema de este tipo de situaciones es el siguiente:

```

for(n=0; n<27; n++){
    inst_1;
    inst_2;
    ...
    if(condicion_extraordinaria){
        continue;
    }
    inst_3;
    ...
}

```

En este caso siempre que sea cierta la condición extraordinaria no se ejecutarán las instrucciones `inst_3` y siguientes. Hay que destacar sin embargo que sí se incrementará el valor de `n` y se comparará su valor con 27 antes de proseguir con una nueva iteración del bucle.

El uso de la sentencia **continue** genera en la mayoría de los casos programas más difíciles de leer que si se usa un **if** para ejecutar la parte final del bucle sólo cuando **no** sea cierta la condición extraordinaria. Así el programa anterior podría escribirse de una manera mucho más clara como:

```
for(n=0; n<27; n++){
    inst_1;
    inst_2;
    ...
    if(!condicion_extraordinaria){
        inst_3;
        ...
    }
}
```

Como se puede observar el flujo del programa está mucho más claro en este caso, pues la ejecución es siempre desde el principio del bucle hasta el final, sin saltos “raros” por el medio.

Tanto la sentencia **break** como **continue** deben evitarse durante la programación. Se incluye su descripción en este libro porque algunos programas las utilizan, aunque generalmente como consecuencia de haber incluido un parche a una situación no considerada en la fase de diseño. Su uso tan sólo es recomendable en aquellos casos extremos como el ejemplo mostrado en la sección 6.8.2, en el que es necesario salir de una serie de **if** anidados.

6.10. Recomendaciones y advertencias

- La mayoría de los fallos en programas que utilizan estructuras **if** son consecuencia de mala anidación o mala utilización del **else**. Es muy importante cuidar el sangrado del código y realizar un buen diseño, porque verificar todas las posibles combinaciones de valores es muy tedioso.
- En caso de detectar algún problema se recomienda ejecutar el programa paso a paso con un *debugger* verificando que el control va pasando correctamente a las secciones de código apropiadas.
- Se recomienda poner la opción **default** en todas las estructuras **switch**, aunque sólo sea para dar el típico y “teóricamente” absurdo mensaje de “error inesperado”.

6.11. Resumen

Las estructuras de control de flujo **if** y **switch** permiten realizar ejecución condicionada; es decir, que distintas secciones de código sólo se ejecuten si se cumplen determinadas condiciones. A diferencia de los bucles, **if** y **switch** no sirven para repetir un conjunto de instrucciones, sino en todo caso para saltárselas.

La utilización de la estructura del **switch** queda restringida en la práctica a los menús, mientras que el **if** se utiliza con mucha frecuencia.

6.12. Ejercicios

1. Repetir el programa Alturas3.c de la sección 6.6 pero sin utilizar **else**. Es decir, preguntando en el mismo **if** si altura es menor que 1.7 y mayor o igual a 1.5.
Pista: Hay que utilizar los operadores lógicos.
2. Modificar el programa RobMan de la página 67 para usar un bucle **while** en lugar del **do-while**. No obstante tenga en cuenta que el **do-while** es lo más apropiado en este caso.
3. Comprobar el funcionamiento del programa escrito en el ejercicio anterior en un ordenador. Como no disponemos del robot ni de las funciones para manejarlo, sustituir las llamadas a las funciones del robot por **printf** que indiquen la operación de movimiento correspondiente para poder comprobar fácilmente el correcto funcionamiento de nuestro programa.
4. Realizar el programa para controlar el robot de forma interactiva presentado en la sección 6.7.2 pero usando un menú con letras. Tenga en cuenta que será necesario usar variables de tipo **char**
5. Escribir el programa anterior en el ordenador. ¿Funciona correctamente?. En caso de que no funcione correctamente ¿Qué funcionamiento anómalo presenta?. Si el funcionamiento anómalo es que después de introducir una opción nos la ejecuta y luego sin que el usuario haga nada más, el programa dice el solito que se ha introducido una opción errónea ¿a qué puede ser debido esto? **Pista:** al introducir una opción después se da al "intro", que el ordenador interpreta como un carácter más llamado '\n'). ¿Cómo se podría solucionar este problema tan desagradable? Realizar la lectura de la variable **char** según: `scanf("%c", &ch);`.
6. Modifique el ejemplo mostrado en la sección 6.8.2 para evitar el uso de la sentencia **break**.
7. Escribir un **bucle** para pedir un número entero al usuario con control de errores. El número ha de estar comprendido en el intervalo (0, 10]. En caso de que no esté, se imprimirá un mensaje de error y se volverá a pedir el número de nuevo, continuando así hasta que el usuario introduzca un número que esté dentro del intervalo solicitado.
8. Se desea calcular el paso por cero de una función. Para ello suponemos que se han definido dos variables: `v_ini` y `v_fin`, de forma que el signo de la función en `v_ini` es negativo y en `v_fin` es positivo, con lo que se asegura un paso por cero entre ambos valores. Realizar un **bucle** que partiendo de `x = v_ini` recorra todos los valores de `x` con una precisión de 0.01 y que termine cuando se detecte el cambio de signo de la función. Fuera del bucle se imprimirán los valores de `x` y de `f(x)` para dicho paso por cero. Si `x` e `y` son variables reales, el valor de la función se puede obtener con la instrucción `y = f(x);`
9. Se pretende hacer un programa para que el usuario adivine un número entero comprendido entre -100 y 100. El programa genera un número aleatorio y lo almacena en la variable `x`, que es de tipo **int**. Escribir **solamente** el **bucle principal** del programa que debe preguntar valores al usuario hasta que adivine el valor de `x`. Cuando el usuario no adivine el número, el programa le dirá si `x` es menor o mayor que su intento. Cuando finalmente lo adivine, no se debe escribir

WEB

La solución al problema7 se muestra en la sección de códigos típicos de la página web del libro.

nada ya que los mensajes de felicitación se harán al final del programa, fuera del bucle que se pide programar.

10. Escribir el **bucle principal** de un programa que funciona por menú. Las opciones del menú son las siguientes:

MENÚ PRINCIPAL

1-Sumar

2-Restar

3-Mostrar resultado

4-Salir

La idea es que este menú se repita todo el rato hasta que el usuario seleccione la opción 4. En caso de introducir una opción inválida, habrá que mostrar un mensaje de error y volver al menú. Mediante un **switch** habrá que distinguir qué opción ha sido elegida y llamar a una función que se encargue del trabajo. No se puede llamar en ningún momento a la función `exit()`.

CAPÍTULO 7

Funciones

7.1. Introducción

Una técnica muy empleada en la resolución de problemas es la conocida vulgarmente como “divide y vencerás”. Los programas de ordenador, salvo los más triviales, son problemas cuya solución es muy compleja y a menudo dan auténticos dolores de cabeza al programador. La manera más elegante de construir un programa es dividir la tarea a realizar en otras tareas más simples. Si estas tareas más simples no son aún lo suficientemente sencillas, se vuelven a dividir, procediendo así hasta que cada tarea sea lo suficientemente simple como para resolverse con unas cuantas líneas de código. A esta metodología de diseño se le conoce como diseño de **arriba-abajo**, mundialmente conocido como *top-down* por aquello del inglés. Otras ventajas de la división de un problema en módulos claramente definidos es que facilita el trabajo en equipo y permite reutilizar módulos creados con anterioridad si éstos se han diseñado de una manera generalista.

Esta metodología se hace imprescindible hasta en los programas más sencillos; por ello, todos los lenguajes de programación tienen soporte para realizar cómodamente esta división en tareas simples. En el caso de C el mecanismo usado para dividir el programa en trozos son las **funciones** (que en otros lenguajes de programación como el FORTRAN o el BASIC, son llamadas subrutinas).

Una función en C consta de unos argumentos de entrada, una salida, y un conjunto de instrucciones que definen su comportamiento. Esto permite aislar la función del resto del programa, ya que la función puede considerarse como un “programa” independiente que toma sus argumentos de entrada, realiza una serie de operaciones con ellos y genera una salida; todo ello sin interactuar con el resto del programa.

Esta metodología de diseño presenta numerosas ventajas, entre las que cabe destacar:

- La complejidad de cada tarea es mucho menor que la de todo el programa, siendo abordable.
- Se puede repartir el trabajo entre varios programadores, encargándose cada uno de ellos de una o varias funciones de las que se compone el programa. Esto permite cosas tales como reducir el tiempo total de programación (si tenemos un ejército de programadores), el que cada grupo de funciones las realice un especialista en el tema (por ejemplo las de cálculo intensivo las puede realizar un matemático, las de contabilidad un contable, etc.).
- Al ir construyendo el programa por funciones, se pueden ir probando estas funciones conforme se van terminando, sin necesidad de esperar a que se termine

de escribir el programa completo. Esto hace que, tanto la prueba de las funciones, como la corrección de los errores cometidos en ellas, sea mucho más fácil al tener que abarcar solamente unas cuantas líneas de código, en lugar de las miles que tendrá el programa completo.

- Una vez construido el programa, también el uso de funciones permite depurar los problemas que aparezcan más fácilmente, pues una vez identificada la función que falla, sólo hay que buscar el fallo dentro de dicha función y no por todo el programa.
- Permite usar funciones creadas por otros programadores y almacenadas en bibliotecas. Esto ya se ha venido haciendo durante todo el libro. Funciones como `printf` o `scanf` son funciones creadas por los programadores del compilador y almacenadas en la biblioteca estándar de C. Además existen infinidad de bibliotecas, muchas de ellas de dominio público, para la realización de tareas tan diversas como cálculos matriciales, resolución de sistemas de ecuaciones diferenciales, bases de datos, interfaz con el usuario, programación de juegos, etc.
- Si se realizan lo suficientemente generales, las funciones se pueden **reutilizar** en otros programas. Así por ejemplo, si en un programa es necesario convertir una cadena a mayúsculas y realizamos una función que realice dicha tarea, esta función se podrá usar en otros programas sin ningún cambio. Si por el contrario la tarea de convertir a mayúsculas se “incrusta” dentro del programa, su reutilización será muchísimo más difícil.

7.2. Estructura de una función

Como se ha dicho en la introducción, una función tiene unos **argumentos** de entrada, un valor de **salida** y una serie de instrucciones que forman el **cuerpo** de la función. Por ejemplo una función que devuelve el cuadrado de su argumento se escribiría de la siguiente manera:

```
double Cuad(double x)
{
    double res; /* Resultado */

    res = x*x;
    return res;
}
```

La sintaxis genérica de la definición de una función es la siguiente:

```
tipo_devueltoNombreFunción(tipo_1 argumento_1, ..., tipo_n argumento_n)
{
    instrucción_1;
    instrucción_2;
    ...
    instrucción_n;
    return expresión_devuelta;
}
```

en donde:



En C99 es obligatorio declarar el tipo de dato devuelto por la función.

- *tipo_devuelto* es el tipo del dato que devuelve la función. Si se omite, el compilador de C supone que es un **int**; pero para evitar errores es importante especificar el tipo **siempre**. Si la función no devuelve ningún valor ha de usarse el especificador de tipo **void**. No obstante, tenga en cuenta que la mayoría de las funciones realizan un cálculo y deben devolver el resultado.
- *NombreFunción* es el nombre de la función y será usado para llamar a la función desde cualquier parte del programa. Este nombre tiene las mismas limitaciones que los nombres de las variables discutidos en la sección 3.2. Como norma general los nombres de las funciones suelen escribirse con la primera letra en mayúscula, para diferenciarlas de las variables que se escriben en minúsculas. Los nombres compuestos suelen escribirse mezclando mayúsculas y minúsculas, por ejemplo una función que borre la pantalla podría llamarse `BorrarPantalla`.
- *tipo_1 argumento_1, ..., tipo_n argumento_n* es la lista de argumentos que recibe la función. Estos argumentos, también llamados parámetros,¹ pueden ser cualquier tipo de dato de C: números enteros o reales, caracteres, vectores, etc.
- `{` es la llave que marca el comienzo del cuerpo de la función.
- *instrucción_1 ... instrucción_n* son las instrucciones que componen el cuerpo de la función y se escriben con un sangrado de algunos espacios para delimitar más claramente dónde comienza y dónde termina la función.
- **return expresión_devuelta**; es la última instrucción de la función y hace que ésta termine y devuelva el resultado de la evaluación de *expresión_devuelta* a quien la había llamado. Si el *tipo_devuelto* de la función es **void**, se termina la función con la instrucción **return**; (sin *expresión_devuelta*). Sólo en este caso se puede omitir la instrucción **return**, con lo cual la función termina al llegar a la llave `}`.
- `}` es la llave que cierra el cuerpo de la función.

7.3. Prototipo de una función

En la sección anterior se ha visto cómo se **define** una función, pero para que una función pueda usarse en otras partes del programa es necesario colocar al principio de éste lo que se denomina el **prototipo** de la función. La misión de este prototipo es la de **declarar** la función al resto del programa, lo cual permite al compilador comprobar que cada una de las llamadas a la función es correcta, es decir, el compilador verifica:

- Que los argumentos son correctos, tanto en número como en tipo, con lo que se evitan errores como el olvido de un parámetro o suministrar un parámetro de un tipo erróneo. En este último caso el prototipo permite realizar una conversión de tipos si ésta es posible. Por ejemplo si una función que tiene como argumento un **double** recibe un **int** se realizará automáticamente la conversión de **int** a **double**. Si la conversión no es posible, se generará un error; así, si a una función que tiene como argumento una matriz se le suministra en la llamada un entero el compilador generará un error, evitándose la creación de un programa que fallaría estrepitosamente cuando se realizara la llamada errónea.

¹No confundir los parámetros de una función con los parámetros del programa, definidos mediante sentencias **#define** y que se verán más adelante.

- Que el uso del valor devuelto por la función sea acorde con su tipo. Por ejemplo no se puede asignar a una variable el valor devuelto por una función si la variable es de tipo **int** y la función devuelve un vector.

El uso de prototipos es opcional: el estándar de C sólo obliga a declarar las funciones si éstas devuelven un tipo distinto de **int**. Sin embargo el uso de prototipos es extremadamente recomendable pues permite al compilador detectar errores que, de no existir el prototipo, pasarían inadvertidamente al programa y éste fallaría por completo al intentar ejecutarlo.

La sintaxis del prototipo de una función es la siguiente:

```
tipo_devuelto NombreFunción(tipo_1 argumento_1, ..., tipo_n argumento_n);
```

Que como puede observarse se corresponde con la primera línea de la definición de la función pero terminada en un punto y coma, con lo que los más avisados ya habrán adivinado que la técnica del cortar y pegar será sumamente útil en la escritura de prototipos: ahorra teclear dos veces lo mismo y además evita errores.

7.4. Ejemplo

Para aclarar ideas veamos un ejemplo sencillo que ilustra el uso de una función dentro de un programa. Se necesita realizar un programa para calcular la suma de una serie de números reales de la forma:

$$\sum_{n=a}^b n = a + (a + 1) + \dots + b$$

La manera correcta de realizar la suma de la serie es definir una función para ello. Esto permitirá aislar esta tarea del resto del programa, poder usar la función en otros programas y todas las ventajas adicionales discutidas en la introducción.

La definición de la función se realizaría de la siguiente manera:

```

38  /* Función: SumaSerie
   *
   * Descripción: Suma la serie aritmética comprendida entre sus
40  *               argumentos a y b.
   *
42  * Argumentos: int a: Valor inicial de la serie.
   *               int b: Valor final de la serie.
44  *
   * Valor devuelto: int: Resultado de la serie.
46  *
   * Revisión 0.1: 19/04/1998.
48  *
   * Autor: El funcionario novato.
50  */

52  int SumaSerie(int a, int b)
   {
54      int suma; /* Almacena la suma parcial de la serie */

```

```

56 suma = 0;

58 while(a <= b){
    suma += a;
60     a++;
    }
62 return suma;
    }

```

En primer lugar cabe destacar la ficha de la función. Es análoga a la ficha del programa pero añadiendo la lista de argumentos y el valor devuelto. Esta ficha es la parte más importante de la función de cara a su reutilización en otros programas, tanto por nosotros como por otros programadores, pues sin necesidad de leer ni una sola línea de código de la función podemos saber la tarea que realiza, los datos que necesitamos suministrarle y el valor que nos devuelve.

A continuación de la ficha se encuentra la definición de la función, en la que apreciamos en la primera línea (52) el tipo devuelto, el nombre de la función y su lista de argumentos entre paréntesis.²

En la siguiente línea se encuentra la llave { que delimita el comienzo del cuerpo de la función.

A continuación, ya dentro del cuerpo de la función, se encuentran las definiciones de las variables necesarias en el interior de la función. Estas variables se denominan **variables locales** puesto que sólo son accesibles desde dentro de la función. En este ejemplo la variable `suma`, declarada en la línea 54, sólo es conocida dentro de la función y por tanto su valor sólo puede ser leído o escrito desde dentro de la función, pero no desde fuera.³

Después de las definiciones de las variables locales se encuentran las instrucciones necesarias para que la función realice su tarea.

Cabe destacar que, tal como se aprecia en las líneas 58, 59 y 60, los parámetros de la función son variables normales que se utilizan igual que las variables locales y que no necesitan redeclararse.

Por último en la línea 62 se usa la instrucción **return** para salir de la función devolviendo el valor de `suma`.

Veamos a continuación el resto del programa para ilustrar el modo de llamar a una función.

```

/* Programa: SumaSerie
2  *
   * Descripción: Calcula la suma de una serie aritmética entre un
4  *               valor inicial y un valor final. Para ello se apoya
   *               en la función SumaSerie.
6  *
   * Revisión 0.1: 19/04/1998
8  *
   * Autor: El funcionario novato.
10 */

```

²Nótese que **no** se termina esta sentencia con un punto y coma

³De hecho las variables locales se crean al entrar en la función y se destruyen al salir de ella, por lo que físicamente es imposible conocer o modificar sus valores fuera de la función porque las variables ni siquiera existen.

```

12 #include <stdio.h>
   #include <stdlib.h>
14
   /* prototipos de las funciones */
16
   int SumaSerie(int a, int b);
18
   int main(void)
20 {
   int inicial; /* Valor inicial de la serie */
22   int final; /* Valor final de la serie */
   int resultado; /* Resultado de la suma de la serie */
24
   printf("Valor inicial de la serie: ");
26   scanf("%d", &inicial);

28   printf("Valor final de la serie: ");
   scanf("%d", &final);
30

   resultado = SumaSerie(inicial, final);
32   printf("El resultado es: %d\n", resultado);
   return 0;
34 }

```

En donde lo primero que cabe destacar es la inclusión del prototipo de la función al principio del archivo (línea 17). Esto permite que el compilador conozca el prototipo antes de que ocurra cualquier llamada a la función `SumaSerie` dentro del archivo y pueda comprobar que las llamadas a dicha función son correctas (número de argumentos, tipos y valor devuelto).

Dentro de la función `main` se piden al usuario los valores inicial y final de la serie y en la línea 31 se **llama** a la función `SumaSerie`. Esta línea ilustra dos aspectos muy importantes de las funciones en C:

- Una función que devuelve un tipo de dato “t” se puede usar en cualquier expresión en donde se puede usar una variable del mismo tipo “t”. En nuestro caso se puede introducir en cualquier lugar en el que introduciríamos un número o una variable de tipo `int`. Lo más habitual es asignar el resultado de la función a una variable del tipo adecuado, como se hace en la línea 31 del programa anterior
- Los nombres de los argumentos en la llamada no tienen por qué ser iguales a los nombres usados para dichos argumentos en la definición de la función. Esta característica, junto con la posibilidad de trabajar con variables locales, es la que permite reutilizar fácilmente las funciones. Es incluso aconsejable utilizar nombres distintos ya que en realidad son variables distintas, como se ve a continuación.

7.5. Paso de argumentos a las funciones. Variables locales

En el ejemplo anterior se acaba de ilustrar el mecanismo de llamada a una función. Sin embargo al lector espabilado igual le ha surgido una duda: ¿qué le ocurre a la variable inicial cuando la pasamos como argumento a la función `SumaSerie`?

Como se puede apreciar en la línea 60 del programa, dentro de la función `SumaSerie` se incrementa el valor del primer argumento, el cual se corresponde con la variable inicial dentro de `main`. La pregunta es si este incremento se realiza también sobre la variable inicial. La respuesta es que **no**. Todas las variables de la función, tanto los argumentos (`a` y `b`) como las definidas localmente (`suma`) son variables temporales que se crean cuando se llama a la función y que desaparecen al salir de ésta. En la llamada a la función (línea 31) los valores almacenados en las variables `inicial` y `final` se copian en las variables temporales `a` y `b` con lo que cualquier operación que se realice con estas copias (lecturas, escrituras, incrementos, etc.) no influirán para nada en las variables originales.

Esto que se acaba de discutir es uno de los aspectos más importantes de las funciones: el hecho de que la función trabaje con sus propias variables la aísla del resto del programa, pues se evita modificar de forma accidental una variable que no pertenezca a la función. Para ilustrar este aspecto, imaginemos que el programa en el que hay que realizar la suma de la serie descrito en el apartado anterior es mucho más largo y que como no hacemos nunca caso del profesor, decidimos no usar funciones en el programa. Para sumar la serie se podría hacer:

```
...
suma = 0;
while(inicial<=final){
    suma += inicial;
    inicial++;
}
resultado = suma;
...
```

Los problemas que plantea esta solución son varios:

- En primer lugar el valor inicial nos lo hemos cargado al ir incrementándolo, por lo que si intentamos realizar la tarea propuesta en el ejercicio 1 nos daremos cuenta que el valor inicial es mayor que el final después de sumarse la serie.
- Hemos usado una variable llamada `suma` para almacenar la suma parcial de la serie, pero ¿qué ocurre si en otra parte del programa se ha usado una variable también llamada `suma`? Obviamente el valor que tuviese desaparece con el cálculo de la serie, y si se usa después el programa fallará estrepitosamente y para colmo será difícil encontrar el error.
- Si en otra parte se quiere calcular otra serie entre las variables `ini` y `fin` y queremos aprovechar este trozo de código tendremos que cortar y pegar esta parte del programa en la parte apropiada y luego cambiar `inicial` por `ini` y `final` por `fin` y tal vez `suma` por cualquier otra cosa si queremos conservar el valor de la serie anterior. Además, si se nos olvida cambiar alguna de las variables, el compilador no dará ningún mensaje de error ya que todas las variables son válidas; pero sin embargo el resultado será incorrecto y el error será muy difícil de encontrar.
- Al menos lo único que sí está bien hecho en este ejemplo es que la variable `suma` se inicializa a cero justo delante del bucle y no en la zona de declaración de variables. Al hacerlo así no afecta a nuestro cálculo que `suma` se haya modificado en otra parte del programa, ni que nuestro código se encuentre dentro de un bucle, ni que se nos olvide la inicialización si lo copiamos a otro sitio.



Para ilustrar lo expuesto en esta sección, realice el ejercicio 1.

Como puede comprobar el lector, las ventajas de usar funciones son muy numerosas y el único inconveniente es aprender a usarlas. No obstante, le aseguramos que el pequeño esfuerzo realizado en aprender su uso, compensa con creces los dolores de cabeza que tendría si se empeñase en hacer programas enormes, sin dividir el trabajo en funciones más pequeñas.

7.6. Salida de una función y retorno de valores

Para salir de una función se usa la instrucción **return** seguida del valor que se desea devolver. Su sintaxis es:

return *expresión*;

En donde *expresión* es cualquier expresión válida de C aunque la mayoría de las veces se trata del nombre de la variable que contiene el resultado de un cálculo. Si el tipo de dato de la expresión no coincide con el tipo de la función, se realizarán las conversiones apropiadas siempre que éstas sean posibles o el compilador generará un mensaje de error si dicha conversión no es posible.

Debido a la gran flexibilidad del lenguaje C (quizás excesiva), la instrucción de salida puede aparecer en cualquier parte de la función, pudiendo existir varias. Por ejemplo una función que devuelve el máximo de sus dos argumentos se puede escribir como:

```
int Max(int a, int b)
{
    if(a > b){
        return a;
    }else{
        return b;
    }
}
```

Sin embargo **no conviene usar esta técnica**, pues cualquier función es más fácil de entender si sólo existe un **return** al final de la misma. El haber mencionado esto en este libro se debe sólo al deseo de los autores de que el lector no se asuste si al leer un código escrito por otro programador ve algo parecido. Una alternativa mucho más estructurada, que es la recomendada por los autores es la siguiente:

```
int Max(int a, int b)
{
    int res;

    if(a > b){
        res = a;
    }else{
        res = b;
    }
    return res;
}
```

7.6.1. Funciones **void**

Si se desea realizar una función que no devuelva nada se debe usar el tipo **void**. Así, si tenemos una función que no devuelve nada, su declaración será:

```
void FuncionQueNoDevuelveNada(int arg1, double arg2);
```

y su definición sería:

```
void FuncionQueNoDevuelveNada(int arg1, double arg2)
{
    ...
}
```

En este tipo de funciones que no devuelven nada, la instrucción de salida es opcional, terminando la función automáticamente cuando el flujo del programa alcanza la llave (}) que cierra el cuerpo de la función. Si se desea poner explícitamente la instrucción de salida o si se desea salir desde otro punto de la función la sintaxis es:

```
return;
```

dando el compilador un aviso en caso de que **return** esté seguido de alguna expresión. Del mismo modo si una función que ha de devolver un valor no lo devuelve mediante la instrucción **return**, el compilador dará un aviso. Un ejemplo de función **void** es una función para dar un mensaje de bienvenida, que sólo muestra texto por la pantalla pero no realiza ningún cálculo y por lo tanto no necesita devolver nada.

7.7. Funciones sin argumentos

Al igual que se pueden definir funciones que no devuelven nada, también se pueden definir funciones que no reciben ningún argumento. Un ejemplo de este tipo de funciones es `getchar`, disponible en la librería estándar, la cual lee un carácter del teclado y lo devuelve como un **int**. El prototipo de esta función es:

```
int getchar(void);
```

donde podemos apreciar que al no recibir ningún argumento se ha puesto **void** como lista de argumentos. En general una función que no toma ningún argumento se declara como:

```
tipo_devuelto nombre_función(void);
```

Un ejemplo habitual de este tipo de funciones es lo que normalmente llamamos el “programa principal”, que no es otro que la función `main`. Esta función normalmente no recibe ningún argumento y devuelve un código de error entero (que suele ser cero si todo ha ido bien): **int** `main(void)`.

7.8. La pila

En la sección 7.5 se ha mencionado que al entrar en una función se crean, como por arte de magia, sus variables locales y sus argumentos y al salir de ésta, también por arte de magia, estas variables se destruyen. Bien, como decía Clarke, “cualquier tecnología suficientemente avanzada es indistinguible de la magia”. En este caso obviamente las variables se están creando y destruyendo gracias a la tecnología avanzada,

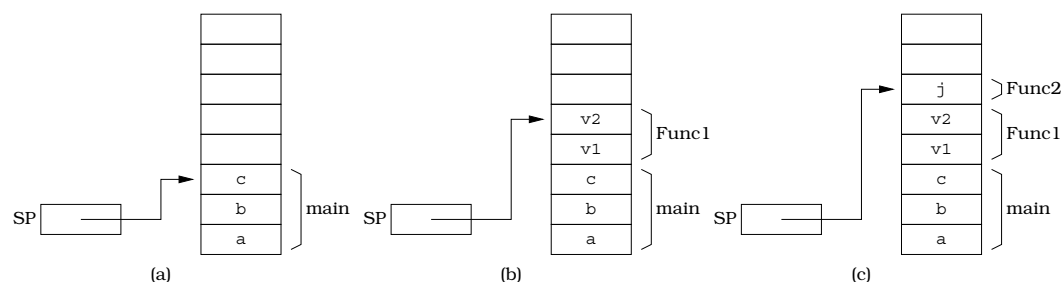


Figura 7.1: Funcionamiento de la pila

no gracias a ningún mago. En esta sección se va a exponer cómo se almacenan estas variables.

En primer lugar hay que tener en cuenta que en un programa una función puede llamar a otras funciones, éstas a su vez pueden llamar a otras y así sucesivamente. Por tanto es necesario buscar algún mecanismo flexible para que cada función pueda disponer de su trozo de memoria para almacenar sus variables. Este mecanismo consiste en una zona de memoria, denominada la pila, que se reserva en el ordenador para este propósito y un registro interno en la CPU, denominado SP⁴ que almacena la dirección del último valor que se ha guardado en esta zona de memoria, tal como se muestra en la figura 7.1. En esta figura se muestra en primer lugar el estado de la pila cuando empieza a ejecutarse la función main, suponiendo que ésta usa tres variables locales denominadas a, b y c. Desde la flecha hacia arriba la memoria está libre (ver figura 7.1(a)). Si se llama a una función denominada Func1, la cual define dos variables, v1 y v2, el estado de la pila durante la ejecución de esta función es el mostrado en la figura 7.1(b). Como se puede observar se han “depositado” estas dos variables en el tope de la pila, incrementando el registro SP para que apunte al nuevo límite.⁵ De este modo la CPU, a través del registro SP puede acceder a las variables de la función. De hecho las funciones sólo pueden acceder a las variables almacenadas en su trozo de pila. Si la función Func1 llama a la función Func2 y ésta define una variable denominada j, la nueva variable se “depositará” también en el tope de la pila, incrementándose el registro SP de nuevo, tal como se muestra en la figura 7.1(c). Nótese que mientras se está ejecutando la función Func2 la CPU sólo puede acceder a la variable j. Cuando termine de ejecutarse la función Func2 se “extrae” la variable j de la pila, decrementándose el registro SP para apuntar a las variables de la función Func1. La pila queda nuevamente en la situación mostrada en la figura 7.1(b). Cuando retorne la función Func1 se extraerán las variables v1 y v2 de la pila, volviendo a la situación original mostrada en la figura 7.1(a).

Por último destacar que el llamar a esta zona de memoria la pila viene del hecho de que su funcionamiento, tal como se ha visto, es similar a la de una pila de papeles en donde se añaden nuevas hojas por arriba (cuando se llama a una función) y se retiran las hojas también por arriba (cuando las funciones retornan).

⁴Del inglés *Stack Pointer* o puntero de pila.

⁵Además de las variables, en la pila se almacena la dirección de memoria desde donde se ha llamado a la función, de forma que cuando ésta termina se pueda continuar ejecutando el programa desde donde se llamó a la función. Para simplificar la exposición, estas direcciones no se han mostrado en las figuras 7.1 y 7.2.

7.9. Funciones recursivas

Tal como se ha expuesto en la sección anterior, como cada función al comenzar a ejecutarse crea en la pila un espacio para sus variables, no hay ningún problema para que una función llame a otra función. Es más, tampoco hay ningún problema si una función se llama a sí misma. Se dice entonces que esa función es una función recursiva. Sin duda se preguntará ahora: ¿y para qué sirve una función que se llame a sí misma? Pues está claro: para programar algoritmos recursivos. El ejemplo más sencillo es el factorial de un número, que puede definirse mediante un algoritmo recursivo como:

$$Fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot Fact(n-1) & \text{si } n > 0 \end{cases}$$

La implantación en C de este algoritmo recursivo es bien sencilla:

```

long int Factorial(long int n)
2 {
    long int fact;
4
    if(n == 0){
6        fact = 1;
    }else{
8        fact = n * Factorial(n-1);
    }
10 return fact;
    }

```

Como puede observarse, la implantación en C es directa a partir de la definición matemática. Lo único a destacar es que en la línea 8 se llama a la función `Factorial`, sin que ello implique nada especial. Para el ordenador es lo mismo llamar a una función con nombre distinto que con el mismo nombre de la función que hace la llamada. Para aclarar el proceso, veamos qué ocurre cuando se llama a la función para calcular el factorial de 3, por ejemplo haciendo:

```
res = Factorial(3);
```

El programa saltará entonces a la función `Factorial`, pasándole como argumento un 3. La función creará espacio en la pila para almacenar el argumento `n` y su variable local `fact`, que de momento estará sin inicializar. La situación de la pila será la mostrada en la figura 7.2(a). Cuando se ejecute la línea 8 de la función, se evaluará la expresión que consiste en multiplicar `n` (que vale 3) por el resultado de ejecutar la función `Factorial` con 2 como argumento. Por tanto, la CPU volverá a llamar a la función `Factorial`, creándose ahora en la pila otras dos variables para la nueva instancia de la función, una para su argumento, que ahora vale 2, y otra para su variable local, que al igual que antes estará sin inicializar. La situación de la pila será la mostrada en la figura 7.2(b). El proceso se repetirá hasta que se llame a `Factorial` con un argumento igual a 0, tal como se muestra en las figuras 7.2(c) y 7.2(d). En esta última llamada –`Factorial(0)`–, se ejecutará la línea 6 y se asignará el resultado 1 a la variable `fact` de esta instancia de la función `Factorial`, tal como se ha mostrado en la figura 7.2(d). Este valor se retornará a quien la había llamado, que no era más que otra instancia de ella misma pero con argumento 1. Por tanto, se terminará de ejecutar la línea 8 de esa instancia de la función, obteniéndose como resultado un 1 que se asigna a la



En la Web del libro hay una demo del programa para calcular el factorial.

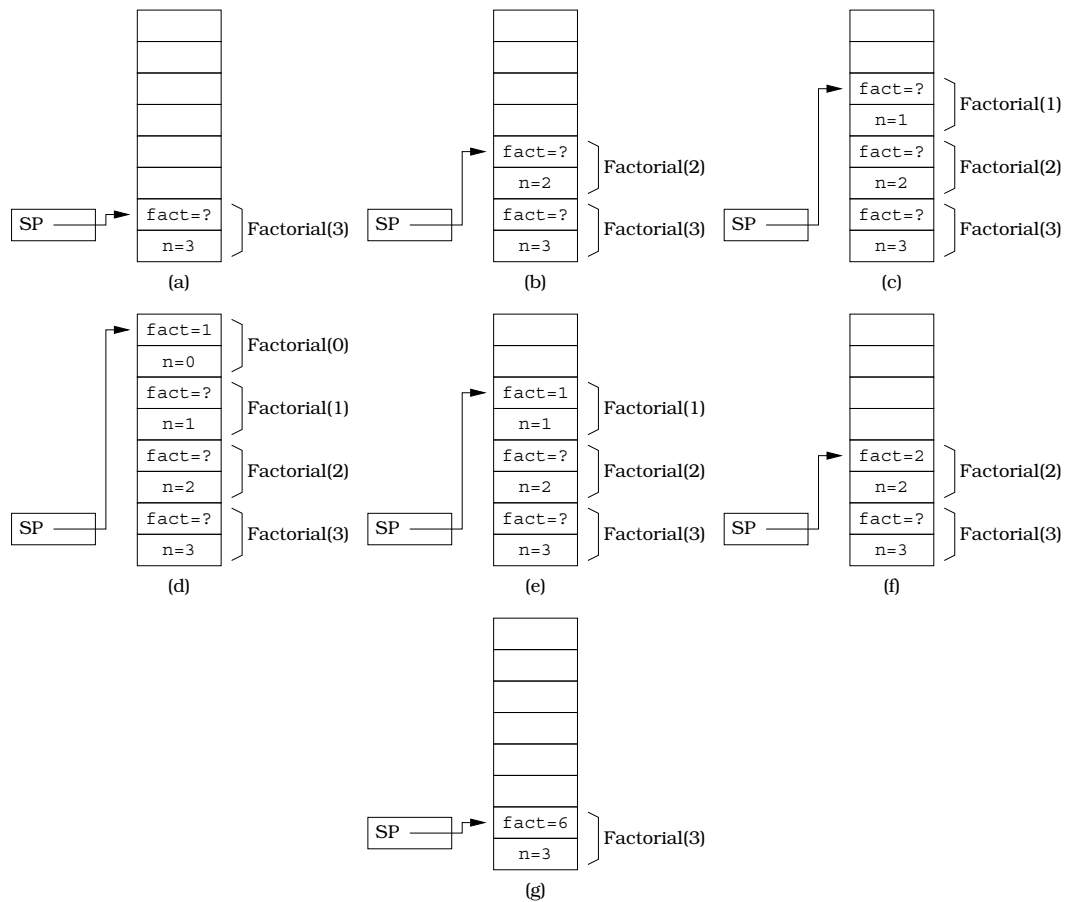


Figura 7.2: Evolución de la pila durante la ejecución de Factorial

variable `fact` de esta instancia de la función. La situación de la pila en ese momento será la mostrada en la figura 7.2(e). `Factorial(1)` terminará de ejecutarse, devolviendo un 1 a `Factorial(2)`. El proceso se repetirá, finalizándose la ejecución de la línea 8 de las instancias en curso de `Factorial`, obteniéndose el valor de la variable `fact` y retornándola. La evolución de la pila se muestra en las figuras 7.2(f) y 7.2(g).

Conviene destacar dos aspectos de este ejemplo:

- Mientras se están evaluando las sucesivas llamadas, existen múltiples instancias de la misma función ejecutándose “a la vez”. El código a ejecutar es el mismo, pero los argumentos son distintos.
- El proceso necesita gran cantidad de memoria para almacenar en la pila los argumentos y las variables locales. Además las sucesivas llamadas recursivas a la función consumen tiempo de CPU, por lo que el proceso es **ineficiente**.
- Todas las funciones recursivas tienen que tener una condición de salida (en este ejemplo cuando `n` vale 0) y debe estar garantizado que la llamada recursiva siempre se realiza con distintos argumentos. Si no se cumplen estas condiciones, el

algoritmo será infinito y el programa sólo terminará cuando se agote la memoria del ordenador.

7.9.1. Recursividad frente a iteratividad

Como se acaba de mencionar, la ejecución de funciones recursivas es ineficiente. Esto hace que sea casi siempre mejor implantar un algoritmo de forma iterativa que de forma recursiva, siempre y cuando dicho algoritmo exista. Por ejemplo, en el caso del factorial, puede usarse en lugar del algoritmo anterior, el siguiente algoritmo iterativo:

$$Fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ \prod_{i=1}^n i & \text{si } n > 0 \end{cases}$$

Que puede implantarse en C de la siguiente manera:

```
long int FactorialIte(long int n)
{
    long int fact; /* valores parciales de factorial */
    int i; /* Índice para el cálculo del factorial */

    fact = 1;
    for(i=1; i<=n; i++){
        fact = fact * i;
    }
    return fact;
}
```

Para ilustrar este punto, se han ejecutado ambas versiones de la función para calcular el factorial de 13 en un ordenador Pentium 4 a 2.6 GHz y mientras que la versión iterativa tarda en ejecutarse 1,5 µs, la versión recursiva tarda 4 µs.⁶

No obstante existen algoritmos que son puramente recursivos y no disponen de versiones iterativas. Es por ello que todos los lenguajes de programación modernos soportan la recursividad. Ejemplos de este tipo de algoritmos son la ordenación mediante QuickSort, la evaluación de expresiones matemáticas, la gestión de árboles de búsqueda o la resolución del juego de las torres de Hanoi.

A modo de conclusión, los algoritmos que se puedan implementar fácilmente de manera iterativa o recursiva se deben programar de manera iterativa por eficiencia. Sin embargo los algoritmos que típicamente se definen de manera recursiva (como los ejemplos anteriores) deben programarse de manera recursiva, porque aunque en algunos casos fuese posible inventarse algún truco que permita programarlos sin recursividad, el código resultante es muy enrevesado, difícil de entender y la posibilidad de obtener mayor eficiencia disminuye.



Realice el ejercicio 8.

7.10. Ejemplos

Veamos a continuación algunos ejemplos de funciones para aclarar ideas.

⁶Para obtener estos datos se realizó un bucle para calcular el factorial 1 millón de veces y se midió el tiempo de ejecución de la función con el *profiler gprof* del proyecto GNU.

7.10.1. Elevar al cuadrado

Como en C no existe el operador para elevar un número a una potencia⁷ y la función `pow` es demasiado ineficiente para calcular un cuadrado, hemos decidido realizar una función que devuelva el cuadrado de su argumento. Si trabajamos con números reales tanto el argumento como el valor devuelto por la función serán de tipo **double**. La función, junto con un pequeño programa que ilustra su uso, se muestra a continuación.

```
/* Programa: Cuadrados
 *
 * Descripción: Ejemplo de uso de la función cuadrado.
 *
 * Revisión 0.0: 15/04/1998.
 *
 * Autor: El funcionario novato.
 */

#include <stdio.h>

/* prototipos de las funciones */
double Cuadrado(double a);

int main(void)
{
    double valor;
    double cuad;

    printf("Introduzca un valor ");
    scanf("%lf", &valor);
    cuad = Cuadrado(valor);
    printf("El cuadrado de %g es: %g\n", valor, cuad);
    return 0;
}

/* Función: Cuadrado
 *
 * Descripción: Devuelve el cuadrado de su argumento.
 *
 * Argumentos: double a: Valor del que se calcula el cuadrado.
 *
 * Valor devuelto: double: El argumento a al cuadrado.
 *
 * Revisión 0.0: 15/04/1998.
 *
 * Autor: El funcionario novato.
 */

double Cuadrado(double a)
```

⁷Recordemos que $^{\wedge}$ se utiliza en otros lenguajes para elevar a una potencia, pero en C tiene un significado diferente.

```
{
    double res;

    res = a*a;
    return res;
}
```



Realice los ejercicios 2 al 4.

7.10.2. Factorial

Una función muy usada en estadística que no está disponible en la librería matemática estándar de C es el factorial. En este ejemplo vamos a realizar una función que calcula el factorial de un número entero. La función, junto con un programa que la usa, se muestra a continuación:

```
/* Programa: Factoriales.
 *
 * Descripción: Ejemplo de uso de la función factorial.
 *
 * Revisión 0.0: 15/04/1998.
 *
 * Autor: El funcionario novato.
 */

#include <stdio.h>

/* prototipos de las funciones */
long int Factorial(long int a);

int main(void)
{
    long int valor;

    printf("Introduzca un valor ");
    scanf("%ld", &valor);
    printf("El factorial de %ld es: %ld\n", valor, Factorial(valor));
    return 0;
}

/* Función: Factorial.
 *
 * Descripción: Devuelve el factorial de su argumento.
 *
 * Argumentos: long int a: Valor del que se calcula el factorial.
 *
 * Valor devuelto: long int: El factorial del argumento a.
 *
 * Revisión 0.0: 15/04/1998.
 *
 * Autor: El funcionario novato.
 */
```



```

long int Factorial(long int a)
{
    long int fact; /* valores parciales de factorial */

    fact = 1;
    while(a>0){
        fact *= a;
        a--;
    }
    return fact;
}

```



Realice el ejercicio 5.

7.11. Recomendaciones y advertencias

- Cualquier aviso (*warning*) del compilador relacionado con el uso de las funciones o con incoherencias con el prototipo, debe ser analizada cuidadosamente. Estas advertencias son siempre síntomas de problemas reales.
- Es muy recomendable utilizar nombres de variables diferentes en el programa principal y en las funciones a las que llama. Al compilador la verdad es que le da igual, pero a nosotros nos viene bien para no pensar que la variable usada en la función es la misma que en el main, ya que sólo inicialmente tienen el mismo valor. (Ver ejemplo 7.10.2)
- Un error muy común es volver a declarar los argumentos dentro de la función, tal como se muestra en el siguiente ejemplo:

```

int Funcion(int arg)
{
    int arg; /* Error grave */
    ...
}

```

Lo que ocurre en estos casos es que la variable local a “oscurece” al argumento. Es decir, cuando dentro de la función escribamos a se usará el valor de la variable local (que no se ha inicializado) en lugar del valor del argumento. El compilador genera un ejecutable a pesar de este error, aunque afortunadamente nos da un aviso. Por ejemplo el compilador GCC nos dice:

aviso: la declaración de ‘arg’ oscurece un parámetro

- Para mayor claridad del código, las funciones deben tener un único **return**, que estará situado al final de la función.
- Salvo casos especiales, las funciones no deben realizar operaciones de entrada salida (`scanf` o `printf`), sino que se limitarán a realizar un cálculo o proceso y devolver su resultado.
- Para no cometer errores con el tipo de dato devuelto, se recomienda declarar una variable del mismo tipo devuelto por la función para asignarle el valor que queramos devolver. Por ejemplo:

```
int Calculo(int a)
{
    int res;

    res = cálculo complicadísimo;
    return res;
}
```

- Los nombres de las funciones se escriben con la primera letra en mayúscula y el resto en minúscula. Si el nombre es compuesto, se comenzará cada palabra con mayúscula para facilitar la lectura, como por ejemplo: `MiFuncion`, `CalcularValorMedio`.

7.12. Resumen

En este capítulo se ha introducido el concepto de función, que permite escribir los programas de una manera organizada, resultando fáciles de entender, de depurar y de reutilizar.

Las funciones en C reciben varios argumentos (o ninguno) cuyo valor inicial es una copia del valor de las variables que se utilizan en la llamada y devuelven un único resultado (o ninguno si la función es de tipo **void**). En el capítulo 9 se verá cómo hacer que una función devuelva varios valores.

También se ha visto que las funciones trabajan con una serie de variables locales que se crean cuando ésta empieza a ejecutarse y se destruyen cuando se sale de la función. Esto permite que la función esté aislada del resto de las funciones del programa evitándose “daños colaterales” como la modificación inadvertida de una variable externa a la función.

7.13. Ejercicios

1. Introduzca el programa mostrado en la sección 7.4 en el ordenador y modifíquelo para que se imprima en la pantalla el valor inicial de la serie, el valor final y su suma. En primer lugar realice la impresión desde `main`, justo después de la llamada a la función. Para ello inserte el nuevo código a partir de la línea 31. En segundo lugar imprima los mismos valores desde dentro de la función `SumaSerie`, justo antes del **return**. ¿Qué diferencias observa?
2. Modificar el programa de la sección 7.10.1 para calcular el cuadrado de un número entero. En primer lugar modificar sólo el programa principal, de forma que se lea un número entero y se llame a la misma función de antes y se imprima el resultado. ¿Funciona todo correctamente? ¿Qué conversiones automáticas se están realizando?
3. Modificar ahora la función `cuadrado` para que calcule el cuadrado de un **int** y volver a probar el programa del ejercicio anterior.
4. Usar la función `cuadrado` diseñada en el ejercicio 3 en el ejemplo de la sección 7.10.1. ¿Funciona correctamente? ¿Qué tipo de conversiones se realizan?
5. Introducir el programa para calcular el factorial mostrado en la sección 7.10.2 en el ordenador y comprobar su funcionamiento para valores positivos y negativos. ¿Da siempre resultados correctos?

6. Modificar la función factorial del ejercicio anterior para que devuelva -1 en caso de que no se pueda calcular el factorial de su argumento. No olvide cambiar la documentación de la función. Corregir el programa principal para que imprima un mensaje de error si factorial devuelve un -1.
7. Escribir una función llamada min() que devuelva el mínimo de tres variables (a, b, c) de tipo **double** que se pasan como argumento.
8. Escriba un programa para calcular la serie de Fibonacci de un número n. Dicha serie se obtiene según la fórmula:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

Para ello creará una función que calcule la serie y un programa principal que pedirá al usuario el número n, calculará la serie llamando a la función e imprimirá el resultado. Deberán escribirse dos versiones de la función, una recursiva y otra iterativa.

CAPÍTULO 8

Vectores y Matrices

8.1. Introducción

En el capítulo 3 se han descrito los tipos de datos básicos que permiten declarar variables. Pero estos tipos de datos sólo permiten construir variables que almacenen un único valor. En este capítulo se ve cómo definir variables capaces de almacenar más de un valor bajo un mismo nombre, lo que nos permite trabajar con mayor comodidad con grandes conjuntos de datos. Estas variables se utilizan principalmente para trabajar con vectores, cadenas de caracteres y matrices.

8.2. Vectores

Un vector es un conjunto de datos del mismo tipo que se almacenan en el ordenador en posiciones de memoria consecutivas y a los cuales se accede mediante un mismo nombre de variable. La característica fundamental es que todos los datos de un vector son del mismo tipo, por ejemplo todos son **int** o todos son **double**.

La definición de vectores es parecida a la definición de variables, salvo que se debe especificar el tamaño del vector entre corchetes []. Por ejemplo, para definir un vector llamado `vec` que pueda almacenar 10 valores tipo **double** se escribe:

```
double vec[10];
```

La manera de acceder a los valores de estas variables es ahora diferente, ya que de todos los elementos que componen el vector es necesario especificar cuál queremos modificar. Esta especificación se realiza indicando entre corchetes el número de orden del elemento, teniendo en cuenta que la numeración de los elementos siempre empieza en cero.¹

```
vec[0] = 54.23;    /*Primer elemento del vector*/  
vec = 4.5;        /* ERROR */  
vec[10] = 98.5;   /* ERROR */
```

La segunda asignación no es correcta porque `vec` no es una variable tipo **double** sino un vector y por lo tanto todas las asignaciones deben indicar el número de orden del elemento al que queremos acceder. El vector `vec` del ejemplo tiene tamaño 10, porque fue declarado como **double** `vec[10]`, esto significa que almacena 10 elementos. Los 10 elementos se numeran desde el 0 hasta el 9 y por lo tanto, el elemento 10 no existe y la tercera asignación tampoco es válida. Es importante destacar que el

¹En matemáticas el primer elemento de los vectores suele ser el elemento 1 y por lo tanto en el vector a se habla de a_1 como primer elemento y en la matriz A se habla de $A_{1,1}$

compilador no dará error en el último caso ya que no comprueba los rangos de los vectores. Este tipo de asignación hace que el valor 98.5 se escriba fuera de la zona de memoria correspondiente al vector `vec`, y probablemente machaca los valores de otras variables del programa. Hay que prestar especial atención ya que sólo en algunos casos aparece un error de ejecución (generalmente cuando el índice del vector es muy grande) pero en otras ocasiones no aparece ningún mensaje de error y el programa simplemente funciona mal.

Aunque es posible declarar 10 variables normales tipo `int` con nombres `v0`, `v1` ... `v9`; la utilización de vectores simplifica los programas ya que el índice que identifica el elemento del vector (escrito entre corchetes) puede ser una variable entera. Por ejemplo, una alternativa a la inicialización anterior sería:

```
i = 0;
vec[i] = 54.23;
```

Esto permite poder acceder a cualquier elemento del vector bajo el control del programa. Así, si se desea “recorrer” un vector, es muy fácil hacerlo variando el índice mediante un bucle `for`. El siguiente ejemplo declara un vector de tamaño 10, lo inicializa dándole valores crecientes y luego escribe todo el contenido del vector.

```
/* *****
Programa: vector.c
Descripción: Inicializa un vector de enteros y luego
             escribe su contenido por pantalla.
***** */

#include <stdio.h>

int main(void)
{
    double x[10];
    int i;

    /* inicialización */
    for(i=0; i<10; i++) {
        x[i] = 2*i;
    }

    /* imprimir valores */
    for(i=0; i<10; i++) {
        printf("El elemento %d vale %4.1f\n", i, x[i]);
    }
    return 0;
}

--- SALIDA -----
El elemento 0 vale 0.0
El elemento 1 vale 2.0
El elemento 2 vale 4.0
El elemento 3 vale 6.0
El elemento 4 vale 8.0
El elemento 5 vale 10.0
```

```
El elemento 6 vale 12.0
El elemento 7 vale 14.0
El elemento 8 vale 16.0
El elemento 9 vale 18.0
```

Por último, conviene destacar que es bastante habitual no conocer, al escribir el programa, cuantos elementos necesitan los vectores. En estos casos lo que se suele hacer es declarar los vectores suficientemente largos y paralelamente mantener una variable entera que indique cuantos valores útiles contiene realmente el vector. Como consecuencia, la definición de vectores en este caso es la siguiente:

```
double vec[10]; /* Vector de 10 elementos o menos */
int vec_n = 0; /* Número de elementos del vector (0 de momento) */
```

8.3. Cadenas de caracteres

Las cadenas de caracteres son conjuntos de letras que forman palabras o frases. Por ejemplo el nombre de una persona o su dirección se almacenan en forma de **cadenas de caracteres** dentro de la memoria del ordenador. En el lenguaje de programación C no existe un tipo de datos específico para manejar cadenas de caracteres, sino que éstas se almacenan en vectores de datos tipo **char**. Si por ejemplo queremos definir una variable capaz de almacenar nombres de personas de hasta 10 caracteres de largo, podemos definir un vector de **char** de longitud 10* de la manera siguiente:

```
char nombre[10];
```

Utilizando este tipo de definición puede deducirse que siempre se puede acceder a cualquier carácter de la cadena de forma individual. Esto se hace especificando el índice del vector al que queremos acceder. Un programa análogo al anterior, que inicializa e imprime un vector, pero aplicado a cadenas de caracteres, es el siguiente:

```
/******
Programa: cadenas1.c
Descripción: Inicializa un vector de char y luego escribe
el contenido del vector carácter por carácter.
Revisión 0.0: 14/ABR/1998
Autor: Rafael Palacios
*****/
```

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

```
int main(void)
{
    char nombre[10];
    int i;

    /* inicialización */
    nombre[0] = 'H';
    nombre[1] = 'o';
    nombre[2] = 'l';
```

*Más adelante veremos que normalmente se define de longitud 11, uno más que el número máximo de caracteres que contendrá la cadena.

```

nombre[3] = 'a';

/* imprimir valores */
for(i=0; i<4; i++) {
    printf("%c", nombre[i]);
}
printf("\n");
return 0;
}

```

En el programa anterior sólo hay cuatro valores en la cadena de caracteres, por lo tanto el número 4 se utiliza para controlar el final del bucle. A pesar de que la variable `nombre` tiene espacio para almacenar 10 caracteres, sería un error poner el valor 10 como control de final de bucle ya que los elementos quinto y siguientes no están inicializados. Si hubiésemos puesto el valor 10 en lugar 4 en el bucle **for** la salida del programa podría ser:

```
Hola!@#$$%&
```

Las cadenas de caracteres suelen tener una longitud variable; es decir, aunque tengamos memoria reservada para almacenar nombres de 10 caracteres unas veces tendremos nombres largos y otras veces tendremos nombres cortos. A la hora de imprimir, copiar o comparar estos nombres es necesario conocer su longitud.

8.3.1. Cadenas de caracteres de longitud variable

Básicamente existen dos métodos para manejar cadenas de caracteres de longitud variable dentro de un ordenador. El primero es almacenar un valor entero que nos indique cuántas letras están escritas realmente (en el ejemplo anterior este valor sería 4). Este método, análogo al recomendado para vectores de números, requiere almacenar en algún lado el valor de la longitud, lo que puede resultar algo incómodo. Una solución sería utilizar el primer elemento del vector de **char** para almacenar la longitud, de esta manera el vector almacena la longitud y los caracteres conjuntamente. Este método no se utiliza en C, aunque se utiliza en otros lenguajes de programación, y tiene como principal inconveniente que existe un límite máximo en las longitudes de las cadenas de caracteres (típicamente 256 caracteres).

La segunda manera de manejar cadenas de caracteres de longitud variable es utilizar un carácter especial para indicar el final de la cadena. Este es el método utilizando normalmente en el lenguaje C y existen un montón de funciones dentro de la biblioteca estándar de C para manejar cadenas de caracteres basadas en este método. Las características de este método son:

- En un vector de tamaño n sólo caben $n - 1$ letras, ya que el carácter de final de cadena ocupa una posición.
- No existe límite en la longitud de la cadena, el único límite viene dado por el tamaño del vector. Si se define un vector de tamaño 1000, como por ejemplo **char** nombre[1000], se pueden almacenar sin ningún problema hasta 999 caracteres.
- En cada posición de la cadena se puede almacenar cualquier carácter, a excepción del carácter definido como final de cadena.

Dado que existe un carácter que indica el final de la cadena, no es posible almacenar en ella cualquier tipo de información ya que dicho carácter no puede formar parte de la cadena. Para evitar el mayor número de problemas, en el lenguaje C se ha elegido como carácter de final de cadena el carácter número 0 de la tabla ASCII (también llamado NULL o carácter nulo y representado mediante `'\0'`). Existen muchos caracteres de uso poco frecuente, como por ejemplo el carácter %, que podrían haberse utilizado para marcar el final de la cadena. Sin embargo estos caracteres darían problemas antes o después. Si por ejemplo se desea realizar un programa para imprimir octavillas con la frase “0,7% YA” para pedir mayor justicia (en este mundo enfermo en el que el 20% de la población de los países del “norte” posee el 80% de la riqueza) el ordenador sólo imprimiría “0,7”, lo cual perturbaría un poco el mensaje original. El carácter seleccionado en C, `'\0'`, no da problemas porque nunca se utiliza como parte de una cadena de caracteres, ya que no es un carácter imprimible. Usando esta técnica, el programa anterior puede escribirse de la siguiente manera:

```

/*****
Programa: cadenas2.c
Descripción: Inicializa un vector de char y luego escribe
             el contenido del vector carácter por carácter hasta llegar
             al carácter NULL.
Revisión 0.0: 14/ABR/1998
Autor: Rafael Palacios
*****/
#include <stdio.h>

int main(void)
{
    char nombre[10];
    int i;

    /* inicialización */
    nombre[0] = 'H';
    nombre[1] = 'o';
    nombre[2] = 'l';
    nombre[3] = 'a';
    nombre[4] = '\0';

    /* imprimir valores */
    i=0;
    while(nombre[i] != '\0') {
        printf("%c", nombre[i]);
        i++;
    }
    return 0;
}

```

Hay que destacar que este bucle escribe los caracteres en pantalla siempre que éstos sean diferentes del carácter `'\0'`. Cuando el contador `i` alcance la posición en la que está almacenado el carácter `'\0'`, el bucle termina sin hacer el `printf`. Además este bucle es válido incluso si la cadena de caracteres está vacía; es decir, si el primer carácter es directamente el carácter `'\0'` entonces el bucle no arranca y no

se escribe nada.

Por haber elegido el carácter 0 como indicador de final de cadena, y gracias a que las condiciones de los bucles sólo se consideran falsas cuando valen 0, la condición del **while** puede ser el propio carácter `nombre[i]`. Sin embargo el código resultante queda menos claro.

8.3.2. Funciones estándar para manejar cadenas de caracteres

Existen una serie de funciones en la biblioteca estándar del lenguaje C que facilitan el manejo de cadenas de caracteres, siempre que estas cadenas cumplan la norma de estar terminadas con un carácter `'\0'`. A continuación se presentan algunas de estas funciones, con una breve explicación y un pequeño programa que muestra su comportamiento. Es más interesante comprender el funcionamiento de los programas que aprender lo que hace cada una de las funciones (dado que esto figura en el manual del compilador). Entendiendo estos programas no es difícil crear nuevas funciones específicas que no se incluyen en la biblioteca estándar, como por ejemplo funciones para cambiar minúsculas por mayúsculas, quitar los espacios, quitar las tildes, etc.

Todas las funciones que se ven a continuación están declaradas en el archivo cabecera `string.h`. Por lo tanto hay que incluir este archivo al principio del programa para que el compilador pueda comprobar que las estamos utilizando correctamente. Sin embargo todas estas funciones son parte de la biblioteca estándar y no hace falta añadir ninguna biblioteca al enlazar el programa.

Longitud de una cadena: `strlen`, *string length*

La función `strlen` calcula la longitud de una cadena de caracteres. Este cálculo se realiza recorriendo la cadena hasta encontrar el carácter `'\0'`.

```
/******
```

Programa: cadenas3.c

Descripción: Inicializa un vector de char y luego escribe el contenido del vector carácter por carácter.

Utiliza la función `strlen` para obtener la longitud de la cadena de caracteres.

Revisión 0.0:

```
*****/
```

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

```
#include <string.h>
```

```
int main(void)
```

```
{
```

```
    char nombre[10];
```

```
    int i;
```

```
    int longitud;
```

```
    /* inicialización */
```

```
    nombre[0] = 'H';
```

```
    nombre[1] = 'o';
```

```
    nombre[2] = 'l';
```

```
    nombre[3] = 'a';
```

```

nombre[4] = '\0';

/* cálculo la longitud */
longitud=strlen(nombre);

/* imprimir valores */
for(i=0; i<longitud; i++) {
    printf("%c", nombre[i]);
}
printf("\n");
return 0;
}

```

Nótese que en la llamada a `strlen` se pone directamente el nombre de la variable, sin los corchetes `[]`.

Este programa es equivalente al siguiente, que no utiliza `strlen`.

```

/*****
Programa: cadenas4.c
Descripción: Inicializa un vector de char y luego escribe
              el contenido del vector carácter por carácter.
              Calcula la longitud de la cadena de caracteres mediante
              un bucle while.
Revisión 0.0: 14/ABR/1998
Autor: Rafael Palacios
*****/
#include <stdio.h>

int main(void)
{
    char nombre[10];
    int i;
    int longitud;

    /* inicialización */
    nombre[0] = 'H';
    nombre[1] = 'o';
    nombre[2] = 'l';
    nombre[3] = 'a';
    nombre[4] = '\0';

    /* cálculo la longitud */
    longitud=0;
    while(nombre[longitud] != 0) {
        longitud++;
    }

    /* imprimir valores */
    for(i=0; i<longitud; i++) {
        printf("%c", nombre[i]);
    }
}

```

```

    printf("\n");
    return 0;
}

```

Copiar o inicializar: strcpy, string copy

Esta función copia cadenas de caracteres. Es importante recordar que no es válido asignar una constante a un vector y por lo tanto las siguientes instrucciones son incorrectas:

```
char nombre[100];
```

```
vec = 4.5;          /* Error si vec es vector*/
nombre = "Pepito"; /* Error */
```

Esta función no verifica que el tamaño del vector de destino sea suficientemente grande para guardar la cadena que se quiere copiar, por lo que el programador será responsable de garantizar que esto ocurra para no producir fallos indeseables en el programa. El formato de la llamada a strcpy es el siguiente:

```
strcpy(destino, origen);
```

Es decir, tiene una sintaxis equivalente a la asignación normal de variables, del tipo:

```
destino = origen; /* Error. Hay que usar strcpy en lugar de = */
```

Esta función es muy útil para inicializar cadenas de caracteres, poniendo una constante entre comillas dobles como argumento *origen*. Esta inicialización evita tener que definir carácter por carácter el vector. Los siguientes principios de programa son equivalentes:

int main(void)	#include <string.h>
{	int main(void)
char nombre[10];	{
	char nombre[10];
/* inicialización */	/* inicialización */
nombre[0] = 'H';	strcpy(nombre, "Hola");
nombre[1] = 'o';	
nombre[2] = 'l';	
nombre[3] = 'a';	
nombre[4] = '\0';	
:	:
:	:
:	:

Es importante destacar que la segunda versión incluye el archivo `string.h` para poder utilizar la función `strcpy` y que no hace falta escribir el carácter `'\0'` porque está implícito en todas las cadenas de caracteres constantes (aquellas que se escriben directamente en el código encerradas entre comillas).

El siguiente programa explica el comportamiento de la función `strcpy` para copiar la variable `a` en la variable `dir` (ambas son vectores de **char** de tamaño suficiente).

```

i = 0;
while(a[i] != '\0'){
    dir[i] = a[i];
    i++;
}
dir[i]=a[i]; /* Copio el NULL */

```

Este bucle **while** copia carácter a carácter la variable *a* en la variable *dir*, borrando la información que *dir* pudiese tener previamente. La última línea de este código es fundamental, ya que garantiza que la variable *dir* tenga carácter de final de cadena. Cuando *a[0]* vale `'\0'` el bucle no arranca y la última línea hace *dir[0]=a[0]* por lo tanto también copia el carácter NULL. Cuando la cadena *a* no está vacía, el bucle copia carácter por carácter todas las letras hasta llegar a NULL (carácter que no copia) entonces la última línea copia el NULL en su sitio.

Comparación de cadenas: **strcmp**, *string compare*

Esta función compara cadenas de caracteres. Principalmente se utiliza para ver si dos nombres son iguales o para ordenarlos alfabéticamente. La función devuelve un valor de tipo **int** que puede ser cero, negativo o positivo. Por ejemplo la comparación de las variables *nombre1* y *nombre2* puede dar los siguientes resultados:

```
i = strcmp(nombre1, nombre2);
```

La variable entera *i* tomará los siguientes valores:

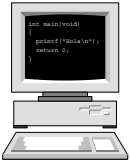
- *i=0* si *nombre1* es igual a *nombre2*. Esto significa que todos los caracteres del vector *nombre1* desde el índice 0 hasta alcanzar el carácter NULL (incluyendo a éste) son iguales a los correspondientes caracteres del vector *nombre2*. Pero puede ocurrir que el resto de los caracteres (desde el carácter NULL hasta el final del vector) sean diferentes, lo cual da lo mismo, ya que estos caracteres no forman parte de la cadena. También puede ocurrir que los vectores sean de diferente tamaño, ya que sólo se comparan los caracteres anteriores a NULL.
- *i<0* si *nombre1* es menor que *nombre2*, entendiendo que *nombre1* se ordena alfabéticamente por delante de *nombre2*. En este caso hay que tener en cuenta que la ordenación sigue los criterios de la tabla ASCII; es decir los números son menores que las letras, las letras mayúsculas son menores que las minúsculas y las letras con tilde son mayores que cualquier otro carácter normal. Por ejemplo: 3Com < Andalucía < Sevilla < boreal < nórdico < África
- *i>0* si *nombre1* es mayor que *nombre2*.

Un programa que tiene un funcionamiento equivalente a `strcmp(a,b)` es:

```

i = 0;
resultado = 0;
while(a[i]!='\0' && resultado==0) {
    if (a[i]<b[i]) {
        resultado = -1;
    } else if (a[i]>b[i]) {
        resultado = 1;
    }
    i++;
}

```



```

    }
    if (resultado==0) { /* el while terminó porque a[i] vale '\0' */
        if (b[i]!='\0') { /* la cadena b es más larga que a */
            resultado = -1;
        }
    }
}

```

Realice el ejercicio 1.

Lectura y escritura de cadenas de caracteres

Existen varias funciones dentro de la librería estándar que permiten leer o escribir cadenas de caracteres directamente, sin necesidad de utilizar bucles que lo hagan carácter a carácter.

La función `printf` puede utilizarse para visualizar cadenas de caracteres, no hace falta ninguna otra función especial. Cuando se pone el código `%s` dentro de la especificación de formato, la función `printf` se encarga de escribir todos los caracteres del vector de **char** hasta encontrar el carácter NULL. Así, si para escribir una cadena carácter a carácter haríamos:

```

printf("Nombre: ");
i=0;
while(nombre[i]) {
    printf("%c", nombre[i]);
}
printf("\n");

```

Para escribirla de una vez se puede hacer como en la siguiente instrucción, que es equivalente al programa anterior:

```
printf("Nombre: %s\n", nombre);
```

Es evidente que esta segunda forma de imprimir la cadena es mucho más sencilla.

Nótese que ahora no se ponen corchetes, ya que queremos pasar a la función `printf` el vector de caracteres `nombre` completo, no uno de sus elementos.

Análogamente las cadenas de caracteres pueden leerse de teclado utilizando `scanf` con formato `%s`:

```

printf("Dame el nombre: ");
scanf("%s", nombre);

```

En este caso particular, la variable no lleva el `&` delante. Además también conviene destacar que la función `scanf` está diseñada para leer palabras, es decir que leerá letra a letra hasta encontrar un espacio, una coma, un tabulador o un cambio de línea.

Para leer frases completas existe la función `gets`, que lee todos los caracteres hasta encontrar un cambio de línea:

```

char frase[10];

gets(frase);

```

Funciones seguras para el manejo de cadenas de caracteres

Las funciones como `gets` o `scanf` no verifican que el número de caracteres leídos quepan en la cadena destino. Esto es muy utilizado para atacar programas informáticos y producir problemas de seguridad. La técnica utilizada se denomina desbordamiento de *buffer* (*buffer overflow*) y consiste en introducir una cadena más larga que

el tamaño del vector donde está previsto almacenarla para sobrescribir la memoria adyacente y provocar así la ejecución de un programa diseñado por el atacante para hacerse con el control del ordenador.

Para evitar que esto ocurra, existe una versión segura denominada `fgets` que impide que se introduzcan más caracteres de los que caben en la cadena. El mismo ejemplo anterior se puede escribir como:

```
char frase[10];

fgets(frase, 10, stdin);
```

El primer argumento es el nombre de la vector, el segundo es la longitud de éste y el tercero indica el dispositivo de lectura (`stdin` significa *standard input*, es decir la entrada estándar, que es el teclado). En su funcionamiento normal `fgets` lee una frase, formada por cualquier combinación de caracteres, y los va almacenando en el vector. Cuando se lee un `'\n'`, el cual se guarda en el vector, finaliza la lectura y se termina la cadena añadiendo un `'\0'` a continuación del `'\n'`. Si se introduce una frase demasiado larga, se activa la protección y una vez leídos los 9 primeros caracteres, se interrumpe la lectura y se añade el `'\0'` para terminar la cadena.

Por lo tanto, en un vector de tamaño 10 leído con `fgets`, típicamente la cadena terminará con un `'\n'` seguido de un `'\0'`, aunque excepcionalmente sólo tendrá el `'\0'` en la última posición (`nombre[9]` en el ejemplo). Para solucionar esto y que la cadena leída tenga sólo un `'\0'` al final, un buen código de lectura de cadenas es el siguiente:

```
char nombre[10];

fgets(nombre, 10, stdin);
if(nombre[strlen(nombre)-1] == '\n'){
    nombre[strlen(nombre)-1] = '\0';
}
```

Existen funciones análogas a `fgets` para el manejo de cadenas de caracteres. Por ejemplo, para comparar y copiar cadenas de caracteres que pueden tener distintos tamaños, existen las funciones `strncmp` y `strncpy`. Los prototipos de ambas funciones son:

```
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *dest, const char *orig, size_t n);
```

El funcionamiento es análogo al de las funciones `strcmp` y `strcpy` discutidas anteriormente, salvo que sólo se comparan/copian los `n` primeros caracteres de las cadenas.

Si algún programador torpe ha olvidado poner el `'\0'` al final de la cadena `orig` y utilizamos `strcpy` el programa fallará, pues continuará copiando después del final de la cadena y no parará hasta que encuentre en la memoria un `'\0'`. Si se utiliza `strncpy` tenemos mayor protección, pues en este caso se parará después de copiar `n` caracteres.

8.4. Matrices

La definición y utilización de matrices es análoga al caso de los vectores, salvo que ahora tenemos dos dimensiones. Una matriz `mat` de tamaño 3x5 para números enteros se define como:

```
int mat[3][5];
```



Realice el ejercicio 2.

El acceso a los 15 elementos de esta matriz se realiza siempre utilizando dos índices que también se escriben entre corchetes y que también empiezan por cero. Por lo tanto los elementos de la matriz `mat` tiene los siguientes nombres:

```
mat[0][0]  mat[0][1]  mat[0][2]  mat[0][3]  mat[0][4]
mat[1][0]  mat[1][1]  mat[1][2]  mat[1][3]  mat[1][4]
mat[2][0]  mat[2][1]  mat[2][2]  mat[2][3]  mat[2][4]
```

Al igual que en el caso de los vectores, no existen funciones especiales para imprimir ni operar con matrices por lo que generalmente se utilizan bucles **for**. En el caso de las matrices de dos dimensiones, como la del ejemplo anterior, se utilizan dos bucles **for** anidados para recorrer todos los elementos. El siguiente programa define una matriz de 3x5 e inicializa a 7 todos sus elementos:

```
/******
Programa: matriz.c
Descripción: Inicializa a 7 todos los elementos de una matriz
de enteros de 3x5
Revisión 0.0: 14/ABR/1998
Autor: Rafael Palacios
*****/
```

```
int main(void)
{
    int mat[3][5];
    int i,j;

    for(i=0; i<3; i++) {
        for(j=0; j<5; j++) {
            mat[i][j]=7;
        }
    }
    return 0;
}
```



Realice el ejercicio 9

En la práctica ocurre lo mismo que con los vectores: es habitual declarar matrices suficientemente grandes y luego llenarlas parcialmente. Es recomendable en este caso declarar variables enteras auxiliares para conocer el número de filas y columnas efectivas de la matriz, tal como se muestra en el siguiente ejemplo:

```
double x[100][1000];

int x_filas = 3;
int x_columnas = 5;
```

8.5. Utilización de **#define** con vectores y matrices

Resulta muy útil definir los tamaños de vectores y matrices en función de parámetros, ya que luego se pueden modificar fácilmente sin tener que revisar todo el programa. Estos parámetros se definen con la instrucción **#define** que es una directiva del preprocesador, al igual que **#include**.

El programa anterior quedaría de la siguiente manera al utilizar un parámetro `N` para el número de filas y un parámetro `M` para el número de columnas.

```

/*****
Programa: matriz2.c
Descripción: Inicializa a 7 todos los elementos de una matriz
              de enteros de N por M.
              Se utiliza #define para definir el valor de N y M.
Revisión 0.0: 14/ABR/1998
Autor: Rafael Palacios
*****/

#define N 3
#define M 5

int main(void)
{
    int mat[N][M];
    int i, j;

    for(i=0; i<N; i++) {
        for(j=0; j<M; j++) {
            mat[i][j]=7;
        }
    }
    return 0;
}

```

Si se quiere modificar el programa para que trabaje con matrices de tamaño 30x50 basta con modificar las dos primeras líneas y volver a compilar. Todo el programa está escrito en función de los dos parámetros, tanto el tamaño de las matrices como los límites de los bucles, y por lo tanto es válido para cualquier valor de N y M.

Ya se dijo en el capítulo 3 que los nombres de variables suelen escribirse en minúsculas, para mayor claridad al leer el programa. Para diferenciar las variables de los parámetros, éstos suelen escribirse en mayúsculas, como se ha hecho en el ejemplo.

En caso de querer preguntar al usuario el tamaño de las matrices con las que quiere calcular, habrá que definir tamaños fijos suficientemente grandes (ya se verá más adelante cómo se definen matrices y vectores de tamaño variable). Al utilizar tamaños fijos es imprescindible comprobar, antes de iniciar los cálculos, que el valor introducido por el usuario no sea mayor que el tamaño asignado para las matrices. El compilador no comprueba si se accede a elementos que se salen del vector por lo que el programa puede fallar estrepitosamente sin darnos muchas pistas. El siguiente ejemplo es un programa para multiplicar una matriz por un vector, donde los vectores y matrices se definen con dimensiones grandes, pero normalmente sólo se rellenan parcialmente.

```

/*
Programa: Matriz1
Descripción: Calcula el producto de una matriz por un vector
Revisión 0.2: mar/98, ago/05
Autor: Rafael Palacios Hielscher
*/
#include <stdlib.h>
#include <stdio.h>

```



```

#define N 100
#define M 100

int main(void)
{
    double mat[N][M]; /* Matriz */
    double vec[M];    /* Vector */
    int i; /* contador */
    int j; /* otro contador (necesario para recorrer la matriz) */
    int n; /* número de filas de la matriz */
    int m; /* número de columnas de la matriz, que es igual al
           número de elementos del vector */
    double tmp; /* variable temporal */

    do{
        printf("Número de filas? ");
        scanf("%d", &n);
        printf("Número de columnas? ");
        scanf("%d", &m);
        if (n>N || m>M) {
            printf("Error, este programa no vale para "
                   "tamaños tan grandes.\n");
        }
    }while (n>N || m>M);

    /**** Lectura de datos ****/
    /* matriz */
    for(i=0; i<n; i++) {
        for(j=0; j<m; j++) {
            printf("mat[%d][%d]= ", i, j);
            scanf("%lf", &mat[i][j]);
        }
    }
    /* vector */
    for(i=0; i<m; i++) {
        printf("vec[%d]= ", i);
        scanf("%lf", &vec[i]);
    }

    /**** Cálculos y Salida ****/
    for(i=0; i<n; i++) {
        tmp=0;
        for(j=0; j<m; j++) {
            tmp += mat[i][j]*vec[j];
        }
        printf("Resultado[%d]=%f\n", i, tmp);
    }
    return 0;
}

```

8.6. Paso de vectores, cadenas y matrices a funciones

En ocasiones es necesario que una función trabaje con vectores o matrices (o ambas cosas). En el capítulo anterior se ha comentado que los valores de las variables que se pasan como argumentos a una función se copian a las variables locales de dicha función. Sin embargo, si cada vez que se llamase a una función se realizara una copia del vector o matriz, el mecanismo de llamada sería tremendamente ineficiente en cuanto las matrices tuviesen una longitud apreciable. Debido a esto, en C no se copia el vector o la matriz cuando se pasan como argumento a una función, sino que se le dice en qué parte de la memoria está para que la función trabaje sobre el vector o la matriz **original**. Esta solución se conoce como paso de variables por **referencia** (o por argumento) a diferencia del mecanismo de variables normales que se denomina paso de variables por **valor**. El paso por referencia, aunque muy eficiente, puede ser muy peligrosa en manos de un programador inexperto, pues cada modificación que se realice en el vector o la matriz desde dentro de la función **modifica el vector o la matriz original**.

8.6.1. Paso de vectores

El prototipo típico de una función que acepta un vector como argumento es:

```
tipo_devuelto NombreFunción(tipo nombre_vector[], int longitud_vector);
```

Es decir, se ponen dos corchetes [] a continuación del nombre del vector. No es necesario especificar el tamaño del vector entre los corchetes porque la función es genérica y vale para cualquier tamaño. Por ello es casi obligatorio enviar la longitud del vector como un argumento adicional. Por ejemplo, una función que calcule el producto escalar de dos vectores de igual dimensión tendrá como prototipo:

```
double ProdEscalar(double vect1[], double vect2[], int dim);
```

Y la definición de la función, junto con un breve programa que ilustra su uso, es la siguiente:

```
1 /* Programa: Producto Escalar
   *
   3 * Descripción: Ejemplo de uso de la función ProdEscalar.
   *
   5 * Revisión 0.0: 15/04/1998
   *
   7 * Autor: El funcionario novato.
   */
9
10 #include <stdio.h>
11
12 /* prototipos de las funciones */
13
14 double ProdEscalar(double vect1[], double vect2[], int dim);
15
16 int main(void)
17 {
    double v1[3]={1, 2, 3};
```

```

19  double v2[3]={3, 2, 1};
    double escalar;

21      escalar =  ProdEscalar(v1, v2, 3);

23      printf("El producto escalar vale %g\n", escalar);
25      return 0;
    }

27  /* Función: ProdEscalar
28  *
29  *  Descripción: Devuelve el producto escalar de dos vectores de 3
30  *                elementos.
31  *
32  *  Argumentos: double vect1[]: primer vector.
33  *                double vect2[]: segundo vector.
34  *
35  *  Valor devuelto: double: El producto escalar de los dos vectores.
36  *
37  *  Revisión 0.0: 15/04/1998.
38  *
39  *  Autor: El funcionario novato.
40  */

43  double ProdEscalar(double vect1[], double vect2[], int dim)
    {
45      int i;
      double producto;

47      producto = 0;
49      for(i=0; i<dim; i++){
          producto += vect1[i]*vect2[i];
51      }

53      return producto;
    }

```

En este ejemplo se han introducido dos novedades: La primera es la inicialización de los vectores realizada en las líneas 18 y 19. Este tipo de inicialización de los vectores asigna al primer elemento del vector el primer elemento de la lista entre llaves, al segundo elemento el segundo valor y así sucesivamente hasta terminar la lista. Así pues la línea 18 es equivalente a las líneas:

```

v1[0] = 1;
v1[2] = 2;
v1[3] = 3;

```

pero mucho más compacta como habrá observado el lector.

La segunda novedad es la manera de llamar a una función pasándole un vector. Esto se ha mostrado en la línea 22, en la que se llama a la función `ProdEscalar`. Como puede observar para pasar los vectores a la función se escribe su nombre sin corchetes en la lista de argumentos: `ProdEscalar(v1, v2, 3)`.

8.6.2. Paso de cadenas a funciones

Todo lo que se acaba de discutir sobre el paso de vectores a funciones es aplicable al paso de cadenas a funciones, pues como recordará las cadenas de caracteres no eran más que vectores de tipo **char**. La única diferencia es que los vectores de caracteres utilizan el carácter `'\0'` para indicar el final de la cadena y por tanto no es necesario indicar su longitud al llamar a la función. Sin embargo en vectores de tipo numérico, sí se indica la longitud. Veamos dos ejemplos:

- Prototipo de una función que cuenta el número de valores positivos de un vector:

```
int Positivos(double vec, int dim);
```

- Prototipo de una función que cuenta el número de vocales de una cadena de caracteres:

```
int Vocales(char cadena);
```

8.6.3. Paso de matrices a funciones

El paso de matrices es un poco distinto al de los vectores. En ellas es necesario indicar todas las dimensiones de la matriz salvo la primera. Adicionalmente es necesario indicar las dimensiones reales en el caso de no aprovechar completamente la matriz. Por ejemplo si queremos realizar una función para inicializar parcialmente matrices de 30x50, el prototipo de la función será:

```
void InicializaMatriz(float matriz[][50], int n, int m);
```

y su definición es:

```
void InicializaMatriz(float matriz[][50], int n, int m)
{
    int i; /* índices para recorrer la matriz */
    int j;

    for(i=0; i<n; i++){
        for(j=0; j<m; j++){
            mat[i][j] = 0;
        }
    }
}
```

y para llamarla desde otra función se escribe el nombre de la matriz sin corchetes de la misma forma que se hace con los vectores:

```
...
float mat[30][50];
int mat_filas = 0, mat_columnas = 0;
...
printf("Introduzca el número de filas: ");
scanf("%d", &mat_filas);
printf("Introduzca el número de columnas: ");
scanf("%d", &mat_columnas);
```

```
InicializaMatriz(mat, mat_filas, mat_columnas);
...
```

Por último cabe destacar dos cosas:

- Que el compilador no realiza ningún tipo de comprobación sobre la coincidencia de las dimensiones de la matriz que se le pasa a la función con las que ésta espera. Es responsabilidad del programador que esto ocurra para evitar fallos catastróficos en el programa.
- Al contrario de lo que ocurriría con los vectores, no se pueden realizar funciones totalmente genéricas que trabajen con matrices de cualquier dimensión, pues el compilador necesita saber algunas dimensiones de la matriz para acceder a ella. Construyendo matrices dinámicas, como se verá en el capítulo de punteros, sí es posible construir funciones genéricas para trabajar con matrices.

8.7. Recomendaciones y advertencias

- El programador es responsable de que nunca se pueda producir un acceso a un elemento no válido de un vector. Si esto ocurriese el resultado sería impredecible.
- Para el manejo de cadenas de caracteres se recomienda utilizar las versiones seguras de las funciones, en especial realizar la lectura desde teclado con `fgets`.
- En las llamadas a funciones es recomendable pasar como argumento las dimensiones de vectores y matrices. No es buena costumbre suponer que los tamaños son conocidos y que los vectores y matrices están completamente llenos. La única excepción son las cadenas de caracteres, que utilizan el carácter `'\0'` para indicar el final y por tanto no requieren una variable entera auxiliar para conocer su longitud.
- Las variables normales que son argumentos de funciones se copian en el momento de realizar la llamada (paso por valor) y por tanto no se pueden modificar desde la función. En cambio los vectores y matrices se pasan por referencia y sí se pueden modificar en la función. En el capítulo 9 se estudiará todo esto en mayor detalle.
- Un error muy habitual al trabajar con cadenas de caracteres es intentar copiar cadenas mediante el operador de asignación `'='`. Tenga en cuenta que las cadenas son vectores y que por tanto no pueden copiarse de esta manera. Afortunadamente existen las funciones `strcpy` y `strncpy` que permiten copiar (o inicializar) cadenas fácilmente. Es decir, en lugar de hacer:

```
cadena = "hola"; /* ERROR GARRAFAL */
```

Haga esto:

```
strcpy(cadena, "hola"); /* Así sí */
```

- El segundo error más común al trabajar con cadenas de caracteres es construir una cadena carácter por carácter y olvidar “cerrarla” con un `'\0'`.

8.8. Resumen

En este capítulo se ha visto como declarar variables para almacenar conjuntos de valores de la misma naturaleza. Estas variables, denominadas vectores y matrices, facilitan el tratamiento de grandes volúmenes de datos y permiten crear un código más claro y compacto. Los bucles estudiados en el capítulo 5 serán nuestros aliados a la hora de manejar estas estructuras de datos con facilidad.

Un caso particular de los vectores son las cadenas de caracteres, que se utilizan para manipular textos dentro del ordenador. En este capítulo también se han estudiado una serie de funciones básicas para el manejo de estas cadenas de caracteres.

8.9. Ejercicios

1. Comprobar el funcionamiento del programa mostrado en la sección 8.3.2 (página 101) para distintos ejemplos. Comprobar si funciona para palabras como “fulmina” y “fulminante”, “alumno” y “alumna”, etc. ¿Da lo mismo en qué variables (a o b) se almacenen las palabras?

2. Escribir un programa que lea una cadena con `scanf`, con `gets` y con `fgets`, comparando los resultados. Introducir una cantidad de caracteres mayor que la dimensión de la cadena y ver si se produce algún resultado anómalo.

3. Escribir una **función** que devuelva un 1 si la cadena que se le pasa como argumento contiene un número y un 0 en caso contrario. El prototipo de la función será:

```
int EsNumero(char cadena[]);
```

Se considera que una cadena contiene un número si esta formada por los caracteres '0' al '9', el punto '.' o el carácter espacio ' '. Por ejemplo la cadena "123.34" contiene un número mientras que la cadena "hola123" no.

4. Escribir una función para buscar la primera posición de un carácter c dentro de una cadena de caracteres s. El prototipo de la función es:

```
int Buscar(char c, char s[]);
```

La función devuelve un entero que es la posición de la primera ocurrencia del carácter c en la cadena s, o bien el valor -1 si s no contiene el carácter c.

5. Repetir el ejercicio anterior pero escribiendo una función que devuelva la posición del número num dentro del vector vec. En este caso el prototipo será:

```
int BuscarEntero(int num, int vec[], int dim);
```

6. Escribir una función para buscar la primera coincidencia entre los caracteres de dos cadenas s1 y s2. El prototipo de la función es:

```
int BuscaCoinci(char s1[], char s2[]);
```

La función devuelve un entero que es la posición de la primera coincidencia entre las cadenas s1 y s2, o bien el valor -1 si ambas cadenas no tienen caracteres coincidentes en ninguna posición. Por ejemplo si la cadena s1 es “hola” y la cadena s2 es “Halo”, la función devolverá un 2, que es la posición en la que ambas cadenas tienen un mismo carácter (la 'l'). Si la cadena s2 contiene “adiós”, la función devolverá un -1.

7. Escribir una función que tome una cadena origen y la copie en una cadena destino pero convirtiendo los caracteres de tabulación ('\t') en un número de espacios especificado en el argumento `n_espacios`.
8. Escriba un programa para probar la función escrita en el ejercicio anterior.
9. Escribir un programa que pregunte el tamaño de una matriz cuadrada, pida todos sus elementos y calcule la traspuesta. Luego debe mostrar el resultado ordenadamente (en forma de matriz).
10. Realizar una función para calcular el determinante de una matriz de 3×3 . El prototipo de la función será:

```
double Det(double mat[][3]);
```

11. Basándose en la función anterior, escriba un programa que pida por teclado los elementos de la matriz e imprima el resultado.
12. Escribir una función para multiplicar dos matrices de 3×3 . El prototipo será:

```
void ProdMat(double A[][3], double B[][3], double Res[][3]);
```

13. Basándose en la función anterior, escriba un programa que calcule el producto de dos matrices a y b, introducidas por el teclado, e imprima el resultado por pantalla.
14. Un programa debe trabajar con un vector de números reales declarado de la siguiente manera:

```
#define N 100
...
double a[N]; /* vector de N elementos */
```

Se quieren ir leyendo por teclado los datos del vector hasta que el usuario introduzca el valor 0 o hasta que el vector se llene totalmente. Escribir la parte del programa que realiza la **lectura de datos**.

15. Un programa debe inicializar un vector de números reales declarado de la siguiente manera:

```
#define N 100
...
double a[N]; /* vector de N elementos */
```

El programa debe preguntar al usuario el número de elementos del vector que se desean inicializar (`n_e`), el valor inicial (`v_ini`) y un incremento (`inc`). A continuación se inicializarán los primeros `n_e` elementos del vector (si `n_e` fuese mayor que `N` se inicializarán sólo los `N` elementos del vector). La inicialización se realizará según una serie aritmética de valor inicial `v_ini` y de incremento `inc`. Por ejemplo si `n_e` vale 4, `v_ini` vale 2 e `inc` vale 1.5, se inicializarán los cuatro primeros elementos del vector con los valores 2, 3.5, 5, 6.5, dejándose los demás elementos sin inicializar. Escribir el **programa** completo.

16. Dentro de un programa se necesita calcular la suma de todos los elementos de un vector de **double** llamado `vec`. Escribir una función que devuelva el valor de la suma de los elementos del vector. La longitud del vector es variable, pero se proporciona el dato como argumento. El prototipo de la función debe ser:

```
double SumaVec(double vec[], int n);
```

Escribir también cómo se haría la llamada desde el programa principal a la función `SumaVec`.

17. Dentro de un programa se necesita calcular la suma de todos los elementos de una matriz de **double** llamada `mat_a`. La matriz se declara de 100x100 elementos para que haya sitio de sobra para las matrices que introducirá el usuario, que serán de dimensión `filas` x `columnas`. Escribir una **función** que devuelva el valor de la suma de todos los elementos válidos de la matriz `mat_a` conociendo que el número de filas con datos válidos está almacenado en el parámetro `filas` y el número de columnas con datos válidos en `columnas`.

El prototipo de la función es:

```
double SumaMat(double a[][100], int filas, int columnas);
```

18. Dentro de un programa se necesita calcular la media de todos los elementos de una matriz cuadrada de **double** llamada `mat_a`. La matriz se declara de 100x100 elementos para que haya sitio de sobra para las matrices que introducirá el usuario, que serán de dimensión `dim` x `dim`. Escribir una **función** que devuelva el valor de la media de todos los elementos válidos de la matriz `mat_a`.

El prototipo de la función es:

```
double MediaMat(double mat_a[][100], int dim);
```


CAPÍTULO 9

Punteros

9.1. Introducción

Los punteros son un tipo de variable un poco especial, ya que en lugar de almacenar valores (como las variables de tipo **int** o de tipo **double**) los punteros almacenan direcciones de memoria. Utilizando variables normales sólo pueden modificarse valores, mientras que utilizando punteros pueden manipularse direcciones de memoria o valores. Pero no se trata simplemente de un nuevo tipo de dato que se utiliza ocasionalmente, sino que los punteros son parte esencial del lenguaje C. Entre otras ventajas, los punteros permiten:

- Definir vectores y matrices de tamaño variable, que utilizan sólo la cantidad de memoria necesaria y cuyo tamaño puede reajustarse dinámicamente.
- Que las funciones devuelvan más de un valor.
- Definir estructuras de datos más complejas como listas de datos o árboles.

9.2. Declaración e inicialización de punteros

Los punteros se declaran igual que las variables normales, pero con un asterisco (*) delante del nombre de la variable. Por ejemplo:

```
int a;  
int *pa;
```

En este ejemplo la variable *a* es de tipo entero y puede almacenar valores como 123 ó -24, mientras que la variable *pa* es un puntero y almacena direcciones de memoria. Como un puntero es un tipo de variable un poco especial, para evitar confusiones es muy conveniente que los nombres de los punteros empiecen con la letra *p*, ya que así tendremos presente, nada más ver el nombre de la variable, que ésta es un puntero.

Aunque todos los punteros almacenan direcciones de memoria, existen varios tipos de puntero dependiendo del tipo de dato al que apuntan. Por ejemplo:

```
int *pi;  
double *pd;
```

Tanto *pi* como *pd* son punteros y almacenan direcciones de memoria, por lo tanto almacenan datos del mismo tipo y ocupan la misma cantidad de memoria, es decir, **sizeof(pi) == sizeof(pd)**. La única diferencia es que el dato almacenado en la dirección de memoria contenida en el puntero *pi* es un entero, mientras que el dato almacenado en la dirección de memoria contenida en *pd* es un **double**. Se dice por

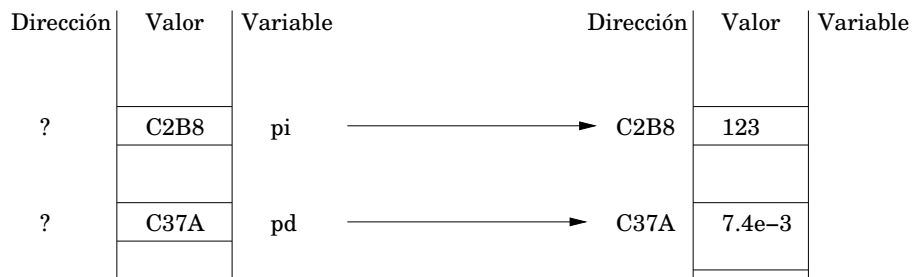


Figura 9.1: Puntero a entero y puntero a double

lo tanto que pi “apunta” a un entero (puntero a entero) mientras que pd “apunta” a un **double** (puntero a **double**). El compilador necesita conocer el tipo de dato al que apunta un puntero para poder operar con él correctamente: recuerde que no se suma igual un número entero que uno en coma flotante. De ahí que en la declaración del puntero haya que especificar el tipo de dato al que va a apuntar.

En la figura 9.1 aparece un ejemplo en el que pi apunta a una dirección de memoria (0xC2B8) donde se encuentra almacenado el valor entero 123, y pd apunta a otra dirección de memoria (0xC37A) donde se encuentra almacenado el valor 7.4e-3.

9.2.1. El operador &

Igual que las variables normales tienen un valor inicial desconocido después de su declaración, también los punteros almacenan inicialmente una dirección desconocida; es decir, apuntan a una dirección aleatoria que puede incluso no existir. Aunque se puede asignar un valor de dirección a un puntero, esto no es nada aconsejable ya que sería una casualidad que en la posición de memoria que se asigne exista un valor del tipo correspondiente al puntero y que además no se esté usando por otro programa. Si se escribe:

```
int *pi;
pi=0xC2B8; /* ojo */
```

sería una casualidad que en la dirección de memoria 0xC2B8 hubiese un valor entero. Por lo tanto en los próximos ejemplos vamos a hacer que nuestros punteros apunten a datos correctos mediante la declaración de variables auxiliares. Esto es algo que no parece muy útil y de hecho no se utiliza en ningún programa real, pero vale para explicar el funcionamiento de los punteros.

El operador & se utiliza para obtener la dirección de memoria de cualquier variable del programa. Si nuestro programa tiene una variable entera que se llama i, entonces podemos obtener la dirección de memoria que el compilador ha preparado para almacenar su valor escribiendo &i.

```
int i;      /* variable entera */
int *pi;    /* puntero a entero */

i=78;      /* inicializo i */
pi=&i;     /* pi apunta a i */
```

Al aplicar el operador & a la variable i no se obtiene el valor entero almacenado en ella (78) sino la dirección de memoria en donde se encuentra dicho valor (por ejemplo

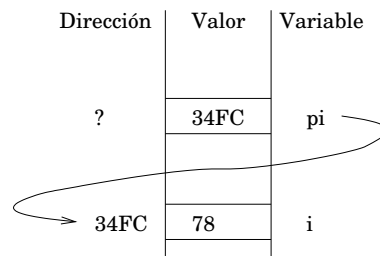


Figura 9.2: Puntero a entero

0x34FC). Por lo tanto `&i` es un puntero a entero porque es una dirección de memoria donde se haya un entero. Suponiendo que el compilador reserva la posición de memoria 0x34FC para la variable `i`, la ejecución de las instrucciones anteriores llevaría a la organización de memoria que se muestra en la figura 9.2.

El compilador dará un aviso si se intenta realizar una asignación en la que no se corresponden los tipos, por ejemplo al asignar `&i` a un puntero a **double**, ya que `&i` devuelve un puntero a **int**. Este tipo de comprobaciones del compilador evita muchos errores de programación. Por tanto hay que estar muy atento a los mensajes del compilador y hay que activar todos los avisos. Asignaciones entre dos punteros también son válidas siempre que los dos punteros sean del mismo tipo.

Por ejemplo, si se compila el programa siguiente:

```
#include <stdio.h>
2
int main(void)
4 {
    int i;
6    double *pd;

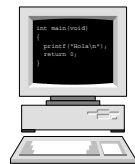
8    i = 27;

10   pd = &i; /* ERROR */

12   printf("%lf\n", *pd * 2);
    return 0;
14 }
```

el compilador (gcc en este caso¹) generará el siguiente aviso:

PruPunt_2.c:10: warning: assignment from incompatible pointer type
donde se avisa que en la línea 10 se asigna al puntero `pd` la dirección de un dato que no es compatible con su tipo. A pesar de este aviso se genera un programa que si se ejecuta imprime por pantalla el valor -3.997604 en lugar de 54.



9.2.2. El operador *

Hasta ahora los punteros sólo han sido variables para guardar direcciones de memoria, pero esto sería poco útil si no pudiésemos manipular el valor almacenado en

Realice el ejercicio 1.

¹Cualquier otro compilador generará un aviso parecido.

dichas posiciones de memoria. El operador unario `*`, llamado **operador de indirección**, permite acceder al valor por medio del puntero.

```
int i;    /* variable entera */
int *pi; /* puntero a entero */

pi=&i;    /* pi apunta a i */
*pi=78;  /* equivale a hacer i=78; */
```

La asignación `*pi=78` introduce el valor entero 78 en la posición de memoria almacenada en el puntero `pi`. Como previamente se ha almacenado en `pi` la dirección de memoria de la variable `i`, la asignación equivale a dar el valor 78 directamente a la variable `i`. Es decir, tras la asignación `pi=&i` disponemos de dos maneras de manipular los valores enteros almacenados en la variable `i`: directamente mediante `i` o indirectamente mediante el puntero `pi`. De momento esto no parece muy útil pero es importante que quede claro.

Advertencias

- El operador `&` sólo puede aplicarse a una variable normal y significa: “dirección de memoria en donde se encuentra la variable”. No tiene mucho sentido poner `&` delante de un puntero.
- El operador `*` sólo puede aplicarse a una variable de tipo puntero y significa: “valor almacenado en la dirección de memoria a la que apunta el puntero”.

9.3. Operaciones con punteros

Una vez entendido lo que es un puntero veamos las operaciones que pueden realizarse con estas variables tan especiales. En primer lugar ya se ha visto que admiten la operación de asignación para hacer que apunten a un lugar coherente con su tipo. Pero también admite operaciones de suma y diferencia que deben entenderse como cambios en la dirección a la que apunta el puntero. Es decir, si tenemos un puntero a entero que almacena una determinada dirección de memoria y le sumamos una cantidad, entonces cambiará la dirección de memoria y el puntero quedará apuntando a otro sitio. Las dos formas de manipular el puntero son:

```
*pi+=8;
pi+=8;
```

La primera línea suma 8 al entero al que apunta `pi` y por lo tanto el puntero sigue apuntando al mismo sitio y es el valor entero el que cambia. La segunda línea suma 8 posiciones de memoria a la dirección almacenada en el puntero, por lo tanto éste pasa a apuntar a otro sitio. (ver figura 9.3). De nuevo hacer que el puntero pase a apuntar a un lugar desconocido vuelve a ser peligroso y habrá que tener cuidado con este tipo de operaciones.

Es muy importante distinguir entre las operaciones con punteros utilizando el operador `*` y sin utilizarlo.

Utilizando el operador `*` el puntero se convierte en una variable normal y por lo tanto admite todas las operaciones de estas variables. Por ejemplo `i=20 * *pi`, `*pi=a+b`, `*pi+=7`, etc. En operaciones normales de suma, resta, multiplicación, división, comparaciones, etc. no hace falta ningún cuidado especial, sólo que el asterisco vaya unido al nombre del puntero para que no se confunda con el operador de multiplicación. Las

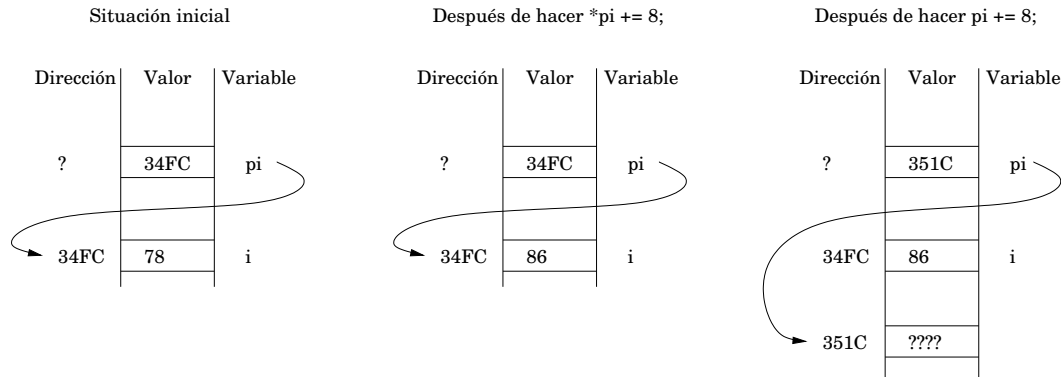


Figura 9.3: Incremento de variables e incremento de punteros

operaciones de incremento y decremento puede dar lugar a confusión, por la mezcla de operadores, y se aconseja escribirlas siempre con paréntesis adicionales: `++(*p)`, `(*p)++`, `--(*p)` y `(*p)--`.

Sin utilizar el operador `*` se manipula la dirección del puntero y por lo tanto cambia el lugar al que apunta. La operación más normal es incrementar la dirección en uno, con ello se pasa a apuntar al dato que ocupa la siguiente posición de memoria. La aplicación fundamental de esta operación es recorrer posiciones de memoria consecutivas. Además de valer para fisgonear la memoria del ordenador, ésto se utiliza para recorrer vectores (cuyos datos siempre se almacenan en posiciones de memoria consecutivas). Pero antes de pasar al apartado de vectores veamos cuanto se incrementa realmente un puntero.

Generalmente cada posición de memoria del ordenador puede almacenar un byte, por lo tanto aumentando la dirección de un puntero en 1 pasaríamos a apuntar al siguiente byte de la memoria. Esto es algo bastante claro aunque difícil de manejar, porque como se ha visto repetidamente a lo largo del libro, cada tipo de variable ocupa un tamaño de memoria diferente. Por ejemplo los **char** sí ocupan 1 byte, pero los **int** ocupan 2 ó 4 bytes, los **double** ocupan 8, etc. Además, estos tamaños dependen del ordenador y del compilador que se utilice. Para solucionar este problema, los punteros de cualquier tipo siempre apuntan al primer byte de la codificación de cada dato. Cuando se accede al dato mediante el operador `*`, el compilador se encarga de acceder a los bytes necesarios a partir de la dirección almacenada en el puntero. En el caso de punteros a **char** sólo se accede al byte almacenado en la dirección de memoria del puntero, mientras que el caso de punteros a **double** se accede a la posición de memoria indicada en el puntero y a los 7 bytes siguientes. Todo este mecanismo ocurre de manera automática y el programador sólo debe preocuparse de declarar punteros a **char** cuando quiere trabajar con valores **char** o punteros a **double** cuando quiere trabajar con valores **double**.

También para facilitar las cosas, la operación de sumar 1 a un puntero hace que su dirección se incremente la cantidad necesaria para pasar a apuntar al siguiente dato del mismo tipo. Es decir sólo en el caso de variables que ocupan 1 byte en memoria (variables **char**) la operación de incremento aumenta en 1 la dirección de memoria, en los demás casos la aumenta más. Lo que hay que recordar es que siempre se incrementa un dato completo y no hace falta recordar cuánto ocupa cada dato en la

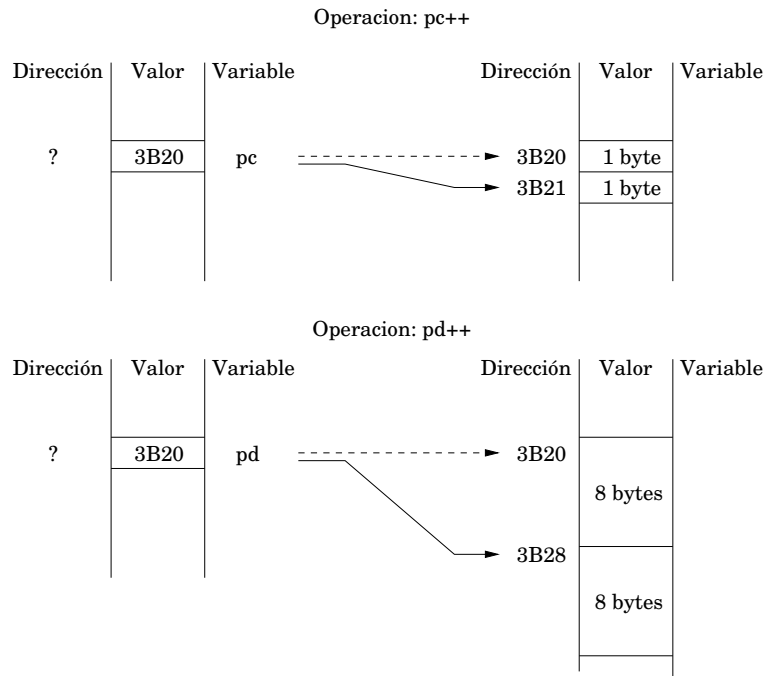


Figura 9.4: Incremento de punteros

memoria. Por ejemplo, si pc es un puntero a **char** que vale 0x3B20, la operación pc++ hará que pase a valer 0x3B21. Por otro lado si pd es un puntero a **double** que vale 0x3B20, la operación pd++ hará que pase a valer 0x3B28 (ver la figura 9.4).

9.4. Punteros y funciones

Una de las aplicaciones principales de los punteros en C es la de permitir que una función pueda modificar sus argumentos, para de esta forma poder “devolver” más de un valor.

Hasta ahora hemos visto que los argumentos de una función no se modifican, ya que la función trabaja sobre copias de los mismos. Esto es útil y muy seguro la mayoría de las veces, pero en otras ocasiones nos puede interesar modificar realmente el valor de dichos argumentos. Por ejemplo, la siguiente función para cambiar el valor de dos variables no hace nada:

```
/* Función: Cambio
   Descripción: Intenta cambiar el valor de dos variables.
   Comentario: No lo consigue, hacen falta punteros.
   Revisión: 1.0
   Autor: Novato.
*/
void cambio(int a, int b)
{
    int tmp;
```

```

    tmp=a;
    a=b;
    b=tmp;
}

```

Esta función no hace nada porque cambia los valores de *a* y *b*, pero éstos son copias de las variables que se utilizan al llamar a la función. Si se realiza la llamada: *cambio(x, y)*, entonces *a* es una copia de *x* y *b* es una copia de *y*. Aunque la función cambia los valores de *a* y *b*, al terminar y volver al programa principal, los valores de *x* e *y* no se ven alterados.

La manera de conseguir que la función *cambio* sea útil es trabajar con punteros. La versión 1.1 de esta función utiliza como argumentos dos punteros en lugar de dos enteros; el nuevo prototipo es:

```
void cambio(int *pa, int *pb);
```

Al recibir las direcciones de las dos variables, sí tenemos acceso a los valores originales² y por lo tanto se puede hacer el cambio. Ahora se trata de cambiar los valores contenidos en las direcciones apuntadas por los punteros *pa* y *pb*.

```

/* Función: Cambio
   Descripción: Intenta cambiar el valor de dos variables.
   Comentario: OK.
   Revisión: 1.1
   Autor: El que sabe de punteros.
*/
void cambio(int *pa, int *pb)
{
    int tmp;

    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

```

La única duda sería ¿Cómo llamamos a la función para darle las direcciones en lugar de los valores? La respuesta ya se ha visto porque el operador *&* nos facilita la dirección de cualquier variable. En este caso la llamada para cambiar los valores de las variables *x* e *y* es la siguiente:

```
cambio(&x, &y);
```

De hecho, llevamos utilizando llamadas de este estilo desde que se mencionó por primera vez la función *scanf*. ¿Por qué en la función *scanf* se escriben las variables con un *&* mientras que en *printf* se escriben sin nada? La respuesta es sencilla después de haber entendido los punteros y las funciones. La función *printf* se encarga de escribir el valor de nuestras variables y le da igual tener acceso directo al valor o trabajar sobre una copia; para mayor facilidad de programación (y para mayor seguridad) se opta por pasarle una copia. Sin embargo la función *scanf* lee un valor del teclado y lo guarda dentro de una de las variables de la función que la llama. Por lo tanto necesita acceso directo a la dirección de memoria donde se almacena el valor de la variable.

²Usando el operador de indirección *** sobre el puntero.



En la página Web tiene una animación que muestra el uso de punteros y funciones.

Y la última pregunta es ¿Por qué es tan importante especificar correctamente el formato de las variables en `scanf`? Porque si el formato dice `%f` la función `scanf` escribe en total 4 bytes (`sizeof(float)`) a partir de la dirección de memoria, pero si el formato dice `%lf` entonces `scanf` escribe 8 bytes. Cuando la variable no coincide con el tipo especificado pueden ocurrir dos cosas: que se escribe fuera de las posiciones de memoria reservadas para la variable (probablemente machacando la variable de al lado) o que no se escribe suficiente (dejando parte de la variable sin inicializar y por tanto con un valor erróneo).

9.4.1. Retorno de más de un valor por parte de una función

Como se ha visto anteriormente una función sólo puede devolver un valor mediante la sentencia **return**. Sin embargo en muchas ocasiones es deseable que una función devuelva más de un valor. En estos casos la técnica usada consiste en pasarle a la función la dirección de las variables donde queremos que nos devuelva sus resultados; de modo que la función pueda modificar directamente el valor de dichas variables. Por ejemplo, si queremos realizar una función que devuelva la suma y el producto de dos valores, dicha función se escribiría:

```
void SumaProd(double *psuma, double *pprod, double dato1,
              double dato2)
{
    *psuma = dato1 + dato2;
    *pprod = dato1 * dato2;
}
```

y la manera de llamarla para que devuelva la suma y el producto de 2 y 3 en las variables `sum` y `prod` sería:

```
double sum;
double prod;

SumaProd(&sum, &prod, 2.0, 3.0);
```

WEB

En la sección de códigos típicos hay ejemplos de prototipos de funciones que manejan punteros.

Por convenio, en estos casos se suelen colocar en la lista de argumentos las variables en las que devuelve la función sus resultados en primer lugar.

Si la función sólo devolviese un valor, típicamente se habría declarado de tipo **double** y se devolvería el resultado mediante un **return**. Sin embargo, al querer realizar dos cálculos dentro de la función nos vemos obligados a utilizar punteros y declaramos la función como **void** (puestos a usar punteros devolvemos los dos resultados por medio de punteros). Nótese que para llamar a una función que trabaje con punteros, en el `main` no se declaran punteros sino variables normales y luego se pasan sus direcciones (con el operador `&`) a la función.

9.5. Punteros y vectores

Una de las aplicaciones más frecuentes de los punteros es el manejo de vectores y cadenas de caracteres. Como se recordará todos los elementos de un vector se almacenan en posiciones de memoria consecutivas y por lo tanto basta conocer la posición de memoria del primer elemento para poder recorrer todo el vector con un puntero. El siguiente ejemplo inicializa un vector de **double** por el método “normal” y luego escribe su contenido con la ayuda de un puntero.

```

/*
Programa de punteros y vectores.
Descripción: Este programa inicializa un vector y
              luego escribe sus valores.
Versión: 1.0
*/
#include <stdio.h>
#define N 10

int main(void)
{
    double a[N]; /* vector */
    int i;        /* contador */
    double *pd;   /* puntero a double */

    /*** Inicialización ***/
    for(i=0; i<N; i++) {
        a[i] = 7.8*i;
    }

    /*** Imprimir valores ***/
    pd = &a[0]; /* Apunto al primer elemento del vector */
    for(i=0; i<N; i++) {
        printf("%f\n", *pd); /* *pd es de tipo double */
        pd++; /* pd pasa a apuntar al siguiente elemento */
    }
    return 0;
}

```

Hay que tener en cuenta que cuando termina el programa el puntero `pd` queda apuntado a un lugar no válido, ya que queda fuera del vector `a`.

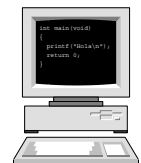
Supongamos ahora que queremos copiar el vector `a` en el vector `b` de manera que sea su inverso; es decir, si `a` vale [1, 2, 3, 4] queremos que `b` valga [4, 3, 2, 1]. Generar `b` a partir de `a` es muy sencillo utilizando dos punteros `pa` y `pb`, uno creciente y otro decreciente:

```

pa=&a[0]; /* pa apunta al primer elemento del vector a */
pb=&b[N-1]; /* pb apunta al último elemento del vector b */
for(i=0; i<N; i++) {
    *pb=*pa; /* copio un elemento */
    pa++; /* avanzo un elemento */
    pb--; /* retrocedo un elemento */
}

```

Con mucha frecuencia se utilizan varios punteros sobre el mismo vector para copiar datos o hacer manipulaciones sin tener que andar con la complicación de manejar varios índices. Los ejemplos más típicos son las transformaciones de cadenas de caracteres.



Realice el ejercicio 2.

9.5.1. Equivalencia de punteros y vectores

Cuando se define un vector, tal como se ha visto en el ejemplo anterior:

```
double a[N];
```

lo que ocurre es que se reserva un espacio en la memoria para almacenar N elementos de tipo **double** y se crea un puntero constante,³ llamado a, que apunta al principio del bloque de memoria reservada.

Por tanto para hacer que el puntero pd apunte al principio de la matriz a se puede hacer `pd=&a[0]` o simplemente `pd=a`.

Uso de corchetes [] con punteros y vectores

Cuando se accede a un elemento de un vector mediante:

```
a[3]=2.7;
```

el programa toma el puntero constante a, le suma el valor que hay escrito entre los corchetes (3) y escribe en dicha dirección el valor 2.7. Es decir, la sentencia anterior es equivalente a escribir:

```
*(a+3)=2.7;
```

Además los corchetes se pueden aplicar sobre cualquier puntero, no sólo sobre los punteros constantes, por lo que los dos trozos de código que siguen son equivalentes:

```
pd=a;
a[3]=2.7;          pd[3]=2.7;
```

A modo de resumen, si tenemos un vector `vec` de 100 elementos y queremos asignar el valor 45 al quinto elemento, pondremos:

```
vec[4]=45;
```

En caso de tener un puntero `punt` que apunta al primer elemento de vector, podemos realizar la misma asignación de tres maneras:

- Moviendo el puntero y luego volviendo atrás

```
punt+=4;
*punt=45;
punt-=4;
```

- Calculando la dirección directamente

```
*(punt+4)=45;
```

- Utilizando corchetes, que es lo más cómodo y lo que se utiliza normalmente.

```
punt[4]=45;
```

³El programa no puede cambiar la dirección almacenada en un puntero constante. Por ejemplo `a=&d` es ilegal

9.6. Asignación dinámica de memoria

Existen infinitud de aplicaciones en las cuales no se conoce la cantidad de memoria necesaria para almacenar los datos hasta que no se ejecuta el programa. Si por ejemplo se desea escribir un programa que calcule el producto escalar de dos vectores, la dimensión de éstos no se conoce hasta que no se le pregunta al usuario al comienzo del programa. En este tipo de situaciones las posibles soluciones son dos:

- Crear vectores de un tamaño fijo, pero lo suficientemente grande.
- Crear el vector dinámicamente al ejecutarse el programa.

La primera solución presenta dos inconvenientes: el primero es que si el usuario necesita calcular el producto escalar de dos vectores de dimensión mayor a la dimensión máxima que se eligió al escribir el programa, tendrá que llamarnos para que incrementemos dicha dimensión máxima (lo cual puede estar bien para cobrarle soporte técnico, pero dará mucho que hablar sobre nuestra habilidad como programadores). El segundo inconveniente es que normalmente se está desperdiciando una gran cantidad de memoria en definir un vector demasiado grande del cual sólo se va a utilizar una pequeña parte en la mayoría de los casos. En esta situación, dado que todos los sistemas operativos actuales son multitarea, si un programa usa toda la memoria del ordenador no se podrán arrancar otros programas.

Estos dos inconvenientes se solucionan con el uso de memoria dinámica: mediante esta técnica, una vez que se está ejecutando el programa y se conoce la cantidad de memoria que se necesita, se realiza una **llamada al sistema operativo** para solicitarle un bloque de memoria libre del tamaño adecuado. Si queda memoria, el sistema operativo nos devolverá un **puntero** que apunta al comienzo de dicho bloque. Este puntero nos permite acceder a la memoria a nuestro antojo, siempre y cuando no nos salgamos de los límites del bloque.⁴ Una vez que se termina de usar la memoria, ésta debe **liberarse** para que los demás programas puedan usarla.

Esta técnica permite por tanto un manejo más racional de la memoria, aparte de permitirnos crear estructuras de datos más complejas como listas enlazadas y árboles (que quedan fuera del alcance de este libro).

9.6.1. Las funciones *calloc* y *malloc*

Estas dos funciones permiten al programa solicitar al sistema operativo un bloque de memoria de un tamaño dado. Ambas funciones devuelven un puntero al principio del bloque solicitado o NULL si no hay suficiente memoria. Es muy importante por tanto verificar **siempre** que se solicite memoria al sistema operativo que éste nos devuelve un puntero válido y no un NULL para indicarnos que no tiene tanta memoria disponible.

Los prototipos de ambas funciones son:

```
void *calloc(size_t numero_elementos, size_t tamaño_elemento);  
void *malloc(size_t tamaño_bloque);
```

y sus declaraciones se encuentran en el archivo cabecera `stdlib.h` que habrá de ser incluido al principio del código para que el compilador pueda comprobar que las lla-

⁴Si intentamos leer o escribir fuera del bloque de memoria que se nos ha asignado, los resultados pueden ser desastrosos en función de la zona de memoria a la que accedamos erróneamente. Por tanto hay que ser muy cuidadosos cuando se usan estas técnicas.

madas se realizan correctamente. Por lo demás, como ambas funciones pertenecen a la librería estándar, no hace falta añadir ninguna librería adicional en el enlazado.

La función `calloc` reserva un bloque de memoria para un *numero_elementos* de *tamaño_elemento*, inicializa con ceros el bloque de memoria y devuelve un puntero genérico (**void ***) que apunta al principio del bloque o `NULL` en caso de que no exista suficiente memoria libre.

La función `malloc` reserva un bloque de memoria de *tamaño_bloque* (medido en bytes) y devuelve un puntero al principio del bloque o `NULL` en caso de que no exista suficiente memoria libre.

Por ejemplo para crear un vector de 100 enteros se podría realizar lo siguiente:

```
int *pent; /* puntero al que asignamos memoria */
...
pent = (int *) calloc(100, sizeof(int));
if(pent == NULL){
    printf("Error: No hay suficiente memoria\n");
}else{
    /* Hay memoria: podemos usar el vector */
    pent[0] = 27; /* Recuerde la equivalencia entre punteros
                  y vectores */
    ...
    free(pent); /* Se explica más adelante */
}
...
```

La petición de memoria del programa anterior podría haberse realizado también de la siguiente manera:⁵

```
pent = (int *) malloc(100*sizeof(int));
```

En ambas llamadas se reserva un bloque de 100 enteros y se devuelve un puntero al principio de dicho bloque. Nótese que en el prototipo de estas dos funciones se especifica que se devuelve un puntero de tipo **void**. Este tipo es un puntero genérico que ha de convertirse mediante un molde (*cast*) al tipo de puntero con el que vamos a manejar el bloque. En el ejemplo, como se deseaba crear un vector de 100 enteros, el puntero devuelto se ha convertido en un puntero a entero mediante el molde (**int ***).

Por último, destacar que en las llamadas a las funciones `calloc` y `malloc` se ha usado `sizeof(int)` para calcular la cantidad de memoria que se necesita. Esto es muy importante de cara a la portabilidad del programa. Si aprovechando que conocemos el tamaño de un **int** en un PC con GCC hubiésemos puesto un 4 en lugar de `sizeof(int)`, el programa no funcionaría bien si lo compilamos en Borland C para MS-DOS, el cual usa enteros de 2 bytes.

9.6.2. La función *free*

Una vez que se ha terminado de usar la memoria es necesario liberarla para que quede disponible para los demás programas. El no hacerlo hará que cada vez que se ejecute nuestra función “desaparezca” una parte de la memoria. La función `free` libera la memoria **previamente asignada** mediante `calloc` o `malloc`. Se ha recalcado lo de

⁵Aunque en este caso la memoria no se inicializaría con ceros.

previamente asignada porque liberar memoria que no ha sido asignada⁶ es un grave error que puede dejar “colgados” a algunos sistemas operativos.

El prototipo de esta función es:

```
void free(void *puntero_al_bloque);
```

que también está en el archivo cabecera `stdlib.h`. Esta función, al igual que sus compañeras de faenas: `calloc` y `malloc`, se encuentra en la librería estándar.

La función simplemente libera el bloque de memoria a cuyo principio apunta el *puntero_al_bloque*.

Tal y como aparece el ejemplo anterior, para liberar la memoria previamente asignada se realiza la llamada:

```
free(pent);
```

Hay que destacar que el *puntero_al_bloque* ha de apuntar exactamente al principio del bloque, es decir, ha de ser el puntero devuelto por la llamada a la función de petición de memoria (`calloc` o `malloc`). Por ejemplo, el siguiente trozo de código hará que el programa aborte al hacer la llamada a `free` o, si el sistema operativo no es muy espabilado, lo dejará incluso colgado:

```
int *pent;

pent = calloc(100, sizeof(int));
if(pent == NULL){
    printf("Error: Se ha gastado la memoria. Compre más.\n");
}else{
    ...
    pent++; /* peligro!! */
    ...
    free(pent); /* Error catastrófico */
}
```

También es un grave error el seguir usando la memoria una vez liberada, pues aunque tenemos un puntero que apunta al principio del bloque, este bloque ya no pertenece a nuestro programa y por tanto puede estar siendo usado por otro programa, por lo que cualquier lectura nos puede devolver un valor distinto del escrito y cualquier escritura puede modificar datos de otro programa, tal como se muestra en el siguiente ejemplo:

```
int *pent; /* puntero al principio de la vector */
int a;     /* Variable auxiliar */

pent = (int *) calloc(100, sizeof(int));
if(pent == NULL){
    printf("Error: No hay suficiente memoria\n");
}else{
    pent[0] = 27; /* recuerde la equivalencia entre punteros
                  y vectores */

    free(pent);
```



En la sección de códigos típicos hay un ejemplo con la estructura general para asignar memoria.

⁶O liberarla varias veces, que es el error más habitual.

```

    a = pent[0]; /* ERROR! El bloque al que apunta pent ya no
                  pertenece a nuestro programa. Por tanto puede
                  que en a no se escriba 27 */
    pent[0] = 40; /* ERROR! Estamos modificando memoria que ya no
                  es nuestra y los resultados pueden ser
                  catastróficos */
}

```

9.6.3. Ejemplo

Para fijar ideas vamos a ver a continuación un programa en el que se hace uso de asignación dinámica de memoria.

Se ha de escribir un programa que pida un vector al usuario y una vez que éste lo haya introducido, ha de generar otro vector que contenga sólo los elementos del primer vector que sean números pares e imprimirlos en la pantalla. Por supuesto la dimensión de ambos vectores es desconocida antes de ejecutar el programa.

Como no se conoce la dimensión de ninguno de los vectores y sabemos ya un montón acerca de asignación dinámica de memoria decidimos, en lugar de crear dos vectores enormes que valgan para cualquier caso, crear los vectores del tamaño justo cuando se ejecute el programa. Un pseudocódigo del programa se muestra a continuación:

```

Pedir dimensión del vector al usuario
Pedir memoria para el vector
Pedir al usuario cada uno de los elementos del vector
Contar el número de elementos pares del vector
Pedir memoria para el vector de números pares.
Copiar los números pares del primer vector al segundo
Imprimir el segundo vector
Liberar memoria

```

El programa completo se muestra a continuación:

```

/* Programa: BuscaPar
 *
 * Descripción: Pide al usuario un vector de enteros y genera otro
 *               que contiene sólo los elementos pares de dicho
 *               vector, imprimiéndolo antes de terminar.
 *
 * Revisión 0.0: 19/05/1998
 *
 * Autor: El programador dinámico.
 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *pvec_usu; /* Puntero al vector introducido por el usuario

```

```

        (creado dinámicamente) */
int *pvec_par; /* Puntero al vector de elementos pares (creado
                dinámicamente) */
int dim_usu;   /* Dimensión del vector del usuario */
int dim_par;   /* Dimensión del vector de elementos pares */
int n;         /* Índice para los for */
int m;         /* Índice para recorrer la matriz de elementos
                pares */

printf("Introduzca la dimensión del vector: ");
scanf("%d", &dim_usu);

/* Asignamos memoria para el vector del usuario. Contiene
   dim_usu enteros */
pvec_usu = (int *) calloc(dim_usu, sizeof(int));

if(pvec_usu == NULL){ /* Estamos sin memoria */
    printf("Error: no hay memoria para un vector de %d "
           "elementos\n", dim_usu);
}else{

    /* Pedimos los elementos del vector */
    for(n=0; n<dim_usu; n++){
        printf("Elemento %d = ", n);
        scanf("%d", &(pvec_usu[n]) );
    }

    /* Contamos los pares en la variable dim_par */
    dim_par = 0; /* De momento no hay ningún elemento par */
    for(n=0; n<dim_usu; n++){
        if( (pvec_usu[n]%2) == 0 ){ /* es par */
            dim_par ++;
        }
    }

    /* Se asigna memoria para los números pares */
    pvec_par = (int *) calloc(dim_par, sizeof(int));
    if(pvec_par == NULL){ /* Estamos sin memoria */
        printf("Error: no hay memoria para un vector de %d "
               "elementos\n", dim_par);
    }else{

        /* Se copian los elementos pares */
        m = 0; /* Índice para el vector de elementos pares
                (inicialmente apunta al primer elemento) */
        for(n=0; n<dim_usu; n++){
            if( (pvec_usu[n]%2) == 0 ){ /* es par */
                pvec_par[m] = pvec_usu[n]; /* copio el elemento */
                m++; /* e incremento el índice del vector par */
            }
        }
    }
}

```



```

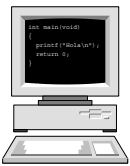
    }

    /* Se imprime el vector de elementos pares */
    printf("\n-----\n"); /* Separación entre
                               los dos vectores */

    for(n=0; n<dim_par; n++){
        printf("Elemento par %d = %d\n", n, pvec_par[n]);
    }

    free(pvec_par);
}
free(pvec_usu);
}
return 0;
}

```



Realice el ejercicio 5.

Un ejemplo de la ejecución del programa se muestra a continuación:

```

Introduzca la dimensión del vector: 3
Elemento 0 = 1
Elemento 1 = 2
Elemento 2 = 4

```

```

-----
Elemento par 0 = 2
Elemento par 1 = 4

```

9.7. Recomendaciones y advertencias

- Las dos aplicaciones principales de los punteros son las funciones y la asignación dinámica de memoria pero tenga en cuenta que:
 - Un programa que utiliza funciones que reciben punteros como argumentos no declara punteros en el main. Sólo declara variables normales y le pasa sus direcciones a la función utilizando el operador &. La función realizará sus operaciones y guardará sus resultados en las posiciones de memoria que le han pasado aplicando el operador * a los punteros que le han pasado como argumento.
 - Un programa sencillo que utiliza memoria dinámica sólo declara punteros en el main y realiza las llamadas a calloc y free dentro del main. La utilización de memoria dinámica se realiza en la práctica mediante los corchetes [] como si fuera memoria estática (igual que en el capítulo 8).
- Como norma general, los nombres de las variables de tipo puntero empiezan con la letra 'p' para diferenciarlas de las variables normales.
- Si se asigna memoria dinámica temporal dentro de una función, dicha memoria ha de liberarse antes de retornar de la función.
- Aunque al principio de este capítulo se han declarado punteros en el main para apuntarlos a variables normales mediante instrucciones del tipo pa = &i, esto se ha hecho para explicar el funcionamiento de los punteros. Los programas de

la vida real sólo declaran punteros en el `main` cuando van a realizar asignación dinámica de memoria. En todo caso se puede declarar un puntero auxiliar dentro de una función para recorrer un vector o una lista.

9.8. Resumen

En este capítulo se ha explicado el concepto de puntero, cómo se declara y cómo se utiliza. Inicialmente resulta un poco confuso tener que pensar en las direcciones de memoria en las que se almacenan los valores que manejan nuestros programas, sobre todo porque se pierde parcialmente el nivel de abstracción que proporcionaban las variables desde el capítulo 3. Los punteros parecen así una vuelta atrás en la programación.

Sin embargo, una vez estudiado el concepto, las dos aplicaciones que se han presentado, **funciones** y **memoria dinámica**, no presentan dificultad alguna. También es posible extrapolar los conceptos a estructuras de datos más complejas, como las matrices dinámicas o las listas enlazadas.

La utilidad principal de los punteros como argumentos de **funciones** es la de poder modificar los valores de las variables del `main`⁷ desde dentro de la función, o bien para conseguir que una función pueda “devolver” más de un resultado.

La **asignación dinámica de memoria** permite escribir programas generalistas que sólo estarán limitados por el *hardware* en el que se ejecuten. Una vez asignada la memoria a un puntero, ésta se utiliza como si fuera un vector estático.



Vea la animación sobre la asignación de memoria para matrices.

9.9. Ejercicios

1. ¿Sabría explicar por qué el programa de la sección 9.2.1 (página 117) imprime un valor erróneo?
2. Escribir un programa que obtenga la simetría de una cadena de caracteres. Por ejemplo, si la cadena vale "avión" el resultado debe ser "aviónnóiva". Escriba una versión del programa manipulando la cadena como un vector (usando índices) y otra usando punteros.
3. Escribir una **función** que devuelva un uno si la cadena que se le pasa como argumento es un palíndromo o cero si no lo es. El prototipo de la función será:

```
int EsPalindromo(char cadena[]);
```

Por ejemplo, si `cadena` contiene "holaaloh" el resultado de la función ha de ser uno y si contiene "Voy a aprobar el examen" el resultado será un cero (el de la función, no el del examen... ¡espero!).

4. Modificar el programa de la sección 9.6.3 para que la petición de datos se realice en una función. El prototipo de dicha función ha de ser:

```
void PideVec(int *pvec, int dimension);
```

5. Puesto que los vectores siempre se recorren desde principio a fin y de elemento en elemento puede ser más fácil usar punteros para recorrer los vectores. Modificar el programa mostrado en la sección 9.6.3 (pág. 128) para que la búsqueda de

⁷O en general de la función que la llama.

elementos pares, la creación del vector de números pares y su impresión, se realicen mediante punteros en lugar de mediante índices. Es necesario definir dos punteros auxiliares e inicializarlos con el mismo valor que los punteros base).

6. Escribir el **conjunto de instrucciones** necesario para asignar al puntero `pa` un espacio de memoria con capacidad para 100 valores de tipo `int`. A continuación habrá que inicializar a 1 los 100 elementos.

Escribir también las instrucciones necesarias para liberar la memoria asignada una vez que se ha terminado de usar.

7. Dentro de un programa es necesario crear un vector dinámicamente para almacenar un conjunto de valores de tipo `double`. La longitud del vector se preguntará al usuario. A continuación se asignará la memoria necesaria al puntero `pd` y se inicializarán a 1.0 todos los elementos.

Escribir las instrucciones necesarias para realizar las tareas descritas y también las instrucciones necesarias para liberar la memoria asignada.

8. Escribir un **programa** que calcule el producto escalar de dos vectores de tipo `double` de cualquier dimensión. Para ello el programa pedirá al usuario en primer lugar la dimensión de los vectores y a continuación les asignará memoria dinámicamente. Una vez conseguida la memoria se pedirán al usuario los valores de cada elemento de ambos vectores y se llamará a la función `ProdEscalar` escrita en la sección 8.6.1 (pág. 107) para calcular el producto escalar de ambos vectores. Finalmente el programa principal imprimirá por pantalla el resultado y liberará la memoria asignada a los vectores.

9. Escribir un programa completo para trabajar con cadenas de caracteres de tamaño fijo y cadenas dinámicas. El programa debe declarar dos vectores tipo `char` de tamaño fijo (especificado mediante una `CONSTANTE`) que se leerán con la función `fgets()`. A continuación se calculará la suma de las longitudes de las dos cadenas y se creará dinámicamente otra cadena de caracteres con espacio suficiente para almacenar las dos cadenas. Por último se copiarán, sobre la cadena recién creada, las dos cadenas originales, una a continuación de otra. El programa finaliza mostrando el resultado por pantalla.

10. Escribir una **función** para calcular la suma y el producto de dos variables `x` e `y`. El prototipo de la función debe ser:

```
void fun(double *suma, double *producto, double x, double y);
```

11. Escribir una función para ordenar tres números enteros de menor a mayor; es decir, si se tienen tres variables que valen respectivamente 3, 5 y 2, después de llamar a la función las tres variables contendrán respectivamente 2, 3 y 5.

Una vez escritas la función y su prototipo, escriba también un programa principal que inicialice tres variables, llame a la función para ordenarlas y luego las imprima por pantalla.

12. Escribir una **función** que calcule el máximo y el mínimo de un vector. El prototipo de la función será:

```
void MinMax(double vec[], int dimension, double *pmin,  
            double *pmax);
```

Como se puede apreciar, la función recibe la dirección del vector y su dimensión, así como dos punteros a sendas variables de tipo **double** en las que la función guardará sus resultados.

Suponiendo que en el programa principal se han realizado las siguientes declaraciones:

```
double vector[20]; /* Vector de datos */  
double min,max;    /* Mínimo y máximo del vector */
```

Escribir como se llamaría a la función MinMax() para calcular el máximo y el mínimo de vector, almacenando el mínimo en la variable min y el máximo en max.

13. Escribir una **función** para leer el nombre de una persona, su edad y su DNI. La cadena que recibe la función para almacenar el nombre tiene una dimensión de 100 caracteres y deberá de protegerse la función adecuadamente para evitar fallos si se introducen más de 100 caracteres. Además habrá que comprobar que la edad y el DNI cumplan las siguientes condiciones:

```
0 < edad < 200  
500000 < dni < 100000000
```

Cuando no se cumpla alguna de estas dos condiciones, habrá que dar un mensaje de error y volver a pedir la variable.

CAPÍTULO 10

Estructuras de datos

10.1. Introducción

En muchas situaciones un objeto no puede describirse por un solo dato, sino por un conjunto de ellos. Por ejemplo el historial médico de un paciente consta de varios campos, siendo cada uno de ellos de un tipo distinto. Un ejemplo de historial médico, junto con los tipos de datos utilizados podría ser el mostrado en el siguiente cuadro:

Dato	Tipo de dato	Tipo en C
Nombre	Cadena de caracteres	char[100]
Edad	Número entero	unsigned short
Peso	Número en punto flotante	float
NºSeguridad Social	Número entero	unsigned long
Historia	Cadena de caracteres	char[1000]

Si se desea realizar un programa para gestionar una base de datos con los historiales de los pacientes, sería mucho más cómodo y más claro poder agrupar todos los datos de un paciente bajo una misma variable que declarar cinco variables diferentes. En C estos tipos de variables definidos como una agrupación de datos, que pueden ser de distintos tipos, se denominan estructuras.

Hasta ahora habíamos visto que C disponía de unos tipos básicos para manejar números y caracteres y que a partir de estos tipos básicos se podían crear vectores y matrices, que eran agrupaciones de datos pero todos de un mismo tipo. Las estructuras de datos son un paso más allá, al permitir agrupar en una misma variable datos de tipos distintos, tal como se va a mostrar en este capítulo.

10.2. Declaración y definición de estructuras

La sintaxis de la declaración de una estructura es:

```
struct nombre_de_estructura{  
    tipo_1 nombre_miembro_1;  
    tipo_2 nombre_miembro_2;  
    ...  
    tipo_n nombre_miembro_n;  
};
```

Esta declaración no reserva espacio en la memoria para la estructura; simplemente crea una **plantilla** con el formato de la estructura. La sintaxis para definir una variable del tipo estructura previamente declarado es:

```
struct nombre_de_estructura var;
```

Esta sentencia reservará un espacio en la memoria para albergar una estructura de tipo *nombre_de_estructura*, a la cual se podrá acceder mediante la variable *var*.

Por ejemplo la declaración de una estructura para almacenar el historial médico de un paciente se escribiría:

```
struct paciente{
    char nombre[100];
    unsigned short edad;
    float peso;
    unsigned long n_seg_social;
    char historia[1000];
};
```

Y para crear una variable del tipo estructura paciente se escribe:

```
struct paciente pepito;
```

Nótese que una vez declarada una estructura, dicha estructura se puede considerar como un nuevo tipo de dato con el nombre **struct** *nombre_de_estructura*, y que la declaración de variables de ese nuevo tipo de dato es idéntica a la del resto de variables; compárense si no las dos definiciones siguientes:

```
unsigned long numero;
struct paciente juanito;
```

10.3. La sentencia typedef

typedef permite dar un nombre arbitrario a un tipo de datos de C, ya sea un tipo básico (**int**, **char**, etc.) o derivado (**unsigned long int**, **struct** paciente, etc.).

Por ejemplo si se sabe que en un PC un **short** ocupa dos bytes, se puede escribir al comienzo del programa la definición de tipos:

```
typedef unsigned short WORD;
```

Y a partir de entonces, WORD se convierte en un sinónimo de **unsigned short**, por lo que si se desea definir una variable de dos bytes se puede hacer:

```
WORD edad;
```

En lugar de:

```
unsigned short edad;
```

Aparte del ahorro al teclear la definición de enteros de dos bytes, esto tiene la ventaja de que si se desea portar el programa a una máquina en la que los enteros de dos bytes son el tipo **int** sólo habría que cambiar la declaración de WORD al principio del programa, dejando intacto el resto de definiciones.

Los **typedef** deben definirse antes de declarar cualquier variable del nuevo tipo definido. Por tanto la posición ideal para poner todos los **typedef** es fuera del main, justo delante de los prototipos.

También se puede usar **typedef** para simplificar la definición de variables de tipo estructura. Para ello se puede unir la declaración de la estructura con una definición de tipos de la forma:

WEB

Véase la estructura general de un programa en C.

```
typedef struct paciente{
    char nombre[100];
    unsigned short edad;
    float peso;
    unsigned long n_seg_social;
    char historia[1000];
}PACIENTE;
```

La definición de variables de tipo **struct** paciente se podrá realizar ahora como:

```
PACIENTE pepito;
```

Lo cual mejora bastante la legibilidad del programa.

Después de definir PACIENTE mediante **typedef**, éste se convierte en un nuevo tipo de datos, con las mismas características que **int** o **double**. Incluso es posible (y bastante habitual) declarar una estructura en la cual uno o varios de sus campos son también de tipo estructura. Por ejemplo:

```
typedef struct fecha{
    int dia;
    int mes;
    int año;
}FECHA;

typedef struct paciente{
    char nombre[100];
    FECHA fecha_nacimiento;
    float peso;
    unsigned long n_seg_social;
    char historia[1000];
}PACIENTE;
```

En resumen el uso de **typedef** tiene dos finalidades:

- Permite una mejor documentación del programa al dar nombres más significativos a los tipos, especialmente a los derivados.
- Mejora la portabilidad del programa, al poder encapsular tipos que dependan de la máquina en sentencias **typedef**.

Nota

Cuando se declara una estructura mediante **typedef** el nombre de la estructura (paciente en el ejemplo anterior) puede omitirse.

10.4. Acceso a los miembros de una estructura

Para acceder a los miembros de una estructura se usa el operador punto. La sintaxis de dicho operador es:

nombre_de_structura.nombre_del_campo

En el ejemplo del historial médico, para inicializar el historial pepito se escribiría:


```
strcpy(pegito.nombre, "José Pérez López");
pegito.edad = 27;
pegito.peso = 67.5;
pegito.n_seg_social = 27345190;
strcpy(pegito.historia, "Alérgico a las Sulfamidas");
```

y para imprimir los datos de un historial se podría escribir:

```
printf("Paciente %s:\n", pegito.nombre);
printf("  Edad: %d\n", pegito.edad);
printf("  Peso: %f\n", pegito.peso);
printf("  N° Seg Soc.: %ld\n", pegito.n_seg_social);
printf("  Historia: %s\n", pegito.historia);
```

Como puede apreciarse, aunque `pegito` es una variable de tipo **struct** `paciente`, o lo que es lo mismo, del tipo `PACIENTE` si se ha realizado el **typedef** de la sección 10.3; sus miembros obtenidos con el operador punto son variables normales, es decir, `pegito.nombre` es una cadena de caracteres y como tal se inicializa con la función `strcpy` y se imprime con el formato `%s`. De la misma manera `pegito.edad` es un **int** y `pegito.peso` es un **float**.

En definitiva el uso de los miembros de una estructura es idéntico al de las variables normales y utilizar el operador punto seguido del nombre del campo es equivalente a usar los corchetes y el índice del elemento en el caso de vectores. Incluso en el caso de poder almacenar la información en un vector, en muchos casos puede resultar más elegante utilizar estructuras. Por ejemplo, para almacenar las coordenadas de un punto en tres dimensiones, se puede optar por declarar el vector **double** `punto[3]` o bien una estructura con tres campos de tipo **double** llamados `x`, `y`, `z`. En el primer caso la inicialización de la coordenada `z` a 3,7 se realiza mediante `punto[2] = 3.7`; mientras que en el segundo caso se realiza mediante `punto.z = 3.7`; que resulta mucho más legible.

10.5. Ejemplo: Suma de números complejos

Se desea realizar un programa que pida al usuario dos números complejos y muestre el resultado de su suma. Para facilitar las cosas se va a declarar una estructura para almacenar un número complejo, de forma que el programa sea más fácil de realizar y de entender.

A continuación se muestra un listado del programa:

```
/* Programa: SumaComp
2  *
  * Descripción: Pide al usuario dos números complejos y calcula
4  *              su suma
  *
6  * Revisión 0.0: 10/05/1999
  *
8  * Autor: El programador estructurado.
  */
10 #include <stdio.h>
12 /* Declaración de una estructura para contener un número
```

```

14     complejo */
15     typedef struct complejo{
16         double real;
17         double imag;
18     }COMPLEJO;

20 int main(void)
21 {
22     COMPLEJO comp1, comp2; /* Los dos números complejos que se
                             pedirán al usuario desde el teclado */
24     COMPLEJO result;      /* Resultado de comp1+comp2 */

26     printf("Introduzca la parte real del primer complejo: ");
27     scanf("%lf", &comp1.real);
28     printf("Introduzca la parte imaginaria del primer complejo: ");
29     scanf("%lf", &comp1.imag);
30
31     printf("Introduzca la parte real del segundo complejo: ");
32     scanf("%lf", &comp2.real);
33     printf("Introduzca la parte imaginaria del segundo complejo: ");
34     scanf("%lf", &comp2.imag);

36     result.real = comp1.real + comp2.real;
37     result.imag = comp1.imag + comp2.imag;
38
39     printf("\nEl resultado de la suma es %lf + %lf j\n",
40           result.real, result.imag);
41     return 0;
42 }

```

Como puede apreciarse, en primer lugar se ha declarado la estructura `complejo` y se le ha asociado el tipo `COMPLEJO`¹ (líneas 13–16). Nótese además que dicha declaración se ha realizado antes de comenzar la función `main`. Esto último permite que el formato de la estructura se pueda usar en todas las funciones del programa. Si por el contrario la declaración de la estructura `complejo` se hubiese escrito dentro de la función `main`, dicha declaración sería privada a la función `main` y por tanto no se podría usar en otras funciones.

En segundo lugar es necesario destacar que el uso de los miembros de una estructura es idéntico al de una variable que tenga su tipo. Por ejemplo en la línea 25 se usa el operador `&` para obtener la dirección del miembro `real` de la estructura `comp1` para que `scanf` pueda guardar en dicha variable el número que introduzca el usuario por el teclado. Conviene destacar también que la precedencia del operador punto es mayor² que la del operador `&` y por tanto la dirección que se obtiene es la del miembro de la estructura.

¹El nombre de la estructura y el nombre del tipo no tienen por qué coincidir, aunque es conveniente para evitar confusiones.

²De hecho el operador punto, junto con el operador corchete (`[]`) para referenciar elementos de matrices y el operador flecha (`->`) que se estudiará mas adelante, son los operadores con la mayor precedencia en C.

10.6. Estructuras y funciones

Dos estructuras del mismo tipo se pueden asignar entre sí, es decir si se desea por ejemplo obtener una copia de una estructura del tipo paciente declarada en la sección 10.2 se puede hacer simplemente:

```
...
PACIENTE pepito;
PACIENTE copia_de_pepito;
...
/* aquí estaría la inicialización de pepito */
...
copia_de_pepito = pepito;
```

Esto es una diferencia importante con respecto a los vectores, que como recordará sólo pueden copiarse elemento a elemento mediante un bucle **for**; o con respecto a las cadenas de caracteres, que han de copiarse mediante un bucle o usando la función `strncpy`.

Como se recordará en las llamadas a funciones los argumentos de la función se **copian** en unas variables locales y el valor devuelto por la función también se copia a la variable a la que se asigna la función en el programa principal. Por tanto una función para inicializar una estructura del tipo `COMPLEJO` se puede escribir:

```
COMPLEJO InicComplejo(double real, double imag)
{
    COMPLEJO resultado;

    resultado.real = real;
    resultado.imag = imag;

    return resultado;
}
```

Para llamar a la función desde un programa se escribiría simplemente:

```
...
COMPLEJO comp1;
...
comp1 = InicComplejo(2.0, 3.7);
...
```

Como se puede apreciar la función recibe dos números reales y crea un número complejo llamado `resultado`. Dicha estructura, al terminar la función, es devuelta al programa principal donde se copia a la estructura `comp1`, que es también una estructura de tipo complejo.

El paso de parámetros es similar. Por ejemplo una función que devuelva la suma de dos complejos que se pasan como argumentos sería:

```
COMPLEJO SumaComp(COMPLEJO c1, COMPLEJO c2)
{
    COMPLEJO res;

    res.real = c1.real + c2.real;
    res.imag = c1.imag + c2.imag;
```

```

    return res;
}

```

En este ejemplo las variables `c1` y `c2` se pasan por valor, es decir, que la función recibe copias de los valores y las variables del `main` están “protegidas” (como cualquier otra variable). Sin embargo el paso de estructuras de un tamaño elevado a una función es un proceso costoso, tanto en memoria como en tiempo. Por ello el paso de estructuras a funciones por valor sólo debe realizarse cuando éstas sean de pequeño tamaño, tal como se acaba de hacer con la estructura `COMPLEJO`. Por ejemplo si se quiere pasar a una función una estructura del tipo `PACIENTE`, descrita en la sección 10.3, se copiarían 1110 bytes. En estos casos se opta siempre por pasar la estructura por referencia, es decir, mediante un puntero a la estructura en lugar de la estructura completa. De esta manera el proceso es el mismo que con los vectores y matrices, que siempre se pasan por referencia.

10.7. Punteros a estructuras

Un puntero a estructura se declara de la misma forma que un puntero a una variable ordinaria. Así por ejemplo en la declaración:

```
struct paciente *ppaciente;
```

Se ha creado un puntero a una estructura de tipo `paciente`, declarada en la sección 10.2. Si se usa **typedef** para definir un “alias” a la declaración de estructura, tal como se hizo en la sección 10.3, se podría crear el mismo puntero de la forma:

```
PACIENTE *ppaciente;
```

En un programa se haría una declaración de este tipo para poder asignar memoria dinámica al puntero `ppaciente` y en definitiva construir un **vector de estructuras** (ver 10.8.2). En el caso de las funciones se declaran punteros a estructuras en la lista de argumentos para manejarlos por referencia. Siguiendo con el ejemplo del apartado anterior, la función para sumar complejos si éstos se pasan por referencia se declarará como:

```
COMPLEJO SumaComp2(COMPLEJO *pc1, COMPLEJO *pc2);
```

Para obtener la dirección de una estructura se usa el operador `&` al igual que con el resto de variables. Por ejemplo, para hacer la llamada a la función `SumaComp2` se haría:

```
COMPLEJO a, b, suma;
/* bla bla bla */
suma = SumaComp2(&a, &b);
```

Para acceder a un elemento de una estructura a través de un puntero se puede utilizar el operador de indirección `*`, al igual que con el resto de variables. Siguiendo con el ejemplo anterior, la función calcularía la parte real del resultado como:

```
res.real = (*pc1).real + (*pc2).real;
```

En estos casos los paréntesis son estrictamente necesarios, pues tal como se dijo antes, el operador punto tiene una precedencia mayor que el operador `*`. Si no se ponen los paréntesis `*pc1.real` es equivalente a `*(pc1.real)`, lo cual quiere decir para el compilador que queremos acceder a la posición de memoria a la que apunte el puntero almacenado en el miembro `real` de la estructura `pc1`. Ahora bien, `pc1` no



Realice el ejercicio 1.

es una estructura sino un puntero a una estructura. Afortunadamente el compilador detectará este error, diciéndonos algo parecido a esto:³

“Error: request **for** member 'real' in something not a structure or **union**”

Que en una traducción un tanto libre al castellano quiere decir: “creo que se ha equivocado, pues ha solicitado acceder al miembro 'real' de algo que no es una estructura”.⁴

Como los punteros a estructuras se usan con bastante frecuencia en C (casi siempre dentro de funciones), existe una notación alternativa para acceder a un miembro de una estructura mediante un puntero, que es el operador “flecha” `->`.⁵ Así por ejemplo, las dos asignaciones siguientes son equivalentes:

```
res.real = (*pc1).real + (*pc2).real;
res.real = pc1->real + pc2->real;
```

Siendo la segunda mucho más cómoda de teclear y de interpretar.

10.7.1. Paso de punteros a estructuras a funciones

Tal como se dijo en la sección 10.6 el paso de estructuras de gran tamaño a funciones mediante copia es tremendamente ineficiente. En estos casos se puede pasar a la función la dirección de la estructura original en lugar de una copia de ésta. Ésto tiene la ventaja de aumentar considerablemente la rapidez del proceso de llamada de la función al tener que copiarse sólo un puntero a la estructura en lugar de la estructura completa. Sin embargo, dado que la función trabaja ahora con la estructura original en lugar de con una copia de ésta, se han de extremar las precauciones para no modificar accidentalmente el contenido original.

Por ejemplo, una función para imprimir una estructura del tipo `PACIENTE` sería:

```
void ImpriPaciente(PACIENTE *ppaciente)
{
    printf("Paciente %s:\n", ppaciente->nombre);
    printf("  Edad: %d\n", ppaciente->edad);
    printf("  Peso: %f\n", ppaciente->peso);
    printf("  N° Seg Soc.: %ld\n", ppaciente->n_seg_social);
    printf("  Historia: %s\n", ppaciente->historia);
}
```

Y una función para inicializar una estructura del tipo `PACIENTE` pidiéndole los datos al usuario sería:⁶

```
void LeerPaciente(PACIENTE *ppaciente)
{
    printf("Nombre del paciente: ");
    gets(ppaciente->nombre);

    printf("Edad del paciente: ");
```

³El mensaje de error es el generado por el compilador gcc. Otros compiladores generarán errores similares.

⁴Las uniones son parecidas a las estructuras, pero dado que su uso no es muy frecuente, no se han tratado en este libro.

⁵El operador flecha está formado por un `-` (símbolo menos) y un `>` (símbolo mayor que)

⁶La función `LeerPaciente` utiliza `printf` y `scanf` porque es una función de lectura. Como ya se ha comentado, las funciones de cálculo no realizan operaciones de entrada/salida.

```

scanf("%d", &ppaciente->edad);

printf("Peso del paciente: ");
scanf("%f", &ppaciente->peso);

printf("Nº de la Seguridad Social: ");
scanf("%ld", &ppaciente->n_seg_social);

while(getchar()!='\n'); /* scanf deja de leer en cuanto termina
    el número, dejando el \n del final de la cadena que introduce
    el usuario en el buffer de lectura. Si no se eliminan
    mediante este bucle los caracteres que scanf no ha leído,
    dichos caracteres serán leídos por gets y se almacenarán en
    la historia del paciente */
printf("Historial del paciente: ");
gets(ppaciente->historia);
}

```

Por último, un programa que muestra cómo se llamaría a ambas funciones es:

```

#include <stdio.h>

typedef struct paciente{
    char nombre[100];
    unsigned short edad;
    float peso;
    unsigned long n_seg_social;
    char historia[1000];
}PACIENTE;

void InitPaciente(PACIENTE *ppaciente);
void ImpriPaciente(PACIENTE *ppaciente);

int main(void)
{
    PACIENTE paciente;

    InitPaciente(&paciente);

    printf("\nLos datos del paciente son:\n\n");
    ImpriPaciente(&paciente);
    return 0;
}

```

Resulta interesante recordar que en la función `InitPaciente` es obligatorio trabajar con punteros porque debe modificar la estructura `paciente`. Sin embargo la función `ImpriPaciente` podría haberse escrito sin punteros, aunque en este ejemplo se han utilizado por los problemas de rendimiento comentados anteriormente.

10.8. Vectores de estructuras

Supóngase que se necesita realizar un programa que calcule el producto vectorial de dos vectores de números complejos. Una solución podría ser crear cuatro vectores, dos para las partes real e imaginaria del primer vector de números complejos y otros dos para las partes real e imaginaria del segundo vector de complejos. Sin embargo la solución ideal a este problema es crear dos vectores de estructuras de tipo COMPLEJO con las que se trabajará mucho más cómodamente.

La declaración de un vector de estructuras es idéntica a la de un vector de cualquier otro tipo de dato. Por ejemplo para crear un vector de 10 estructuras de tipo COMPLEJO basta con escribir:

```
COMPLEJO vector_complejo[10];
```

Y para acceder a los miembros de una estructura que a su vez es un elemento del vector se hace:

```
vector_complejo[0].real = 4.3;
vector_complejo[0].imag = 23.27;
...
vector_complejo[9].real = 2.1;
vector_complejo[9].imag = 3.7;
```

Nótese que como ocurre con el resto de vectores, la numeración de los elementos comienza en 0.

Conviene también tener en cuenta que al igual que con el resto de los vectores, al definir un vector de estructuras se reserva un espacio suficiente en la memoria para contener a las estructuras que componen el vector y se crea un puntero constante que apunta al comienzo de dicha zona de memoria. En este ejemplo `vector_complejo` es un puntero constante de tipo COMPLEJO * que apunta al principio del vector de cuatro estructuras de tipo COMPLEJO.

10.8.1. Ejemplo: Cálculo del producto escalar de dos vectores de complejos

Se desea realizar un programa que calcule el producto escalar de dos vectores de números complejos que el usuario introducirá por el teclado. La dimensión de los vectores será variable, pidiéndose al usuario al principio del programa.

Tal como se puede apreciar en el listado del programa, lo primero que se escribe es la declaración de la estructura complejo junto con la definición del “alias” COMPLEJO (líneas 16–19). De esta forma, a partir de la línea 19 el tipo COMPLEJO puede usarse al igual que el resto de tipos de datos de C, tal como se aprecia por ejemplo en los prototipos de las funciones (líneas 22 y 23).

```
/* Programa: PVecComp
2  *
   * Descripción: Pide al usuario dos vectores de números complejos
4  *               y calcula su producto escalar.
   *
6  * Revisión 0.0: 19/05/1999
   *
8  * Autor: El programador estructurado.
   */
10
```

```

#include <stdio.h>
12
#define MAX_DIM 100 /* Dimensión de los vectores de complejos */
14
/* Declaración de una estructura para contener un número
16     complejo */
typedef struct complejo{
18     double real;
    double imag;
20 }COMPLEJO;

22 /* Prototipos de las funciones */
void PideVec(COMPLEJO *pvec, int dim);
24 COMPLEJO ProdEsc(COMPLEJO *pvec1, COMPLEJO *pvec2, int dim);

26 int main(void)
{
28     COMPLEJO vec1[MAX_DIM], vec2[MAX_DIM]; /* Vectores introducidos
                                                por el usuario */
30     COMPLEJO res; /* Resultado del producto escalar */
    int dim; /* Dimensión de los vectores */
32
    printf("Introduzca la dimensión de los vectores: ");
34     scanf("%d", &dim); /* Ojo, falta control */

36     printf("\nIntroduzca los complejos del vector 1\n");
    PideVec(vec1, dim);
38
    printf("\nIntroduzca los complejos del vector 2\n");
40     PideVec(vec2, dim);

42     res = ProdEsc(vec1, vec2, dim);
    printf("\nEl producto escalar es: %lf + %lf i\n",
44         res.real, res.imag);
    return 0;
46 }

48 /* Función: PideVec
    *
50 * Descripción: Pide al usuario un vector de números complejos de
    *              dimensión dim
    *
52 *
    * Argumentos: COMPLEJO *pvec: Puntero al vector de estructuras
54 *              de complejos donde se escribirán los
    *              números introducidos por el usuario.
56 *              int dim: Dimensión del vector de complejos.
    */
58
void PideVec(COMPLEJO *pvec, int dim)
60 {

```



Realice el ejercicio 2.


```

    int n;
62
    for(n=0; n<dim; n++){
64        printf("Parte real del elemento %d: ", n);
        scanf("%lf", &(pvec[n].real) );
66        printf("Parte imaginaria del elemento %d: ", n);
        scanf("%lf", &(pvec[n].imag) );
68    }
    }
70
    /* Función: ProdEsc
72    *
    * Descripción: Calcula el producto escalar de dos vectores de
74    *                números complejos
    *
76    * Argumentos: COMPLEJO *pvec1: Punteros a los vectores de
    *                COMPLEJO *pvec2: complejos.
78    *                int dim: Dimensión de los vectores de complejos.
    *
80    * Valor devuelto: COMPLEJO: Resultado del producto escalar.
    */
82
    COMPLEJO ProdEsc(COMPLEJO *pvec1, COMPLEJO *pvec2, int dim)
84 {
    int n;
86    COMPLEJO resul;

88    resul.real = 0; /* Inicialización del resultado */
    resul.imag = 0;
90
    for(n=0; n<dim; n++){
92        resul.real += pvec1[n].real*pvec2[n].real -
                        pvec1[n].imag*pvec2[n].imag;
94        resul.imag += pvec1[n].real*pvec2[n].imag +
                        pvec1[n].imag*pvec2[n].real;
96    }
    return resul;
98 }

```

La función main consta simplemente de las definiciones de los vectores de estructuras (línea 27) y de las llamadas a las funciones PideVec y ProdEsc, que es donde se realiza todo el proceso del programa.

La función PideVec se encarga de pedir al usuario las partes real e imaginaria de cada elemento del vector y de almacenarlas en vector cuya dirección se le pasa como argumento (pvec). Nótese que se ha usado el operador corchete para acceder a cada uno de los elementos del vector. En la definición de la función se ha puesto COMPLEJO *pvec, aunque igualmente se podría haber puesto COMPLEJO pvec[].

Por último la función ProdEsc se encarga de calcular el producto escalar mediante la fórmula:

$$p = \sum_{i=0}^{dim-1} [\Re V_1(i) \cdot \Re V_2(i) - \Im V_1(i) \cdot \Im V_2(i)] + [\Re V_1(i) \cdot \Im V_2(i) + \Im V_1(i) \cdot \Re V_2(i)] \cdot i$$

En donde $\Re V_1(i)$ y $\Im V_1(i)$ son las partes real e imaginaria del elemento i del vector de complejos V_1 .

Para calcular el sumatorio de la fórmula anterior, en cada iteración del bucle **for** de las líneas 88–91 se calcula un sumando de dicho sumatorio y se acumula dicho sumando en el número complejo `resul`, que previamente se ha inicializado a cero (líneas 85–86). La función devuelve (mediante **return**) el valor de dicho número complejo al programa principal.

10.8.2. Vectores dinámicos de estructuras

Dado que el **typedef** de una estructura se comporta igual que cualquier tipo básico como pueda ser **int**, declarando un puntero a estructura se puede crear un vector dinámico de estructuras de una forma análoga al vector dinámico de enteros. Por ejemplo:

- Un vector de n enteros se crea:

```
int *pi;
pi = (int *) calloc(n, sizeof(int));
```

- De la misma forma, un vector con np pacientes se crea:

```
PACIENTE *pp;
pp = (PACIENTE *) calloc(np, sizeof(PACIENTE));
```



Véase la estructura de asignación de memoria en la sección de códigos típicos.

10.9. Recomendaciones y advertencias

- Es más cómodo declarar las estructuras usando **typedef**.
- También es recomendable usar el operador flecha (`->`) en lugar del operador `*` para acceder a los campos de una estructura a través de un puntero.
- Al trabajar con vectores de estructuras (estáticos o dinámicos) se utilizan los corchetes para indicar el índice del elemento del vector, al igual que con cualquier otro vector. Sin embargo hay que tener en cuenta que `pp[3].nombre` es el nombre del cuarto paciente del vector de pacientes `pp`; mientras que `pac.nombre[3]` es la cuarta letra del nombre de un paciente `pac`, que es una estructura normal, no un vector.
- Cuando las estructuras ocupen mucha memoria, lo cual ocurrirá si contienen una gran cantidad de campos o bien si estos campos son vectores o cadenas, es conveniente pasarlas a las funciones por referencia en lugar de por valor.

10.10. Resumen

En este capítulo se han estudiado las estructuras de datos en C, las cuales permiten crear variables compuestas por varios campos de distintos tipos. Se ha visto también cómo usar el operador punto para acceder a los campos de una estructura

y cómo usar el operador de asignación para copiar estructuras. También se ha visto que es posible pasar estructuras a funciones y que éstas las devuelvan, aunque si las estructuras ocupan una cantidad de memoria elevada, este proceso puede ser ineficiente, ya que al pasar estructuras como parámetros o devolverlas se copia toda la estructura.

También se ha visto cómo crear punteros a estructuras y cómo usar el operador flecha (->) para acceder a los campos de la estructura a través de un puntero a ella.

Para terminar se ha expuesto cómo crear vectores de estructuras, tanto estáticos como dinámicos.

10.11. Ejercicios

1. Modificar el programa de la sección 10.5 (pág. 138) para que la suma de los dos números complejos se realice mediante una llamada a la función `SumaComp` mostrada en la sección 10.6 (pág. 140).
2. Modificar el programa mostrado en la sección 10.8.1 (pág. 144) para impedir que el usuario introduzca una dimensión errónea para los vectores: si el usuario intenta introducir una dimensión mayor del tamaño máximo del vector `MAX_DIM` o menor o igual a 0, se dará un mensaje de error y se volverá a pedir la dimensión de nuevo.
3. Modificar el ejercicio anterior para crear los vectores dinámicamente. Una vez conocida la dimensión de éstos, se asignará memoria para los dos vectores mediante una llamada a la función `calloc`.
4. Modificar la función `ProdEsc` del ejercicio anterior para trabajar con el operador flecha -> en lugar del operador corchete [] para acceder a los elementos de los vectores de complejos en el cálculo del producto escalar. Se recomienda declarar 2 punteros auxiliares dentro de la función.
5. La organización de la Maratón popular villa de Laredo nos ha encargado la realización de un programa para calcular las velocidades medias de los corredores de la prueba. El programa pedirá al usuario el nombre del corredor (80 caracteres máximo), el número de dorsal y el tiempo empleado en recorrer los 42 km y 195 m, expresado en horas, minutos y segundos. Todos estos datos se almacenarán **obligatoriamente** en una estructura. A continuación el programa ha de calcular **mediante una función** la velocidad media en km/h y el tiempo medio empleado en recorrer cada kilómetro. Para finalizar, el programa imprimirá el número de dorsal, el nombre del corredor, su tiempo en horas minutos y segundos, su velocidad media en km/h y su tiempo medio por kilómetro, expresado en minutos y segundos. A continuación se muestra un ejemplo de ejecución:

```
Nombre del corredor: Pepe Joseón Canillas
dorsal: 218
tiempo:
  horas: 2
  minutos: 47
  segundos: 52
```

```
Dorsal: 218, Nombre: Pepe Joseón Canillas, Tiempo: 2:47:52
Velocidad media: 15.08 km/h, Tiempo medio por km: 3:58
```

CAPÍTULO 11

Archivos

11.1. Introducción

Cualquiera que haya manejado un ordenador habrá experimentado con desolación como un corte de corriente hace que desaparezca en un momento todo el trabajo que no había guardado en el disco. La razón de semejante desastre estriba en que la memoria RAM del ordenador se borra en cuanto se corta la corriente. Además la memoria es muy cara por lo que su cantidad es limitada, pero sin embargo es necesario que dentro de un ordenador se almacenen una gran cantidad de programas y de datos que suelen ocupar varios Gigabytes. Por todo esto es necesario un sistema de **almacenamiento secundario** que no se borre al cortar la corriente y que tenga una gran capacidad. El sistema actual de almacenamiento secundario más usado es el disco duro, dada su rapidez y capacidad de almacenamiento, aunque existen una gran variedad de sistemas como disquetes, memorias flash, CD, DVD, etc.

El manejo de todos estos sistemas lo realiza el sistema operativo, el cual organiza la información en forma de **archivos** repartidos en directorios (también llamados carpetas). De esta forma cada archivo tiene una identificación única dentro del ordenador compuesta por el nombre del archivo y por la **ruta** o camino de acceso hasta él por el árbol de directorios.

Hasta ahora los programas que hemos realizado en este libro han tomado datos desde el teclado y han impreso sus resultados en la pantalla. Sin embargo en muchas ocasiones es necesario leer datos desde un archivo que se ha almacenado previamente en el disco (bien por el propio usuario o bien por otro programa) o escribir los resultados en un archivo de forma que pueda ser usado posteriormente tanto por el usuario como por otro programa (por ejemplo en aplicaciones de bases de datos).

Antes de empezar a estudiar las funciones de la librería estándar que permiten trabajar con archivos, debemos distinguir entre dos tipos: los **archivos de texto** y los **archivos binarios**.

Los **archivos de texto** utilizan caracteres numéricos ('0' a '9') para representar los números y letras para representar texto. Son perfectamente legibles y se pueden ver con cualquier editor de texto (como por ejemplo el "bloc de notas" o "notepad" de Windows). Un ejemplo de este tipo de archivos es:

Rafael	607123456
Ana	609123456
Javier	915422800

Ejemplo.txt

Por el contrario, los **archivos binarios** se escriben copiando una imagen del contenido de una zona de la memoria al disco y por lo tanto los valores numéricos aparecen como unos caracteres extraños que se corresponden con la codificación de dichos va-

lores en la memoria del ordenador, pero que parecen sin sentido para el humano. Si se intenta abrir un archivo binario con el editor podemos encontrarnos con algo así:

```
Rafael±∞#$Ana+*õ
Javier♂%ª_
```

Ejemplo.dat

Los archivos de texto, que en este libro denominaremos con extensión `.txt` son ideales para intercambiar datos entre aplicaciones y para proporcionar datos de entrada de programas que se deban ejecutar muchas veces. Los archivos binarios, que denominaremos con extensión `.dat`, son más eficientes¹ y en consecuencia más apropiados para grandes volúmenes de información o bases de datos. Además este tipo de archivos permiten acceso directo. Esto permite, por ejemplo, leer el tercer registro sin tener que leer antes el primero y el segundo, lo cual es fundamental en bases de datos

11.2. Apertura de archivos. La función `fopen`

Antes de usar un archivo en disco es necesario decirle al sistema operativo que lo localice, que evite que otros procesos accedan al archivo mientras nuestro programa lo tenga abierto y que reserve unas zonas de memoria para trabajar con el archivo. Esto se realiza con la función `fopen` cuyo prototipo es:

```
FILE *fopen(char *Nombre_completo_del_archivo, char *modo);
```

En donde *Nombre_completo_del_archivo* es una cadena de caracteres que contiene el nombre del archivo, incluyendo el camino completo si dicho archivo no esta situado en el directorio actual. Este nombre ha de ser además un nombre legal para el sistema operativo. Por ejemplo si se trabaja en MS-DOS el nombre de archivo no puede tener más de ocho caracteres seguidos de un punto y tres caracteres más para la extensión.

El *modo* del archivo es otra cadena de caracteres que indica el tipo de operaciones que vamos a realizar con él. En el cuadro 11.1 se muestran todos los modos aceptados por la función.

Modo	Descripción
r	Abre el archivo para leer . El archivo ha de existir. Se posiciona al principio del archivo.
r+	Abre el archivo para leer y escribir . El archivo ha de existir. Se posiciona al principio del archivo.
w	Abre el archivo para escribir . Si el archivo existe, borra su contenido y si no existe, crea uno nuevo.
w+	Abre el archivo para escribir y leer . Si el archivo existe, borra su contenido y si no existe, crea uno nuevo.
a	Abre el archivo para añadir . Si el archivo no existe crea uno nuevo, pero si existe no borra su contenido. Se posiciona al final del archivo de forma que sucesivas escrituras se añaden al archivo original.
b	Ha de añadirse a cualquiera de los modos anteriores si el archivo es binario en lugar de ser de texto.

Cuadro 11.1: Modos de apertura de los archivos.

¹La razón de que sean más eficientes es porque no hace falta traducir la codificación interna de los números a código ASCII.

El valor devuelto por la función es un “puntero a archivo”, que es un puntero que apunta a una estructura de datos llamada FILE que está definida en `stdio.h` y que contiene toda la información que necesitan el resto de las funciones que trabajan con archivos, como el modo del archivo, los *buffers* del archivo, errores, etc. Si ocurre algún tipo de error en la apertura, como intentar abrir un archivo que no existe en modo “r”, `fopen` devuelve un NULL. Por tanto después de la llamada a `fopen` hay que verificar que el puntero devuelto es válido (al igual que se realiza con los punteros devueltos por `calloc` y `malloc`).

Por ejemplo si queremos abrir un archivo cuyo nombre es “pepe.txt” para leer su contenido habría que escribir:

```
#include <stdio.h>
...
int main(void)
{
    FILE *pfich;
    ...
    pfich = fopen("pepe.txt", "r");
    if(pfich == NULL){
        printf("Error: No se puede abrir el archivo.\n");
    }else{
        /* El archivo se ha abierto correctamente y podemos trabajar
           con él */
        ...
        fclose(pfich); /* Se explica más adelante */
    }
    return 0;
}
```

en donde se ha supuesto que el archivo “pepe.txt” está situado en el directorio de trabajo actual. En caso de que no exista el archivo “pepe.txt” o si no se encuentra en el directorio actual, `fopen` no podrá abrirlo y devolverá un NULL, por lo que el programa después de imprimir un mensaje de error terminará su ejecución.

Si estamos realizando un programa en Windows, queremos abrir un archivo de texto para añadir al final de él más datos, y dicho archivo está situado en el directorio `c:\dani` con el nombre `datos.m`, el programa sería igual que el anterior salvo que la llamada a `fopen` sería:

```
pfich = fopen("c:\\dani\\datos.m", "a");
```

Recuérdese que `\\` es traducido por el compilador de C por una `\` al generar la cadena de caracteres, de la misma manera que `\n` se traduce por un salto de carro. El archivo abierto será por tanto `c:\dani\datos.m`.

Si en cambio estamos trabajando en Linux, Mac OS X o cualquier otra versión de Unix y el archivo `datos.m` está situado en `/home/dani`, la llamada a `fopen` sería:

```
pfich = fopen("/home/dani/datos.m", "a");
```

En donde no hace falta hacer nada especial porque `/` es un carácter normal, no el carácter de escape `\\`.

11.3. Cierre de archivos. La función `fclose`.

Una vez que se ha terminado de usar un archivo hay que cerrarlo. Con la operación de cierre se “desconecta” el archivo del programa y se libera el puntero al archivo. Como la mayoría de los sistemas operativos tienen limitado el máximo número de archivos que un programa puede tener abiertos a la vez, es conveniente cerrar los archivos una vez que ya no se necesite usarlos. Sin embargo hay que ser cuidadoso para no seguir usando el puntero al archivo que se acaba de cerrar, pues los resultados serán impredecibles. Por si esto fuera poco, también es muy peligroso cerrar un archivo más de una vez.

Para evitar pérdidas de datos, cuando un programa termina normalmente² la función `fclose` se llama automáticamente para cerrar todos los archivos abiertos. A pesar de esto, es conveniente acostumbrarse a cerrar uno mismo los archivos que haya abierto por si acaso falla el cierre automático (pues ello ocasionaría pérdidas en nuestros datos) y así se hará en este libro.

El prototipo de la función `fclose` es:

```
int fclose(FILE *puntero_al_archivo);
```

En donde *puntero_al_archivo* es el puntero a archivo devuelto por la función `fopen` al abrir el archivo que se desea cerrar ahora.

El valor devuelto por la función es cero si el archivo se cerró con éxito o -1 si ocurrió algún tipo de error al cerrarlo; aunque independientemente de este resultado el archivo se cierra igualmente. Errores típicos pueden ser que el puntero que le pasamos no apunta a ningún archivo o que el disco se ha llenado y no se ha podido vaciar el *buffer* con la consiguiente pérdida de datos. Por esto último es conveniente verificar que el archivo se ha cerrado correctamente comprobando el valor de retorno, pues aunque no podemos hacer ya nada para solucionar la pérdida de datos, por lo menos podemos avisar al usuario de que ha ocurrido algún problema.

Para terminar veamos por ejemplo cómo se haría para cerrar el archivo que se abrió en el ejemplo de la sección 11.2:

```
if(fclose(pfich) != 0){
    printf("Error al cerrar el archivo\n"
           "Es posible que se haya producido una pérdida de "
           "datos.\nLo siento.\n");
}
```

Como resumen de lo expuesto anteriormente con relación a `fopen` y `fclose`, podemos presentar la siguiente lista de advertencias, que son análogas a las de asignación dinámica de memoria:

- Todo archivo abierto (con `fopen`) debe cerrarse (con `fclose`).
- No se debe intentar cerrar un archivo que no se ha abierto ni un archivo que ya ha sido cerrado.
- Si falla `fopen` no podemos utilizar el archivo.

²Aquí la palabra normalmente incluye cualquier salida controlada del programa, es decir, por la llegada al final del `main` o mediante una llamada a la función `exit`. Un programa que se aborta con Ctrl-C o que termina inesperadamente por un error de programación (división por cero, etc.) no se considera como terminado normalmente.

WEB

La estructura general para abrir y cerrar un archivo es análoga a la de pedir y liberar memoria. Una estructura general se muestra en la página web.

- Si una función abre un archivo, para evitar errores lo mejor es que la misma función lo cierre antes de salir.

La estructura general recomendada es:

```
fp = fopen("nombre", "r");
if(fp == NULL){
    printf("Error, no puedo abrir el archivo\n");
    /* Por tanto ni lo utilizo ni lo cierro */
}else{
    /* Utilizo el archivo */
    ...
    fclose(fp);
}
```

11.4. Lectura y escritura en archivos de texto. Las funciones `fprintf` y `fscanf`.

Para escribir y leer de archivos de texto se usan principalmente las funciones `fprintf` y `fscanf`, cuyo funcionamiento es **idéntico** al de `printf` y `scanf` respectivamente, salvo que ahora es necesario indicar en qué archivo han de escribir o leer.

11.4.1. La función `fprintf`

El prototipo de la función `fprintf` es:

```
fprintf(FILE *puntero_al_archivo, const char *cadena_de_formato, ...);
```

En donde `puntero_al_archivo` es el puntero a archivo devuelto por la llamada a `fopen` al abrir el archivo sobre el que se desea escribir. La `cadena_de_formato` es una cadena que especifica el formato en el que se imprimen el resto de argumentos de la llamada. Esta cadena se construye de la misma forma que la de la función `printf`, la cual se describió en el la sección 3.7. Por último los tres puntos (...) del prototipo de `fprintf` indican que a la `cadena_de_formato` le sigue un **número variable de parámetros**, los cuales se imprimirán según lo especificado en ella.

Por ejemplo, para escribir en el archivo abierto en la sección 11.2 no hay más que hacer:

```
fprintf(pfich, "El valor de %d en hexadecimal es %x\n", 15, 15);
```

Nota

Un programa puede trabajar sobre varios archivos a la vez, lo cual requiere declarar varias variables de tipo `FILE *` y asignar con `fopen` cada una de ellas a un archivo. Al especificar el puntero al archivo en cada llamada a `fprintf` al ordenador le queda totalmente claro sobre qué archivo tiene que imprimir .

11.4.2. La función `fscanf`

La función `fscanf` tiene por prototipo:

```
int fscanf(FILE *puntero_al_archivo, const char *cadena_de_formato, ...);
```


El valor devuelto por `fscanf` es el número de variables que se han leído correctamente. Si sólo se quiere leer un valor (como por ejemplo `fscanf(pfich, "%d", &num)`) el valor devuelto es 1 si se ha leído correctamente y distinto de 1 en caso de error. Para indagar un poco más en el tipo de error que se ha producido debemos saber que la función devuelve 0 si ha ocurrido un error en la lectura por una mala especificación del formato o un error en el archivo (por ejemplo si en este caso que se esperaba un entero (%d) en el archivo se encuentra una letra). Si no se ha leído nada porque se ha llegado al final del archivo, el valor devuelto será EOF.

Lo más habitual es leer varios valores en una misma llamada a `fscanf` (Por ejemplo `fscanf(pfich, "%d %d", &n1, &n2)`). En este caso `fscanf` devuelve el número de valores que se han leído y que por tanto estarán ya escritos en sus respectivas variables. Si ocurre un error antes de haber realizado alguna lectura, el comportamiento es igual al descrito anteriormente para el caso en el que sólo se leía un argumento (0 si hay error o EOF si se ha llegado al final del archivo). Si en cambio el error ocurre cuando ya se ha realizado alguna lectura, pero no todas, el valor devuelto es igual al número de valores leídos, independientemente de que el error producido haya sido de formato o de fin de archivo.

En conclusión, para saber si la lectura ha sido totalmente correcta basta con comprobar que el número devuelto por `fscanf` es igual al número de variables que se pretendía leer.

Para ilustrar el uso de `fscanf` supongamos que en el archivo abierto en la sección 11.2 hay escritos una serie de números en punto flotante de los cuales se desea calcular su suma. En el siguiente listado se muestra un programa que abre el archivo, lee los datos y calcula la suma:

```

2  /* Programa: Suma
   *
   * Descripción: Lee el archivo "pepe" que contiene números en
   *              punto flotante y calcula su suma.
   *
   * Revisión 1.0: 19/09/2005
   *
   * Autor: José Daniel Muñoz Frías.
   */
10
12 #include <stdio.h>
14 int main(void)
16 {
18     int ctrl;          /* Valor devuelto por fscanf */
19     FILE *pfich;       /* Puntero al archivo */
20     double dato;       /* dato leído del archivo */
21     double sum_tot;    /* Suma de los datos */
22
23     pfich = fopen("pepe", "r");
24     if(pfich == NULL){
25         printf("Error: No se puede abrir el archivo \"pepe\"\n");
26     }else{
27         sum_tot = 0; /* Inicialización de la suma antes de entrar en
28                     el bucle*/

```

WEB

En la página web se muestra un ejemplo completo de lectura de un archivo con `fscanf` vigilando el código de error devuelto por la función.

```

do{
28     ctrl = fscanf(pfich, "%lf", &dato);
    if(ctrl == 1){
30         sum_tot += dato;
    }
32 }while(ctrl == 1); /* termina en cuanto ocurra un error de
                        lectura*/
34
    printf("El valor de la suma es: %lf\n", sum_tot);
36
    if(fclose(pfich) != 0){
38         printf("Error al cerrar el archivo\n");
    }
40 }
return 0;
42 }

```

Este programa, si el archivo contiene:

```

1.3
3.4
4.5
123.4

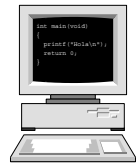
```

al ejecutarse imprimirá por pantalla:

El valor de la suma es: 132.6000000

Como se puede apreciar en la línea 27, el uso de `fscanf` es idéntico al uso de `scanf` salvo por la existencia del puntero al archivo desde donde se lee (`pfich`).

También cabe destacar la condición de salida del bucle **do-while**. Como se puede apreciar en la línea 31 se sale cuando el valor devuelto por `fscanf` sea distinto del número de variables que se quieren leer (1 en este caso). Nótese también que esta misma condición se usa para sumar el valor leído (`dato`) sólo cuando la lectura haya sido correcta (líneas 27–30).



Realice los ejercicios
1 y 2

11.4.3. Ejemplo usando archivos de texto

Para fijar ideas vamos a mostrar a continuación un ejemplo completo en el que se trabaja con archivos de texto. En los primeros tiempos de la informática sólo existían terminales en modo texto, por lo que cuando algún programa tenía que representar gráficamente una función se recurría frecuentemente a usar la impresora como salida gráfica. Vamos a retroceder por tanto unos cuantos años en la historia de la informática y vamos a realizar un programa que, partiendo de un archivo en el que están escritas las notas de los alumnos de programación, genere otro archivo con un histograma de las notas. En dicho histograma una barra representará el número de ocurrencias en el archivo de la nota correspondiente. Por ejemplo si el archivo de notas es el siguiente:

```

7.1
7.2
9.2
6.5
8.3

```

```

8.4
7.5
6.7
5.6
9.4
10.0
9.0
8.0
5.5
5.6
5.0
6.7
6.4
6.2
6.0
6.1
7.0
7.9
7.6
7.7
8.5
10.0

```

lo cual indica que todos los alumnos han estudiado un montón, el programa generaría el siguiente archivo:

```

0
1
2
3
4
5 ****
6 ****
7 ****
8 ****
9 ***
10 **

```

La forma de interpretar esta gráfica es la siguiente: como en la línea del 5 hay cuatro asteriscos, esto indica que hay cuatro notas entre 5 y 6 (sin incluir el 6) o dicho de una manera más formal, el número de notas contenidas en el intervalo $[5, 6)$ es cuatro. La excepción es el 10; su línea indica el número de notas iguales a 10.

El programa a realizar se puede dividir en varias tareas:

- Abrir los archivos.
- Leer el archivo de notas y contar cuantas notas están dentro de un intervalo dado. Para ello se usará un vector de enteros en el que en su primer elemento almacenaremos el número de notas contenidas en el intervalo $[0, 1)$, en el segundo las contenidas en $[1, 2)$ y así sucesivamente. La excepción será el elemento número 10 que sólo almacenará las notas iguales a 10.
- Escribir en un archivo el histograma. Para ello no hay más que imprimir una línea para cada intervalo, en la que se escribirán tantos asteriscos como notas

haya en dicho intervalo (lo cual está indicado en el número almacenado en el elemento del vector correspondiente a dicho intervalo).

La primera tarea se ha realizado dentro de la función main, pero las otras dos se han realizado en las funciones CuentaNotas y EscribeHisto respectivamente. El listado del programa completo se muestra a continuación:

```
/* Programa: ImpGraf
*
* Descripción: Lee un archivo que contiene las notas de los
*              alumnos (una nota (con decimales) por línea
*              almacenada) y genera otro archivo con un
*              histograma de las notas.
*
* Revisión 1.0: 19/09/2005
*
* Autor: José Daniel Muñoz Frías.
*/

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

void CuentaNotas(FILE *pfnot, int histo[]);
void CreaHisto(FILE *pfhis, int histo[]);

int main(void)
{
    int histo_not[11]; /* Vector para almacenar el número de
                        * ocurrencias de la nota correspondiente. Si histo_not[9]
                        * vale 50 es que 50 alumnos han sacado una nota entre 9
                        * y 10 (ánimo chicos/as) */

    FILE *pfnotas; /* Puntero al archivo de notas */
    FILE *pfhisto; /* Puntero al archivo de histograma */

    /* Se abren los dos archivos */
    pfnotas = fopen("notas.txt", "r");
    pfhisto = fopen("histo.txt", "w");

    if(pfnotas == NULL || pfhisto == NULL){
        printf("Error:\n");
        if(pfnotas == NULL){
            printf("No se ha podido abrir el archivo \"notas.txt\"\n");
        }else{
            fclose(pfnotas); /* Si se había abierto lo cierro antes de
                             salir */
        }
        if(pfhisto == NULL){
            printf("No se ha podido abrir el archivo \"histo.txt\"\n");
        }else{
            fclose(pfhisto);
        }
    }
}
```



```

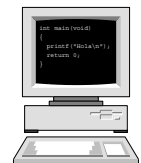
                                "columna" del histograma que
                                hay que incrementar */
        histo[i]++;
    }
    ctrl = fscanf(pfnot, "%lf", &nota_act);
    linea++; /* Hemos leído otra línea más */
}
}

/* Función: CreaHisto()
*
* Descripción: Representa "gráficamente" en un archivo el
*              histograma que se pasa en el vector histo. Para
*              ello por cada elemento de dicho vector se
*              transforma en una fila de asteriscos (un
*              asterisco representa una unidad)
*
* Argumentos: FILE *pfhis: Puntero al archivo de notas.
*              int histo[]: Histograma.
*
* Revisión 0.0: 25/05/1998.
*
* Autor: José Daniel Muñoz Frías.
*/

void CreaHisto(FILE *pfhis, int histo[])
{
    int i; /* Índice para recorrer el vector de notas */
    int ca; /* Índice para contar los asteriscos escritos */

    for(i=0; i<11; i++){
        fprintf(pfhis,"%2d ", i); /* Escribe la nota a la que corres-
                                   ponde la "barra" del histograma */
        /* Se escribe la barra de asteriscos */
        for(ca=0; ca < histo[i]; ca++){
            fprintf(pfhis, "*");
        }
        fprintf(pfhis, "\n"); /* Escribe el salto de línea para que la
                               siguiente nota se escriba en otra línea */
    }
}

```



11.5. Otras funciones de entrada y salida para archivos de texto

Hasta ahora toda la entrada y salida a archivos se ha realizado exclusivamente con las funciones `fprintf` y `fscanf`. Estas funciones tienen la ventaja de realizar las complejas conversiones de formato que son necesarias para poder transformar los valores binarios usados internamente en el ordenador a cadenas de caracteres usadas por los humanos. Por ejemplo antes de imprimir un número entero, que conviene recordar que se almacena como una secuencia de bits en la memoria del ordenador,

Realice los ejercicios
3 al 8

es necesario transformar esa secuencia de bits en una secuencia de caracteres que represente al número en la base elegida (normalmente base 10 (%d) o base 16 (%x). El inconveniente de esto es que el código encargado de realizar las conversiones ocupa memoria y tarda algo de tiempo en ejecutarse. Si embargo en la mayoría de los casos no es necesario escribir o leer números. Por ello en la librería estándar de C existen funciones que escriben o leen caracteres (o cadenas de caracteres) sin conversión de formatos. Estas funciones son `fgetc/fputc` para leer/escribir un carácter de/en un archivo y `fgets/fputs` para leer/escribir una cadena de caracteres de/en un archivo. Los prototipos de estas funciones son:

```
int fgetc(FILE *puntero_a_archivo);
int fputc(int carácter, FILE *puntero_a_archivo);
char *fgets(char *cadena, int tam_cad, FILE *puntero_a_archivo);
int fputs(const char *cadena, FILE *puntero_a_archivo);
```

La función `fgetc` lee el siguiente carácter desde el archivo como un **unsigned char** y lo devuelve convertido a **int**. Si se llega al final del archivo u ocurre algún error el valor devuelto es EOF.³

La función `fputc` escribe el *carácter* (convertido a **unsigned char**) que se le pasa como argumento en el archivo al que apunta *puntero_a_archivo*. El valor devuelto es el carácter escrito (convertido a **int**) o EOF si ha ocurrido algún error.

`fgets` lee una cadena de caracteres del archivo *puntero_a_archivo* y los almacena en la cadena de caracteres *cadena*. La lectura se acaba cuando se encuentra el carácter de nueva línea '\n' (que se escribe en la cadena), cuando se encuentra el fin del archivo (en este caso no se escribe '\n' en la cadena) o cuando se han leído *tam_cad* - 1 caracteres. En todos estos casos se escribe un carácter '\0' en la cadena a continuación del último carácter leído. Por supuesto la cadena de caracteres ha de tener espacio suficiente como para almacenar todos los caracteres que se puedan leer en la función (*tam_cad*). El valor devuelto es un puntero a la cadena leída o NULL si ha ocurrido algún error o se ha llegado al final del archivo.

`fputs` escribe la *cadena* en el archivo al que apunta *puntero_a_archivo*. El valor devuelto es un número positivo si se ha escrito la cadena correctamente o EOF en caso de error.

Para ilustrar el uso de estas funciones vamos a estudiar un ejemplo:

Imprimir un archivo

Vamos escribir en C un programa que imprima por pantalla el contenido de un archivo cuyo nombre se pide al usuario al principio del programa (el funcionamiento del programa será similar al del comando `type` de MS-DOS o al comando `cat` de Unix). El programa, después de abrir el archivo, leerá carácter a carácter el contenido del archivo e irá imprimiendo dichos caracteres por pantalla. Un pseudocódigo del programa es el siguiente

³Puede parecer un poco extraño el leer un carácter y devolverlo como un **int**. La razón de esto es la de poder distinguir los caracteres normales de los errores, pues si se devolviese el carácter leído como un **unsigned char**, al ocupar la tabla ASCII todos los valores que se pueden representar mediante un **unsigned char**, no quedan valores libres para representar el EOF. La solución es devolver un entero en el que los caracteres ocupan de 0 a 255 de forma que el resto de valores puedan usarse para devolver códigos de error.

Pedir al usuario el nombre del archivo que se ha de imprimir.
 Mientras no se llegue al final del archivo{
 leer un carácter del archivo
 imprimir el carácter
 }
 Cerrar el archivo.

Y un programa que realiza esta tarea se muestra a continuación.

```
/* Programa: ImpArch
 *
 * Descripción: Imprime el contenido del archivo cuyo nombre se
 *             pide al usuario por la pantalla.
 *
 * Revisión 0.0: 26/05/1998
 *
 * Autor: José Daniel Muñoz Frías.
 */

#include <stdio.h>

#define TAM_CAD 256

int main(void)
{
    char nom_fich[TAM_CAD]; /* Nombre del archivo que se va a abrir*/
    char letra; /* Carácter leído desde el archivo y escrito en
                 pantalla */
    FILE *pfich; /* Puntero al archivo que se imprime */

    printf("Introduzca el nombre del archivo que desea imprimir: ");
    fgets(nom_fich, TAM_CAD, stdin);
    if(nom_fich[strlen(nom_fich)-1] == '\n'){
        nom_fich[strlen(nom_fich)-1]= '\0'; /* Se elimina el \n si
                                              se ha leído */
    }

    pfich = fopen(nom_fich, "r");
    if(pfich == NULL){
        printf("Error al intentar abrir el archivo \"%s\"\n",
              nom_fich);
    }else{
        letra = fgetc(pfich);
        while(letra != EOF){
            printf("%c", letra);
            letra = fgetc(pfich);
        }

        if(fclose(pfich) != 0){
            printf("Error al cerrar el archivo.\n");
        }
    }
}
```



```

    }
    return 0;
}

```

Nótese que en la llamada a `fopen` el primer argumento es una variable de tipo cadena de caracteres en lugar de un nombre fijo (escrito entre comillas) como en los ejemplos anteriores.

11.6. Almacenamiento de estructuras. Archivos binarios

Según se ha comentado en la introducción, existen dos tipos de archivos en función del formato de almacenamiento: los archivos de texto y los archivos binarios. Aunque las estructuras se pueden almacenar en archivos de texto, normalmente se utilizan archivos binarios por su mayor rapidez y facilidad de programación.

11.6.1. Diferencia entre archivos de texto y archivos binarios

En la introducción se ha visto que los archivos de texto sólo contienen caracteres imprimibles como letras, números y signos de puntuación. Estos archivos se pueden abrir y modificar con un editor de texto como por ejemplo el “Bloc de notas” de Windows. Normalmente se escriben con la función `fprintf` que se encarga de convertir las variables numéricas a caracteres de acuerdo al formato especificado. Por ejemplo si una variable entera vale 2563, al escribir el valor de esta variable en pantalla mediante la función `printf` o en un archivo mediante la función `fprintf` se obtiene una secuencia de caracteres que depende de la especificación de formato. Con formato `"%5d"` se generan los 5 caracteres `' ', '2', '5', '6' y '3'`; mientras que con formato `"%06d"` se generan los caracteres `'0', '0', '2', '5', '6' y '3'`.

```

a=2563;
printf("a vale:%5d\n",a);      --->   a vale: 2563
printf("a vale:%06d\n",a);    --->   a vale:002563

```

En definitiva los archivos de tipo texto son legibles porque sólo contienen caracteres como números, letras y signos de puntuación. Las funciones `fprintf` y `fscanf` se encargan de hacer las traducciones de formato numérico a caracteres y viceversa.

Los archivos binarios se utilizan para almacenar la información como una copia exacta de la memoria del ordenador. Por ejemplo, si una variable entera corta vale 2563 estará almacenada en la memoria del ordenador en forma de 2 bytes (en un PC con Windows `sizeof(short int)` vale 2). De acuerdo a la codificación de los números enteros estos dos bytes serán (en hexadecimal) el 0A y el 03 o lo que es igual el 0000 1010 y el 0000 0011 en formato binario. Estos bits son los que realmente se encuentran almacenados en la memoria.

Cuando este número entero se guarda en un archivo binario se guardan 2 bytes dentro del archivo (como ya se ha dicho serán los bytes 0A y 03). Como norma general los archivos binarios no se pueden ver con un editor de texto, ya que sería una casualidad que los bytes que contienen fuesen caracteres legibles. Siguiendo con el ejemplo anterior, ni el carácter 0A ni el carácter 03 son imprimibles. También se puede decir que los archivos binarios y los programas que utilizan archivos binarios no son fácilmente portables a otros ordenadores u otros compiladores. Esto es evidente ya que el tamaño de las variables no es siempre igual: una variable tipo `int` ocupa 2 bytes en un PC con sistema operativo MS-DOS, pero ocupa 4 bytes en el mismo PC con un sistema operativo Windows o Linux. Además, no es sólo un problema de tamaño sino

que unos procesadores ordenan la memoria de una manera (primero 0A y luego 03) y otros al revés (primero 03 y luego 0A). Es importante tener en cuenta todas estas características de los archivos binarios cuando se adaptan programas que funcionan en una configuración determinada a otra configuración distinta o cuando se desea que programas que funcionan en plataformas distintas (PC y Mac por ejemplo) compartan archivos binarios.

11.6.2. Almacenamiento de estructuras en archivos de texto

Los miembros de una estructura pueden almacenarse individualmente dentro de un archivo de texto mediante la función `fprintf`. Para ello hay que abrir el archivo, escribir el valor de la estructura (campo por campo) y cerrar el archivo. En el siguiente programa de ejemplo se utiliza un vector de estructuras con el nombre y la cantidad de 2 productos de una tienda de *souvenirs*.

```
/*
Programa: texto.c
Descripción: Inicializa un vector de dos elementos tipo estructura
              y escribe todos los datos en un archivo de texto.
Revisión 0.0: 1/jun/1999
Autor: El programador de texto.
*/
#include <stdio.h> /* printf, fprintf, fopen, fclose */
#include <stdlib.h>
#include <string.h> /* strcpy */

#define N 2 /* Dimensión del vector (número de productos) */

/* Declaración de una estructura para los productos */
typedef struct {
    char nombre[12];
    short int cant;
} PROD;

int main(void)
{
    PROD producto[N]; /* Vector de productos en el almacén */
    int i; /* Contador para los bucles */
    FILE *f; /* Descriptor del archivo de texto */

    /*** Inicializo el vector ***/
    strcpy(producto[0].nombre, "Boina negra");
    producto[0].cant = 2563;
    strcpy(producto[1].nombre, "Botijo");
    producto[1].cant = 2628;

    /*** Creo el archivo ***/
    f=fopen("datos.txt", "w");
    if (f==NULL) {
        printf("Error al crear el archivo datos.txt\n");
    }else{
```

```

    for (i=0; i<N; i++) {
        fprintf(f, "%s\n", producto[i].nombre);
        fprintf(f, "%6d\n", producto[i].cant);
    }
    fclose(f);
}
return 0;
}

```

El archivo `datos.txt` que produce el programa anterior tiene cuatro líneas porque se ha puesto un carácter de cambio de línea al final de cada instrucción `fprintf`. El archivo tiene el siguiente aspecto:

```

Boina negra
  2563
Botijo
  2628

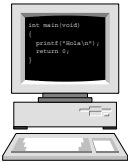
```

Los números no aparecen ajustados al margen izquierdo porque se ha puesto un formato `%6d` que escribe dos espacios delante de cada número de cuatro cifras. El archivo tiene en total 33 caracteres que son los siguientes:

```
Boina.negra␣..2563␣Botijo␣..2628␣
```

Nota

Se han representado los espacios como `␣` y los caracteres de cambio de línea como `␣` para facilitar la lectura.



Realice los ejercicios
9 y 10

11.6.3. Almacenamiento de estructuras en archivos binarios

Las estructuras pueden almacenarse dentro de un archivo binario mediante la función `fwrite`. Para ello hay que abrir el archivo en modo binario, escribir las estructuras y cerrar el archivo. La función `fwrite` no utiliza especificadores de formato sino que realiza un volcado de la memoria en el archivo. El prototipo de la función `fwrite` es:

```

size_t fwrite(void *pestructura, size_t tamaño, size_t numero,
              FILE *archivo);

```

Donde `size_t` es equivalente a **unsigned long int**, es decir se refiere a un número, y `void *` es un puntero genérico que vale para cualquier tipo de estructura. Por lo tanto, esta función recibe como argumentos un puntero a una estructura (la dirección de memoria de la estructura), luego el tamaño de la estructura en bytes (que se obtiene con **sizeof**), el número de estructuras que queremos guardar (que normalmente es una) y finalmente el descriptor del archivo en el cual queremos escribir. Conviene destacar que la existencia del parámetro `numero` permite escribir un vector completo de estructuras al archivo con una sola llamada a la función. En este caso, como es obvio, `pestructura` debe apuntar al principio del vector.

La función `fwrite` devuelve el número de estructuras escritas al archivo. Si ocurre un error de escritura (por ejemplo porque el disco se ha llenado), el número devuelto será menor que el número solicitado (parámetro `numero`). Para simplificar los ejemplos de este libro, dado que es muy raro que ocurra un error al escribir en un fichero, no se comprobará el valor devuelto por `fwrite`.

El siguiente programa de ejemplo es equivalente al programa del apartado anterior.

```

/*
Programa: binario.c
Descripción: Inicializa un vector de dos elementos tipo estructura
              y escribe todos los datos en un archivo binario.
Revisión 0.0: 1/jun/1999
Autor: El programador binario.
*/
#include <stdio.h> /* printf, fwrite, fopen, fclose */
#include <stdlib.h>
#include <string.h> /* strcpy */

#define N 2 /* Dimensión del vector (número de productos) */

/* Declaración de una estructura para los productos */
typedef struct {
    char nombre[12];
    short int cant;
}PROD;

int main(void)
{
    PROD producto[N]; /* Vector de productos en el almacén */
    int i; /* Contador para los bucles */
    FILE *f; /* Descriptor del archivo binario */

    /*** Inicializo el vector ***/
    strcpy(producto[0].nombre, "Boina negra");
    producto[0].cant = 2563;
    strcpy(producto[1].nombre, "Botijo");
    producto[1].cant = 2628;

    /*** Creo el archivo ***/
    f=fopen("datos.dat", "wb");
    if (f==NULL) {
        printf("Error al crear el archivo datos.dat\n");
    } else {
        for (i=0; i<N; i++) {
            fwrite(&producto[i], sizeof(producto[i]), 1, f);
        }
        fclose(f);
    }
    return 0;
}

```

Nótese que al abrir el archivo se especifica el modo `wb`. La `b` indica acceso binario. También conviene destacar que en la llamada a `fwrite` se podría haber escrito todo el vector de una vez, como se ha comentado antes, pero para mayor claridad se ha utilizado un bucle y se han escrito las estructuras de una en una.

El archivo `datos.dat` que produce el programa anterior no puede verse en un editor de texto porque contiene caracteres especiales. Ocupa en total 28 caracteres ya que cada estructura ocupa 14 bytes, que son 12 caracteres del primer nombre y 2 bytes

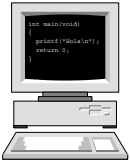
para el **short int** de la cantidad. Por lo tanto el archivo puede representarse como:

Boina-negra??Botijo???????

Nota

Se han representado los espacios como , los caracteres `'\0'` de final de cadena como y otros caracteres no imprimibles como `?`. En este caso no hay cambios de línea. Los 12 primeros caracteres corresponden al nombre del primer producto, que es un vector de **char** de longitud 12 que en este caso particular están todos inicializados (11 de Boina-negra y el 12º del carácter NULL que pone la función `strcpy`). A continuación viene el número 2563 codificado como entero corto que ocupa 2 bytes (que son el 0A y el 03 en hexadecimal). Los siguientes 12 caracteres corresponden al nombre del segundo producto, que ahora sólo tiene 7 caracteres ocupados (6 de Botijo y el `'\0'`) el resto de caracteres de la cadena no están inicializados y por tanto pueden contener cualquier cosa, por lo que en el ejemplo se han representado también por `?`. Otra manera de representar el contenido del archivo es la siguiente:

```
'B' 'o' 'i' 'n' 'a' ' ' 'n' 'e' 'g' 'r' 'a' '\0' 0A 03
'B' 'o' 't' 'i' 'j' 'o' '\0' ??? ??? ??? ??? ??? 0A 44
```



Realice los ejercicios
11 al 13

11.6.4. Lectura de estructuras desde archivos binarios

Para leer estructuras de un archivo binario se utiliza la función `fread` cuyo manejo es equivalente a la función `fwrite` que se ha visto anteriormente, tal como se puede apreciar en su prototipo:

```
size_t fread (void *estructura, size_t tamaño, size_t numero,
              FILE *archivo);
```

Al igual que la función `fwrite`, `fread` realiza un acceso directo a la memoria del ordenador a partir de la dirección de la estructura. La función `fread` lee del archivo `tamaño*numero` bytes y los guarda a partir de la dirección de memoria `estructura`. Al igual que `fwrite`, `fread` devuelve un valor de tipo `size_t` que es el número de estructuras que ha sido capaz de leer. El valor devuelto será igual al valor solicitado (`numero`) salvo que se produzca un error o se llegue al final del archivo. En el caso de `fread` es muy importante verificar si la lectura ha sido correcta.

Para ilustrar el uso de la función `fread` se muestra a continuación un programa que lee el archivo generado por el programa de la sección anterior y muestra su contenido por la pantalla.

```
/*
2 Programa: binario1.c
  Descripción: Lee los datos de una serie de productos desde
4              un archivo binario, escritos por el programa
                binario.c y los almacena en un vector de
6              estructuras.
  Revisión 0.1: 11/abr/2006
8 Autor: El programador binario.
  */
10 #include <stdio.h> /* printf, fwrite, fopen, fclose */
    #include <stdlib.h>
```



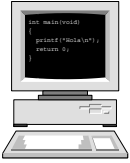
En la página web
hay un ejemplo de
lectura de un archi-
vo binario para
contar todos sus
registros.

```

12  #define N 100 /* Dimensión del vector (número de productos) */
14  /* Declaración de una estructura para los productos */
16  typedef struct {
17      char nombre[12];
18      short int cant;
19  }PROD;
20
21  int main(void)
22  {
23      int n_datos; /* Número de datos leídos del archivo */
24      int err;     /* Código de error devuelto por fread */
25      PROD aux;
26      PROD producto[N]; /* Vector de productos en el almacén */
27      int i;       /* Contador para los bucles */
28      FILE *pf; /* Descriptor del archivo binario */
29
30      /** Abro el archivo **/
31      pf = fopen("datos.dat", "rb");
32      if (pf==NULL) {
33          printf("Error al abrir el archivo datos.dat\n");
34      } else {
35          n_datos = 0;
36          do {
37              err = fread(&aux, sizeof(aux), 1, pf);
38              if(err == 1){ /* Datos leídos correctamente */
39                  producto[n_datos] = aux;
40                  n_datos++;
41              }
42          } while(err == 1);
43          fclose(pf);
44      }
45
46      /* Imprimimos los datos leídos */
47      printf(" Producto  Cantidad\n");
48      printf("-----\n");
49      for (i=0; i<n_datos; i++) {
50          printf("%-11s  %4d\n", producto[i].nombre, producto[i].cant);
51      }
52      return 0;
53  }

```

Hay varias cosas que destacar de este programa. En primer lugar hay que hacer notar que la definición de la estructura en la que se leen los datos del archivo ha de ser **exactamente igual** a la de la estructura usada para escribirlos, tal como se puede apreciar en las líneas 16–19. En segundo lugar fíjese que el archivo se ha abierto en modo lectura (línea 31). Por último conviene destacar que este programa lee todos los datos y los carga en un vector, pero podría ir leyendo y escribiendo los datos de uno



Realice el ejercicio 14.

en uno hasta que `fread` llegase al final del archivo.⁴ En este caso no habría límite de tamaño.

11.6.5. Acceso directo en archivos binarios

Una de las características más importantes de los archivos binarios de estructuras es que cada registro ocupa un espacio constante, que es el tamaño de la estructura. Esto permite avanzar o retroceder en el archivo para ir a leer un registro concreto, es decir, se permite el **acceso directo** a los datos. La función que permite moverse en un archivo a una posición concreta se llama `fseek` y tiene el siguiente prototipo:

```
size_t fseek(FILE *archivo, long desplazamiento, int origen);
```

La función desplaza `desplazamiento` bytes el índice de acceso del archivo. El parámetro `origen` especifica el origen del desplazamiento y puede ser:

1. `SEEK_SET`. El origen es el principio del archivo. **Es la opción más normal.**
2. `SEEK_CUR`. El origen es la posición actual. Permite avanzar o retroceder (usando un desplazamiento negativo).
3. `SEEK_END`. El origen es el final del archivo. Permite ir rápidamente al final del archivo.

Ejemplos:

```
fseek(fp, 0, SEEK_SET); /* Va a la primera posición del archivo */
fseek(fp, 10*sizeof(producto[0]), SEEK_SET); /* Al 11º producto */
fseek(fp, 2*sizeof(producto[0]), SEEK_CUR); /* Salta 2 productos */
```

Normalmente se utiliza siempre `SEEK_SET` y se especifica la posición absoluta a la que se quiere acceder dentro del archivo. Es muy habitual abrir un archivo binario y leer todos los registros hasta el final con objeto de contarlos. A partir de ese momento se puede utilizar el archivo como si fuese un vector si se realiza un `fseek` delante de cada `fread` o `fwrite`. De esta manera se pasa de manejar volúmenes de datos del orden de MB (vectores limitados por la RAM del ordenador) a volúmenes del orden de GB (archivos binarios limitados por la capacidad del disco duro). Obviamente el aumento del tamaño no viene exento de inconvenientes: el tiempo de acceso es varios órdenes de magnitud mayor en el caso de archivos en disco (del orden de ms) que en los accesos a memoria RAM (del orden de ns).

Una función equivalente a `fseek(fp, 0, SEEK_SET)` pero más fácil de escribir es `rewind(fp)`.

11.7. Recomendaciones y advertencias

- Salvo casos excepcionales, si una función abre un archivo, debe cerrarlo cuando termine de usarlo.
- Una función que recibe un puntero al archivo como argumento asume que está ya abierto. Además tampoco debe cerrarlo, pues esa tarea debe realizarla la función que lo ha abierto.

⁴Lo que se detecta comprobando que el código de error devuelto por `fread` es distinto de 1, tal como se aprecia en la línea 38.



Ver ejemplo para contar el número de registros que contiene un archivo en la sección de Códigos Típicos.

- Aunque existe una función denominada `feof` para comprobar si la lectura ha llegado al final del archivo, es conveniente averiguarlo analizando los códigos de error devueltos por las funciones de lectura `fscanf` o `fread`; ya que en algunas situaciones `feof` puede dar problemas.
- Aunque al puntero devuelto por `fopen` se le denomina puntero al archivo, en realidad es un puntero a la estructura de control del archivo y sólo debe de usarse para pasárselo a las funciones que trabajan con archivos (`fprintf`, `fread`, `fclose`, etc.). Bajo ningún concepto ha de cambiarse su valor, por ejemplo incrementándolo, ya que a partir de ese momento no podremos hacer nada con el archivo.
- En relación con la advertencia anterior, para acceder al siguiente carácter de un archivo basta con llamar a una función de lectura o escritura, como por ejemplo `fgetc`. El sistema operativo mantiene un índice internamente que le dice qué byte del archivo fue el último al que se accedió y realiza el siguiente acceso a partir de ahí, incrementando este índice automáticamente. Bajo ningún concepto incrementa el puntero al archivo para acceder al siguiente byte de un archivo. Si desea acceder a un Byte concreto dentro del archivo use la función `fseek`.

11.8. Resumen

En este capítulo se han estudiado las posibilidades que nos ofrece el lenguaje C para el manejo de los archivos. Se ha visto que existen dos tipos de archivos: de texto que sólo contienen caracteres como letras y números y binarios que contienen una imagen de la memoria y por tanto no son directamente imprimibles.

Se ha mostrado que el proceso para trabajar con un archivo consiste en abrirlo en primer lugar, leer o escribir los datos que se necesiten y por último cerrarlo cuando ya no se necesite acceder más a él.

También se ha visto que para leer o escribir en archivos de texto existen las funciones `fscanf` y `fprintf` cuyo uso es similar al de `scanf` y `printf`. Sin embargo el acceso a archivos binarios es totalmente distinto, existiendo las funciones `fread` y `fwrite` para transferir bloques de datos entre la memoria RAM y el disco.

11.9. Ejercicios

1. ¿Cuál sería la salida del programa mostrado en la sección 11.4.2 (pág. 153) si el archivo de datos de entrada fuera el siguiente?

```
2.3
3
hola tío
4
5.89
```
2. Modifique el programa de la sección 11.4.2 (pág. 153) para que muestre el resultado de la suma sólo si el archivo se ha leído hasta el final o que muestre un mensaje de error si hay un fallo en el archivo.
3. Modificar el programa de la sección 11.4.3 (pág. 155) para que sea fácil cambiar el carácter con el que se imprime el histograma. Para ello usar la sentencia **#define** (por ejemplo **#define CAR_HISTO '#'**).

4. Realizar lo mismo que en el ejercicio anterior pero pidiéndole al usuario al principio del programa el carácter con el que se desea que se imprima el histograma.
5. En el programa de la sección 11.4.3 (pág. 155), la función CuentaNotas termina en cuanto fscanf no devuelva un 1, lo cual puede ocurrir o bien porque se haya terminado el archivo o bien porque se haya producido un error de formato. Modifique la función para que en el segundo caso se imprima un mensaje de error por la pantalla indicando esta situación y la línea del archivo de notas en la que se ha detectado el error.
6. Aún con la modificación realizada en el ejercicio anterior, si hay un error de formato en el archivo (por ejemplo una palabra en lugar de un número) se termina la lectura, aunque el archivo contenga más datos. Modificar la función escrita en el problema anterior para que aunque se detecte un error de formato, la lectura del archivo continúe hasta que se llegue al final. En este caso se imprimirán todos los errores encontrados en el archivo, tanto notas fuera de rango (menores que 0 y mayores que 10) como datos no numéricos. **Nota:** tenga en cuenta que cuando fscanf encuentra un carácter no numérico en el archivo, lo deja ahí, es decir, no incrementa el índice de acceso al archivo; con lo que las siguientes llamadas a fscanf volverán a encontrarse con el mismo carácter y volverán a dar el mismo error de formato. Para evitar que el programa se quede en un bucle sin fin, cuando fscanf detecte un valor no numérico será necesario leer con fgetc los caracteres que queden en la línea.
7. Realizar un programa que a partir de un archivo que contiene una serie de números (en el mismo formato que el usado para el archivo de notas) calcule la media y la varianza de la serie y las imprima por pantalla. Para ello se aconseja usar funciones que lean el archivo y que devuelvan el resultado de sus cálculos. Así por ejemplo para calcular la media se podría crear una función con el prototipo **double** media(FILE *pfich);.
8. Realizar un programa que a partir del archivo de notas cree un archivo con el histograma de notas y a continuación imprima la media y la varianza de las notas. **Nota:** Para volver a leer un archivo desde el principio se puede usar la función rewind, cuyo prototipo es: **void** rewind(FILE *pfich);.
9. Comprobar que en el programa realizado en la sección 11.6.2 (pág. 163) cada estructura escribe un número diferente de caracteres en el archivo.
10. ¿Cuál sería el archivo resultante si los fprintf del programa de la sección 11.6.2 (pág. 163) fuesen los siguientes?:

```
fprintf(f, "%15s\n", producto[i].nombre);
fprintf(f, "%-6d\n", producto[i].cant);
```

¿Cuántos caracteres ocuparía el archivo?

11. Comprobar que en el programa de la sección 11.6.3 (pág. 164), en lugar del bucle **for** se podía haber escrito la siguiente línea:

```
fwrite(&producto[0], sizeof(producto[0]), N, f);
```

Esta instrucción escribe las N estructuras del vector producto de una sola vez.

12. En el programa de la sección 11.6.3 (pág. 164), ¿Cuántos caracteres ocuparía el archivo si la cantidad se guardase en una variable tipo **long** en lugar de tipo **short int**?
13. En el programa de la sección 11.6.3 (pág. 164), ¿Cuál sería el archivo resultante si el nombre del primer producto fuese "Camiseta" y la cantidad 200?
14. ¿Qué ocurrirá en el programa de la sección 11.6.4 (pág. 166) si el archivo binario tiene más de 100 estructuras? Modifique el programa para que funcione correctamente en este caso.
15. Modifique el programa de la sección 11.6.4 (pág. 166) para que pueda leer un archivo con cualquier número de estructuras.
16. Escribir un **programa** para leer el contenido de un archivo y mostrarlo por pantalla. El programa debe abrir el archivo `pepe.txt`, luego leerá carácter a carácter hasta encontrar el final del archivo e irá escribiendo en pantalla. Finalmente cerrará el archivo. La lectura carácter a carácter puede realizarse mediante `fgetc(pf)` o mediante `fscanf(pf, "%c", &c)`. La escritura por pantalla debe realizarse mediante `printf("%c", c)`.
17. Modifique el programa anterior para que el nombre del archivo a imprimir se le pida al usuario. Si el archivo no puede ser abierto se dará un mensaje de error y terminará el programa.
18. Escribir un programa para leer un archivo y guardarlo en una cadena de caracteres. En primer lugar, el programa debe leer el archivo `datos.txt` con la función `fgetc()`. Se leerá todo el archivo desde el principio hasta el final contando cuántos caracteres tiene. A continuación se asignará memoria suficiente a un puntero a **char** y se volverá a leer el archivo almacenando todos los caracteres en el vector.
19. Escribir un **programa** para leer una serie de números enteros desde el teclado y escribirlos en un archivo llamado `pepe.txt`. El programa debe abrir el archivo en modo escritura y luego, dentro de un bucle, leer un valor entero con `scanf()` y escribirlo a continuación en el archivo con `fprintf()`. La condición de salida del bucle será la introducción por el usuario del valor 0. Finalmente se cerrará el archivo antes de abandonar el programa. No se deben utilizar vectores.
20. Modifique el programa del ejercicio anterior para que la condición de salida del bucle sea la introducción por el usuario de cualquier valor no numérico (por ejemplo: "Ya me he cansado de introducir números". Tenga en cuenta que `scanf` devuelve los mismos códigos de error que `fscanf`.
21. Escribir un programa que abra el archivo de texto `entrada.txt` en modo de lectura y cree un archivo llamado `salida.txt`. El archivo de entrada sólo contiene texto (Letras mayúsculas, minúsculas o números). El programa irá leyendo los caracteres del primer archivo y los escribirá en el segundo, pero cambiando las mayúsculas por minúsculas y las minúsculas por mayúsculas, así si el archivo `entrada.txt` contiene:

Hola, soy el Archivo de ENTRADA 123.

una vez ejecutado el programa, el fichero `salida.txt` contendrá:

hOLA, SOY EL aRCHIVO DE entrada 123.

La lectura puede realizarse carácter a carácter con `fscanf` o con `fgetc`. La escritura puede realizarse con `fprintf` o con `fputc`.

Para realizar el programa puede usar las funciones de la librería estándar:

- **int isupper(int c)** Devuelve un valor distinto de cero si el carácter `c` es mayúscula y cero si no lo es.
- **int islower(int c)** Devuelve un valor distinto de cero si el carácter `c` es minúscula y cero si no lo es.
- **int toupper(int c)** Si el argumento `c` es minúscula, devuelve la letra mayúscula correspondiente. Si no lo es devuelve el carácter sin cambiar.
- **int tolower(int c)** Si el argumento `c` es mayúscula, devuelve la letra minúscula correspondiente. Si no lo es devuelve el carácter sin cambiar.

Los prototipos de estas funciones están en el archivo cabecera `ctype.h`.

22. Los organizadores de las próximas Olimpiadas nos ha seleccionado para realizar un programa que mejore la presentación de los resultados de la prueba de 1500 m. Resulta que el sistema de cronometraje automático genera un archivo en el que sólo se escribe el número del dorsal del corredor junto con su tiempo en centésimas de segundo. Sin embargo este formato es muy pobre para su presentación tanto a la prensa como a los espectadores, por lo que es necesario realizar un programa que a partir del archivo generado por el sistema de cronometraje, genere otro en el que se incluya el número de dorsal, el tiempo expresado en minutos, segundos y centésimas de segundo y la velocidad media del corredor durante la prueba en kilómetros por hora (con dos decimales).

El archivo con los datos del sistema de cronometraje se llama `tiempos.txt` y el archivo que ha de generar nuestro programa se llamará `salida.txt`.

Por ejemplo si el archivo `tiempos.txt` contiene:

```
961 20765
527 20873
365 21057
357 21134
363 21183
```

El archivo de salida generado por el programa realizado por el alumno ha de ser:

```
Dorsal 961  Tiempo 3'27.65''  Velocidad Media 26.01 km/h
Dorsal 527  Tiempo 3'28.73''  Velocidad Media 25.87 km/h
Dorsal 365  Tiempo 3'30.57''  Velocidad Media 25.64 km/h
Dorsal 357  Tiempo 3'31.34''  Velocidad Media 25.55 km/h
Dorsal 363  Tiempo 3'31.83''  Velocidad Media 25.49 km/h
```

APÉNDICE A

Consejos de Estilo de Programación en C

A.1. Estructura del programa

La estructura básica de un programa de un sólo módulo está compuesta por una cabecera, instrucciones **#include**, instrucciones **#define**, instrucciones **typedef**, prototipos, la función **main**, y el resto de las funciones (ver ejemplo al final). Con este orden se garantiza la coherencia de todas las definiciones, mientras que utilizar un orden diferente sólo sería válido en algunas situaciones particulares.

Además es necesario crear una cabecera de comentarios para documentar cada módulo y cada función.

A.2. Variables

Escribir los nombres de variables en minúsculas.

Si el nombre es compuesto, utilizar el guión bajo para separar palabras. Por ejemplo: `suma_complejos`.

Es conveniente que el nombre de las variables tenga sentido, para ayudar a entender el programa. Es habitual utilizar algún criterio para identificar el tipo de variable por su nombre, por ejemplo que todos los punteros a entero utilicen nombres de variables empezando por `pi_`.

Es muy recomendable añadir un comentario al lado de la declaración para explicar mejor su significado. Declarar muchas variables es una sola línea impide escribir el comentario.

Evitar utilizar variables globales. Sólo en casos muy especiales tiene sentido utilizar variables globales; en esos casos hay que documentar en cada función qué variables globales se utilizan.

Es recomendable inicializar variables justo delante del bucle que requiere la inicialización. Esto facilita la revisión del programa, evita que la inicialización se pueda perder antes de llegar al bucle, y facilita la reutilización de fragmentos de código. En cualquier caso el código es más claro si la inicialización se realiza en una línea diferente a la declaración de la variable.

No inicializar variables que no requieran inicialización. Esto anula la capacidad del compilador de detectar el uso de una variable antes de haberle dado valor (por ejemplo cuando uno se confunde de nombre de variable u olvida el signo `&` en un `scanf`).

A.3. Funciones

Escribir los nombres de las funciones con la primera letra mayúscula y las siguientes minúsculas.

Si el nombre es compuesto, utilizar mezclas de minúsculas y mayúsculas. Por ejemplo: `SumarComplejos()`. Resulta más claro utilizar nombres de acción para nombres de funciones.

Cada función debe llevar su propia cabecera de descripción, que es equivalente al manual de referencia.

Evitar escribir funciones muy largas. Es conveniente dividir una función larga en varias funciones pequeñas para facilitar la comprensión del programa y la depuración, aunque dichas funciones sólo se llamen una vez.

A.4. Claridad del Código

Seguir las recomendaciones de estilo de nombres de parámetros y tipos (mayúsculas), nombres de variables (minúsculas) y nombres de funciones (mezcla).

En general utilizar nombres en español para objetos declarados por el programador. Esto permite diferenciarlos de los objetos propios del lenguaje que son nombres en inglés.

Utilizar un correcto sangrado del texto para localizar rápidamente el inicio y el final de cada bloque de código.

No escribir líneas de código demasiado largas. En C todas las expresiones matemáticas, expresiones lógicas y llamadas a función se pueden separar en varias líneas. Las cadenas de caracteres se pueden separar en varias líneas cerrando las comillas en una línea y abriéndolas en la siguiente.

No utilizar operaciones crípticas, porque puede no quedar clara la instrucción. Por ejemplo, evite códigos como este: `if ((b[i]=a[++i])==0) ...`

Por otro lado, algunas de estas operaciones son ambiguas. Por ejemplo `a = ++c + c` es dependiente del compilador [Kelley and Pohl, 1984].

Además, aunque algunas operaciones están bien definidas en el lenguaje, no resultan intuitivas. Por ejemplo: `*p++` no es igual a `(*p)++`

Tenga en cuenta que escribir instrucciones complejas en varias líneas y utilizar paréntesis evita ambigüedades y ayuda a entender el código.

Es más seguro que las llamadas a `calloc` y `free`, o las llamadas a `fopen` y `fclose` aparezcan dentro de la misma función. Asignar memoria en una función y liberar en otra supone mayor riesgo de no hacer las cosas bien. Son una excepción a esta regla las estructuras especiales de datos como listas y árboles, o las funciones específicas para asignación de memoria.

En programas con interfaz gráfica es fundamental separar las funciones de control de la interfaz de las funciones de cálculo. Esto permite depurar las funciones de cálculo de manera independiente y con programas auxiliares, además de facilitar la portabilidad si se decide cambiar de herramienta gráfica.

Debe utilizarse la estructura de bucle adecuada para cada caso: **for**, **while** o **do-while**. Como norma general el bucle **for** se utiliza cuando el número de iteraciones es fijo y conocido desde el principio. Por lo tanto la condición de salida del bucle **for** no debería ser muy compleja, ni tiene sentido modificar el contador dentro del bucle.

Las salidas a mitad de bucle con **break** o **continue** están totalmente desaconsejadas. Estas instrucciones se pueden evitar utilizando bloques **if** o añadiendo variables de control de flujo en los bucles **while** y **do-while**

Está desaconsejada la utilización de `exit` en cualquier parte del código y la utilización de **return** en medio de una función. Salvo casos excepcionales, resulta más claro que las funciones tengan un único punto de entrada y un único punto de salida.

A.5. Documentación

En apuntes, trabajos, enunciados de prácticas y exámenes, facilita la comprensión utilizar tipos de letra monoespaciados para escribir el nombre de funciones como `printf`.

Los listado de programas deben imprimirse utilizando un tipo de letra monoespaciado (como por ejemplo Courier o también Lucida Console) y un tamaño adecuado. La conversión de tabuladores a espacios evita que se descoloque el texto.

A.6. Ejemplo

```

/*
PROGRAMA: Ejemplo de Estilo. Programa para calcular la suma de un
          conjuntos de números complejos
FECHA: 5/may/2003
Autor: Rafael Palacios
*/

#include <stdio.h>

#define N 10

typedef struct {
    double real;    /* Parte Real */
    double imag;    /* Parte Imaginaria */
} COMPLEJO;

COMPLEJO SumarComplejos(COMPLEJO a[], int n);

void main(void)
{
    COMPLEJO v[N]; /* vector de complejos */
    int n;         /* número de elementos */
    int i;         /* contador */
    COMPLEJO suma; /* suma de los complejos */
    /*** Pido el número de elementos ***/
    do {
        printf("Cuantos elementos quieres introducir? ");
        scanf("%d",&n);
        if (n>N) {
            printf("Este programa no soporta tantos elementos\n");
        }
    } while (n>N);
    /*** Pido los n elementos ***/
    for(i=0; i<n; i++) {
        printf("Parte real del elemento %d: ",i+1);
        scanf("%lf",&v[i].real);
        printf("Parte imaginaria del elemento %d: ",i+1);
        scanf("%lf",&v[i].imag);
    }
}

```

```

    /** Calcula la suma de todos los complejos **/
    suma=SumarComplejos(v,n);
    /** Salida de resultados **/
    printf("La suma es: (%lf,%lf)\n",suma.real, suma.imag);
}
/* ***** F U N C I O N E S ***** */
/*
Función: SumarComplejos
Descripción: Calcula la suma de un vector de complejos
Argumentos:
    a vector de complejos
    n número de elementos válidos del vector
Valor retornado:
    suma complejo resultados
Advertencias:
    En caso de error, por ejemplo el número de elementos es menor
    que 1, la función devuelve el complejo (0,0);
*/
COMPLEJO SumarComplejos(COMPLEJO a[], int n)
{
    COMPLEJO ret; /* valor a retornar */
    int i; /* contador */
    ret.real=0;
    ret.imag=0;
    for(i=0; i<n; i++) {
        ret.real += a[i].real;
        ret.imag += a[i].imag;
    }
    return ret;
}

```

APÉNDICE B

Consejos para Programación Segura y Estable en C

B.1. Controlar Buffer Overflow

Poner todos los controles necesarios para evitar acceder (lectura o escritura) fuera del espacio de memoria de vectores y matrices.

Leer cadenas de caracteres mediante la función `fgets`, en lugar de `gets`, `scanf` o `fscanf`.

En caso de duda sobre el tamaño de las cadenas de caracteres utilizar `strncpy`, `strncpy` y `strncat`, en lugar de `strcpy`, `strcmp` o `strcat`.

Es muy peligroso, y en la mayoría de los casos incorrecto, hacer asignaciones del tipo `cadena="Hola";`.

B.2. Medidas para evitar bucles infinitos

En bucles **while** y **do-while** comprobar que las variables que intervienen en la condición se modifican dentro del bucle. Verificar que para cualquier valor inicial, existe solución o verificar el rango de valores iniciales válidos antes de entrar en el bucle.

En funciones recursivas, comprobar que las variables que intervienen en la condición de salida se modifican antes de hacer la siguiente llamada recursiva.

B.3. Desconfiar del valor de las variables

Hay que tener en cuenta que los valores de las variables pueden no ser válidos o estar fuera de rango, por lo tanto:

- Verificar si los rangos son válidos antes de realizar operaciones críticas (teniendo en cuenta resultados intermedios de la operación): división (¿es cero?), raíz cuadrada (¿es negativo?), operaciones con enteros (¿se produce desbordamiento?).
- Controlar los errores de lectura de archivos o de teclado. Todas las funciones de lectura devuelven códigos de error o proporcionan la información necesaria para detectarlos.

B.4. Estabilidad del programa

Inicializar variables siempre que sea necesario para evitar comportamientos imprevistos.

Llevar buen control de la memoria dinámica:

- Evitar perder memoria asignada, pues el programa agotaría la memoria del sistema.
- Garantizar que el programa nunca libera memoria no asignada (o libera memoria dos veces).

Tener en cuenta casos poco probables: el archivo no existe, el disco está lleno, no hay memoria suficiente, la conexión no responde, etc. Es decir, hay que verificar los valores retornados por las funciones para controlar los errores (imprescindible: `fopen`, `malloc`, `calloc`, `realloc`).

APÉNDICE C

Contenidos disponibles en la web del libro

En la página web del libro <http://www.iit.upcomillas.es/libroc> se puede acceder a los siguientes contenidos:

- Compilador de C de dominio público.
- Ejemplos de código de uso frecuente.
- Presentaciones de los temas más importantes.
- Referencia *on-line* de las funciones estándar de C.
- Hoja resumen de C.
- Contribuciones de los lectores.
- Fe de erratas de este libro.

Bibliografía

- [Antonakos and Mansfield, 1997] Antonakos, J. L. and Mansfield, K. C. (1997). *Programación estructurada en C*. Prentice Hall Iberia, Madrid.
- [Delannuy,] Delannuy, C. *El Libro de C como Primer Lenguaje*. Eyrolles, ediciones Gestión 2000 S. A., Barcelona.
- [Kelley and Pohl, 1984] Kelley, A. and Pohl, I. (1984). *A Book on C. An introduction to programming in C*. The Benjamin/Cummings publishing company, Inc., Menlo Park, CA.
- [Kernighan and Ritchie, 1991] Kernighan, B. W. and Ritchie, D. M. (1991). *El lenguaje de programación C*. Prentice Hall, segunda edición.
- [Roberts, 1995] Roberts, E. S. (1995). *The Art and Science of C*. Addison Wesley.

Índice alfabético

- #define**, 104
- #include**, 14
- 0X, 22
- 0x, 22
- API, 11
- Archivo
 - cabecera, 23
- Archivos
 - binarios, 149, 162, 164
 - acceso directo, 168
 - cabecera, 14
 - de texto, 149, 162, 163
- ASCII, 7
- Biblioteca de funciones, 13
- break**, 68, 174
- Código fuente, 12
- Código objeto, 12
- calloc, 125, 174, 178
- char**, 21
- Compilación, 12
- Constante
 - carácter, 22
 - secuencias de escape, 23
 - L, 22
 - real, 22
 - U, 22
 - UL, 22
- continue**, 70, 174
- do-while**, 48, 174
- double**, 20
- Edición, 11
- Enlazado, 13
- exit, 56, 174
- fclose, 152, 174
- fgetc, 160
- fgets, 102, 103, 160, 177
- Fibonacci, 92
- float**, 20
- fopen, 150, 174, 178
 - FILE, 151
 - modo del archivo, 150
 - nombre del archivo, 150, 151
- for**, 40, 174
- fprintf, 153
- fputc, 160
- fputs, 160
- fread, 166
- free, 126, 174
- fscanf, 153, 177
- fseek, 168
- Función recursiva, 85, 177
- fwrite, 164
- gets, 102, 177
- IEEE 854, 7
- if**, 55
- if-else**, 55
- int**, 20
- islower, 172
- ISO 10646, 7
- ISO 8859, 7
 - tabla, 8
- isupper, 172
- long**, 21
- malloc, 125, 178
- Numeración
 - base 10, 3
 - base 16, 4
 - base 2, 4
 - base 8, 4
 - rango, 6
- Operador
 - *, 117

++, 33
--, 33
=, 31, 34, 36
%, 33
&, 116
asignación, 31, 34, 36, 177
cast, 35
decremento, 33
flecha (->), 142
incremento, 33
lógico, 58
molde, 35
punto, 137
relacional, 57
resto, 33
Ordenador
 buses, 3
 CPU, 2
 entrada/salida, 3
 memoria, 2

Pila, 84
Preprocesador, 14
printf, 23–27
Programa ejecutable, 13

Recursividad, 85
referencia, paso por, 107, 110
return, 77, 82, 83, 90, 174
rewind, 170

scanf, 27, 177
short, 21
sizeof, 23
standard input, 103
stdin, 103
strcat, 177
strcmp, 101, 177
strcpy, 100, 177
strlen, 98
strncat, 177
strncmp, 103, 177
strncpy, 103, 177
struct, 135
switch-case, 63

tolower, 172
toupper, 172
typedef, 136

Unicode, 7

unsigned, 21

valor, paso por, 107, 110
Variables de control, 41

while, 45, 174

Ayuda para la programación en C

Estructura de un programa C

```
/*
Programa de Ejemplo
Fecha_
Autor_
*/
#include ____
#define ____
typedef ____
[Prototipos]

int main(void)
{
    [variables] /* descripcion */

    [instrucciones]
    return 0;
}
```

Caracteres especiales

'\n' cambio de línea (newline)
'\r' retorno de carro
'\0' carácter 0 (NULL)
'\t' TAB
'\"' comilla simple '
'\"' comilla doble "
'\\' la barra \

Formatos de printf y scanf

%d int
%hd short
%ld long
%u unsigned int
%hu unsigned short
%lu unsigned long
%f float, double
%lf double (sólo scanf)
%c char
%s cadena de caracteres

Operadores

Aritméticos int: + - * / %
Aritméticos double: + - * /
Otros aritméticos: ++ -- += -= *= /=
Lógicos y relacionales:
> < >= <= == != && || !

Bucles

Bucle for

```
for(inicialización, condición, instrucción_fina) {
    [instrucciones]
}
```

Ejemplo: for(i=0; i<10; i++)

Bucle while

```
while (condición) {
    [instrucciones]
}
```

Bucle do-while

```
do {
    [instrucciones]
} while(condición);
```

Bloque if

caso 1:
if (*condición*) {
 [instrucciones]
}

caso 2:

```
if (condición) {
    [instrucciones_1]
} else {
    [instrucciones_2]
}
```

caso 3:

```
if (condición_1) {
    [instrucciones_1]
} else if (condición_2) {
    [instrucciones_2]
...
} else if (condición_n) {
    [instrucciones_n]
} else {
    [instrucciones]
}
```

Sintaxis del switch

```
switch(expresión_entera) {
case constante_1:
    [instrucciones_1]
    break;
case constante_2:
    [instrucciones_2]
    break;
...
case constante_3:
    [instrucciones_3]
    break;
default:
    [instrucciones]
}
```

Vectores y matrices

```
double vector[10];
char cadena[256];
char matriz[10][20];
```

```
vector[2]=3;
scanf("%lf",&vector[7]);
```

Cadenas de caracteres

```
char cadena[N];
```

Lectura:

```
scanf("%s",cadena);
    lee una palabra
```

```
gets(cadena);
```

lee una frase hasta fin de línea

```
fgets(cadena, N, stdin);
```

lee una frase con control de tamaño. También lee \n

Escritura:

```
printf("%s",cadena);
```

escribe una cadena por pantalla, vale para frase o palabra

Funciones estandar de string.h

```
size_t strlen( char *str );
    devuelve la longitud de la cadena
```

```
strcpy( char *to, char *from );
    copia o inicializa
```

```
int strcmp(char *s1, char *s2 );
    compara las cadenas s1 y s2
0 → s1 es igual a s2
<0 → s1 es menor que s2
>0 → s1 es mayor que s2
```

Funciones

Prototipo:

tipo NombreFun(*tipo* var1, ... , *tipo* varN);

Estructura de la función:

```
tipo NombreFun(tipo var1, ... , tipo varN)
/* Descripción general
Argumentos: ...
Valor Retornado: ...
Advertencias de uso: ...
*/
{
    [variables locales]

    [instrucciones]

    return expresión;
}
```

Ejemplos de prototipos y llamadas:

```
int Sumar(int a, int b);
void Cambio(int *a, int *b);
double CalcularMedia(double a[], int n);
float Traza(float mat[][20], int n, int m);

res=Sumar(x,y);
Cambio(&x, &y);
med=CalcularMedia(vec,n);
tra=Traza(mat,n,m);
```

Asignación Dinámica de Memoria

```
char *pc;
```

```
pc=(char *)calloc(100, sizeof(char));
pc=(char *)malloc(100*sizeof(char));
pc=(char *)realloc(pc, 200*sizeof(char));
free(pc); /*libera memoria */
```

Estas funciones devuelven NULL en caso de error

Estructuras

Declaración de un tipo estructura

```
typedef struct persona {
    char nombre[N];
    int edad;
    long dni;
} PERSONA;
```

Declaración de variables:

```
PERSONA p; /* una estructura */
PERSONA *pp; /* puntero a estructuras */
PERSONA vec[20]; /* vector de estructuras */
```

Acceso a los miembros:

```
p.edad=27;
pp->edad=30;
vec[7].edad=37;
```

Declaración de listas enlazadas:

```
typedef struct lista {
    char nombre[N];
    int edad;
    long dni;
    struct lista *siguiente;
} LISTA;
```

Archivos

Abrir y cerrar

```
FILE *fopen(char *nombre, char *modo);
    Devuelve NULL en caso de error
    modo="r" Lectura
    modo="r+" Lectura (y escritura)
    modo="w" Escritura
    modo="w+" Escritura (y lectura)
    modo="a" Añadir al final
    modo="a+" Añadir al final (y lectura)
    modos=rb, rb+, wb, wb+, ab, ab+ binario
int fclose(FILE *puntero al archivo);
    Devuelve 0 si no hay error
```

Archivos de texto

```
int fscanf(FILE *fp, char *cadena_formato, ...);
    Devuelve el número de variables leídas
    Devuelve 0 si no hay podido leer ninguna variable
    Devuelve EOF si ha llegado al final de fichero
int fprintf(FILE *fp, char *cadena_formato, ...);
char *fgets(char *cadena, int tam_cad, FILE *fp);
    Devuelve el puntero a la cadena si no hay error
    Devuelve NULL en caso de error
int fputs(char *cadena, FILE *fp);
```

Archivos binarios (acceso directo)

```
int fwrite(void *variable, size_t tamaño, size_t num, FILE *fp);
int fread(void *variable, size_t tamaño, size_t num, FILE *fp);
    Devuelve el número de elementos leídos, normalmente num
```

```
int fseek(FILE *fp, long desplazamiento, int origen);
    El tercer argumento puede tomar los valores: SEEK_SET (comienzo), SEEK_END (final), SEEK_CUR (actual)
```

Otras Funciones generales

```
int fgetc(FILE *fp);
    Devuelve el caracter leído (lo devuelve como int)
    Devuelve EOF si ha llegado al final de fichero

int fputc(int caracter, FILE *fp);
int feof( FILE *fp );
    Devuelve distinto de cero si estamos al final del fichero. En caso contrario, devuelve cero

void rewind( FILE *fp );
    Vuelve al principio del archivo. Equivale a fseek(fp,0,SEEK_SET);
```

