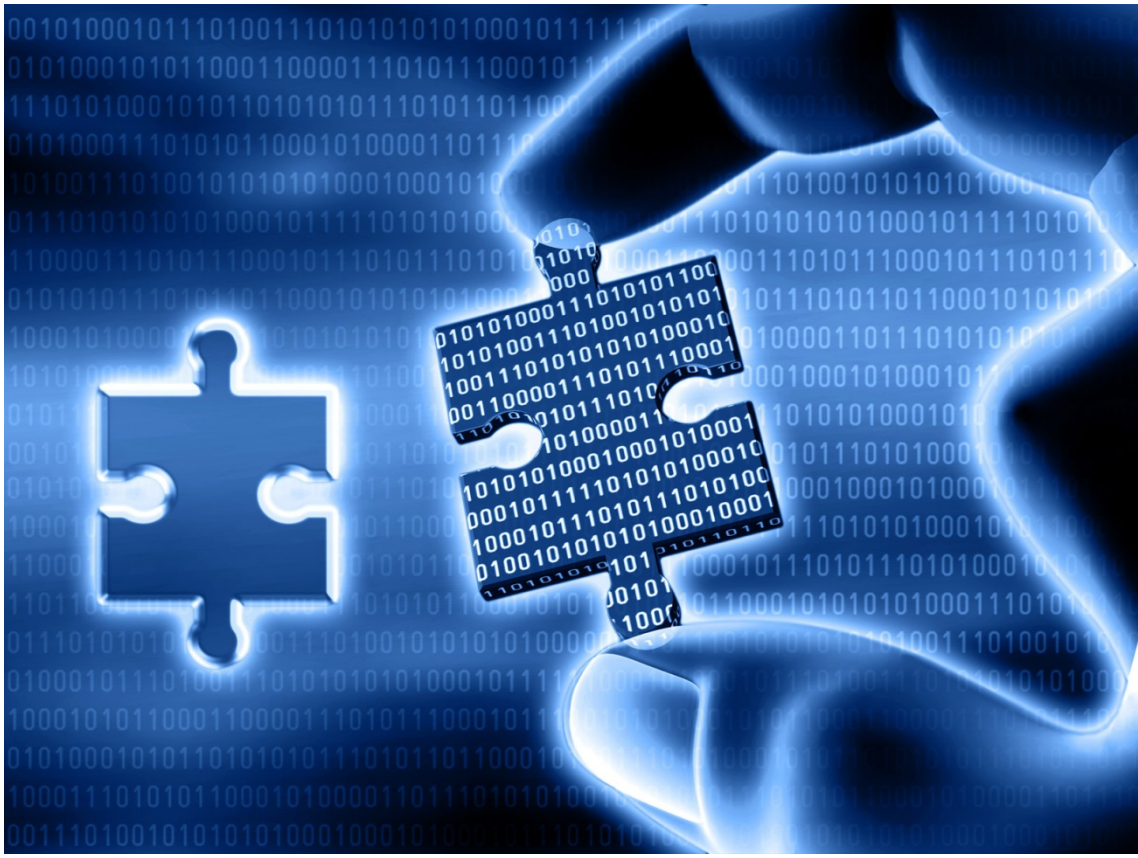
industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

INFORMÁTICA APLICADA
PROGRAMACIÓN EN LENGUAJE C



Universidad Politécnica de Cartagena

Pedro María Alcover Garau

Lenguajes y Sistemas Informáticos

INFORMÁTICA APLICADA.

PROGRAMACIÓN EN LENGUAJE C.

Pedro María Alcover Garau

Área de Lenguajes y Sistemas Informáticos
Universidad Politécnica de Cartagena

Septiembre, 2010
(Revisado Enero, 2011)
(Revisado Agosto, 2011)
(Revisado Agosto, 2012)

Edita Universidad Politécnica de Cartagena
Agosto 2011.

ISBN: 978-84-693-9245-4

D.L. MU–1533–2010.

Imprime Morpi, S.L.


¡BUENO!
NO DEJES PARA MAÑANA LO QUE PUEDAS HACER HOY


¡DESDE MAÑANA MISMO EMPIEZ

ÍNDICE.

PRESENTACIÓN

Hay una recomendación clásica, que los profesores de programación repetimos frecuentemente a los alumnos: en el aprendizaje de un lenguaje de programación hay que programar: oír permite olvidar; estudiar permite entender; sólo programar permite aprender. No basta con ir a clase. No basta con estudiar este ladrillo. Hay que instalar un compilador y un editor de programas y ponerse delante de la pantalla. Y programar.

El aprendizaje de la programación es una tarea inicialmente ingrata. Las horas de trabajo y estudio no parecen cundir. A una hora se sucede otra, en la que el aprendiz no entiende apenas sobre qué se le está instruyendo. Y otra hora…, y otra…, y otra…

Y uno se enfrenta al primer programa —una auténtica simpleza— y no logra más que una colección de 26 errores. ¡Veintiséis!: ¡en tan sólo cinco líneas de código!

Hay que seguir. Y detectar el punto y coma que falta en esa línea; y el paréntesis que en realidad debía ser una llave; y el archivo de cabecera, que no sé bien para qué sirve y que para colmo el compilador no me la

reconoce, porque la he deletreado mal; y mil pedradas más que animan a cualquier cosa menos a seguir trabajando con el dichoso lenguaje C. Todos hemos tenido que aprender a programar. Para todos, esos primeros pasos han sido una personal pesadilla.

Cada año, en el primer día de clase con los nuevos alumnos, dibujo en la pizarra del aula una gráfica que muestra el rendimiento del estudio de una materia como ésta de un lenguaje de programación (cfr. Figura 0.1.). Es una curva a la que le cuesta levantarse del eje de las abscisas. Muestra, en su primera fase, una desesperante pendiente casi nula. Pero, en un entorno de valores determinado, distinto para cada estudiante, esa gráfica sufre un cambio de pendiente, casi de forma súbita. Estudio y no comprendo… estudio y no comprendo… estudio y… ¡ah!, ¡ahora lo veo!: un pequeño avance. Y con el paso de las horas, de repente se clarifican un bloque de conceptos, casi de golpe. Y el estudio cambia de cariz: el aprendiz se pica con la programación. Ha llegado el momento de avanzar a buen ritmo. Se disfruta aprendiendo, porque se logran cosas nuevas, cada vez más difíciles. Se caza el modo en que se deben plantear las cosas al ordenador. Se coge la filosofía de cómo se construyen los programas. A uno se le ocurren nuevos retos, y se atreve a intentar buscar una solución. Y se aprende entonces a programar. Y, por supuesto, se aprueba la asignatura.

Lo que he contado en estas líneas previas es estadísticamente cierto. Porque a juzgar por las notas de los alumnos, o los exámenes son muy fáciles, o los exámenes son muy difíciles. Porque la media de los aprobados siempre está en el notable alto, con no pocos sobresalientes. Y la media de los no aprobados no alcanza al 2. Porque el que no llega a franquear el tiempo de estudio de inflexión, se queda en nada o en casi nada. Y el que lo supera, ya después con poco esfuerzo alcanza un buen nivel. Que programar no es difícil; pero el inicio es la más difícil de las fases de este aprendizaje.

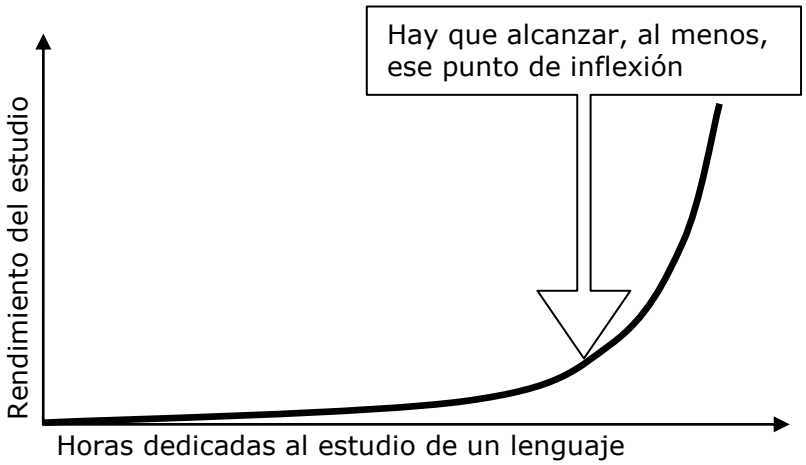


Figura 0.1.

Voy a insistir en esta idea del estudio y trabajo personal, porque es clave. Casi se puede garantizar que aquellos alumnos que dediquen sus seis horas semanales a trabajar y estudiar esta asignatura, lograrán, al final del recorrido, aprender y asimilar los conocimientos propuestos en este curso; y es más que probable que superen con buena nota el examen y la asignatura. Y casi se puede garantizar que aquellos alumnos que no trabajen con constancia, y no dediquen un tiempo semanal a implementar código, no aprenderán ni comprenderán unos conocimientos mínimos; y es más que probable que no superen la asignatura: ni de lejos. No sueñe con aprender a programar en un encierro de fin de semana, encadenado a la pantalla. Programar es un oficio que se adquiere programando con constancia. Es la constancia en el trabajo la que hace el oficio.

Muchos alumnos creen haber aprendido a programar cuando logran comprender un código que da solución a un problema planteado. Esa percepción es un engaño. Al poco de iniciar el estudio de un lenguaje de programación, cualquiera puede entender los problemas sencillos que aquí, en este manual, se plantean. Pero programar no es entender el

código escrito por otros: es crear código nuevo de la nada. Creo que no pierde el tiempo quien, en los inicios de su estudio, invierte una mañana entera para resolver una simpleza; e incluso tampoco ha echado el tiempo a perder quien, después del esfuerzo prolongado, no ha llegado a una solución válida. Y tengo certeza, evidencia empírica, de que quien se contenta con comprender las soluciones propuestas por otro no logra aprender: ni mucho, ni poco; y no logra aprobar.

En Agosto de 2012 se ha publicado un nuevo manual de prácticas, complementario a éste y, como éste, disponible en el Repositorio Digital de la UPCT y en nuestro magnífico servicio de Reprografía. El nuevo manual está editado por el Centro Universitario de la Defensa (CUD) de San Javier. Es fruto de dos años de trabajo docente de los dos autores, en dos cursos académicos consecutivos, con los alumnos de la Escuela de Industriales de la UPCT y los del CUD en la Academia General del Aire. Ese manual de prácticas marca una pauta sistemática de trabajo.

Creo que no le va a faltar documentación. Ahora hace falta que usted encuentre tiempo para hacer buen uso de ella. Ánimo.

Agradeceré recibir todas las sugerencias que ayuden a presentar de forma más clara todo lo que se pretende explicar en este manual. Así se podrá ofrecer, a quienes vengan detrás, una versión mejorada. Se puede contactar conmigo a través del correo electrónico. Mi dirección es pedro.alcover@upct.es.

Muchas gracias.

Cartagena, 15 de agosto de 2012

CAPÍTULO 1

INTRODUCCIÓN Y CONCEPTOS GENERALES.

El objetivo de este capítulo primero es introducir algunos conceptos básicos manejados en el ámbito de la programación, y algunas palabras de uso habitual entre quienes se ven en la necesidad de programar: léxico común, de poca complejidad, pero que es necesario conocer bien.

Se presenta muy sucintamente una descripción de la arquitectura de las computadoras, que permita comprender de forma intuitiva cómo trabaja un ordenador con las instrucciones que configuran un programa y con los datos que maneja en su ejecución; y cómo logra un programador hacerse entender con una máquina que tiene un sistema de comunicación y de interpretación completamente distinto al humano.

Un ordenador es un complejísimo y gigantesco conjunto de circuitos electrónicos y de multitud de elementos magistralmente ordenados que logran trabajar en total armonía, coordinación y sincronía, bajo el

gobierno y la supervisión de lo que llamamos sistema operativo. Es un sistema capaz de procesar información de forma automática, de acuerdo con unas pautas que se le indican previamente, dictadas mediante colecciones de sentencias o instrucciones que se llaman programas; un sistema que interactúa con el exterior (el usuario a través del teclado o del ratón, internet, el disco de almacenamiento masivo, etc.), recibiendo los datos (información) a procesar, y mostrando también información; un sistema, además, que permite la inserción de nuevos programas: capaz por tanto de hacer todas aquellas cosas que el programador sea capaz de expresar mediante sentencias y codificar mediante estructuras de información.

Dos son los elementos fundamentales o integrantes del mundo de la programación: los datos y las instrucciones. Un programa puede interpretarse únicamente de acuerdo con la siguiente ecuación: **programa = datos + instrucciones**.

Para lograr alcanzar el principal objetivo de este manual, que es el de marcar los primeros pasos que ayuden a aprender cómo se introducen y se crean esos programas capaces de gestionar y procesar información, no es necesario conocer a fondo el diseño y la estructura del ordenador sobre el que se desea programar. Pero sí es conveniente tener unas nociones básicas elementales de lo que llamamos la arquitectura y la microarquitectura del ordenador.

Cuando hablamos de **arquitectura** de un ordenador nos referimos a la forma en que ese ordenador está construido; a la distribución física de sus componentes principales; una desripción de su funcionalidad. Al hablar de **microarquitectura** nos referimos más bien a la forma concreta en que el ordenador implementa cada una de las operaciones que puede realizar.

Hay diferentes arquitecturas. La más conocida (la que se presenta aquí) es la llamada de **Von Neumann**. La característica fundamental de esta arquitectura es que el ordenador utiliza el mismo dispositivo de

almacenamiento para los datos y para las instrucciones. Ese dispositivo de almacenamiento es lo que conocemos como memoria del ordenador. Desde luego, es importante conocer esta arquitectura: ayuda a comprender sobre qué estamos trabajando.

Estructura funcional de las computadoras

Un esquema muy sencillo que representa la estructura básica de una computadora queda recogido en la Figura 1.1.

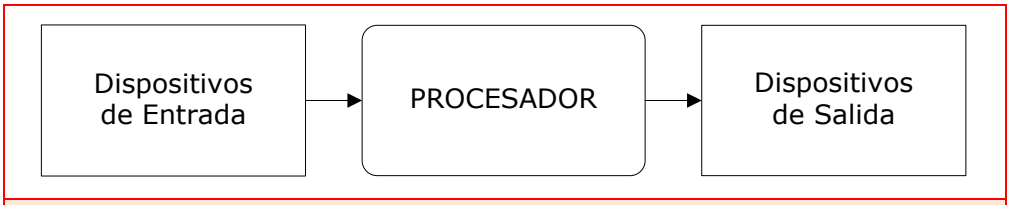


Figura 1.1. Estructura básica de una computadora

El procesador recibe los datos desde los dispositivos de entrada (por ejemplo, teclado o ratón) y los muestra, o muestra los resultados del procesamiento, en los dispositivos de salida (por ejemplo pantalla o impresora). Los dispositivos de memoria masiva (disco duro, lápiz USB, disquete, CD/DVD…) son dispositivos de entrada y de salida de información: el procesador puede leer la información grabada en ellos y puede almacenar en ellos nueva información. A estos dispositivos de memoria les llamamos de memoria masiva.

La estructura básica del procesador, de acuerdo con la "**arquitectura de Von Neumann**" queda esquematizada en la Figura 1.2. Von Neumann definió, de forma abstracta y conceptual, cuales debían ser las partes principales de la arquitectura de una computadora. Después de décadas de gran desarrollo tecnológico, este diseño arquitectónico no ha sufrido apenas cambios.

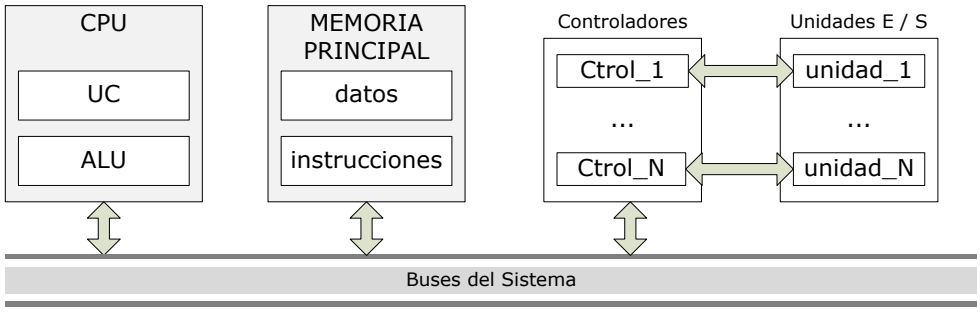


Figura 1.2. Arquitectura de Von Neumann.

Cinco son los elementos básicos de esta arquitectura: (1) la Unidad de Control (UC); (2) la Unidad Aritmético Lógica (ALU); (3) la Memoria Principal, donde se almacenan los datos y las instrucciones de los programas; (4) los dispositivos de entrada y de salida; y (5) los buses de datos, de instrucciones y de control, que permiten el flujo de información, de instrucciones o de señales de control a través de las partes del ordenador. Se llama CPU (Unidad Central de Proceso) al conjunto de la UC y la ALU con sus buses de comunicación necesarios. La CPU es un circuito integrado compuesto por miles de millones de componentes electrónicos integrados, y que se llama microprocesador.

En **memoria principal** se almacenan los datos a procesar o ya procesados y los resultados obtenidos. También se almacenan en ella los conjuntos de instrucciones, o programas, a ejecutar para el procesado de esos datos. Todo programa que se ejecute debe estar almacenado (cargado) en esta memoria principal. En la Figura 1.2., la memoria principal no forma parte de la CPU del ordenador; sin embargo, hay autores que sí la consideran parte de ella. Al margen de esa memoria principal, tanto la UC como la ALU disponen de registros propios de memoria y de bloques de memoria de acceso extremadamente rápido.

La memoria principal está construida mediante circuitos de electrónica digital que alcanzan dos estados estables posibles. Habitualmente se

llaman a estos estados el estado cero y el estado uno. Es lo que se conoce como BIT (BInary digiT: dígito binario). El **bit** es la unidad básica de información. Con un bit es posible decir verdadero o falso; sí o no.

Los circuitos electrónicos que forman la memoria son circuitos integrados, en los que se logra acumular una cantidad enorme de bits. Se podría hablar de la capacidad de memoria de una computadora, indicando el número de bits de que dispone. Pero habitualmente no se hace así, sino que se agrupan los bits para formar bloques de mayor capacidad de información. El agrupamiento más conocido de bits es el llamado BYTE. Un **byte** es una agrupación de 8 bits. Con 8 bits juntos y agrupados en una única unidad de información ya se pueden codificar muchos más que los dos valores posibles que alcanzábamos a codificar con un bit. Con un byte es posible obtener hasta $256=2^8$ combinaciones distintas de ceros y unos: 00000000; 00000001; 00000010; 00000011; …; 11111101; 11111110; 11111111.

Podemos decir, por tanto, que toda la memoria de la computadora está dividida en bloques (bytes). Estos bloques se disponen ordenadamente, de forma que se puede hacer referencia a uno u otro bloque concreto dentro del circuito integrado de memoria. Se puede hablar, por tanto, de posiciones dentro de la memoria; cada posición está formada por un byte. Así es posible crear un índice de posiciones de memoria. Cada posición tiene su índice, al que llamamos **dirección de memoria**. Así se logra que el ordenador tenga identificadas, de forma inequívoca, cada una de las posiciones de memoria: e identifica a cada una de ellas con su dirección.

Y si codificamos las direcciones de memoria con 32 bits, entonces podremos hacer referencia a 2^{32} bytes distintos o, lo que es lo mismo, a 4×2^{30} bytes distintos. Ya verá un poco más adelante que 2^{30} bytes es un Gigabyte. Así, entonces, un ordenador que codifique las direcciones de memoria con 32 bits podrá dar identidad a 4 Gigabytes: no a más. Si deseamos tener ordenadores con una memoria principal mayor de esos

4 Gigas, deberemos codificar las direcciones con más de 32 bits. Hace ya muchos años que existen supercomputadoras, o servidores, o estaciones de trabajo, que codifican sus direcciones con 64 bits; pero en los últimos años, buscando eliminar esa barrera de los 4 Gigabytes, esos tamaños de direcciones de memoria se han extendido también a los ordenadores personales. Ahora, con 64 bytes, la limitación en la cantidad de bytes identificables de forma inequívoca es de 16×2^{60} bytes, que es una cantidad enorme y quizá desorbitada: 16 Exabytes. Desde luego, los ordenadores PC que están hoy en el mercado no llevan tal descomunal cantidad de memoria: muchos han superado, sin embargo, los 4 Gigabytes de memoria RAM a la que están limitados los ordenadores con arquitectura de 32 bits.

La llamada **memoria masiva** es distinta cualitativamente de la memoria principal. Ya nos hemos referido a ella al presentarla como ejemplo de dispositivo de entrada y salida de información. También se la conoce o llama memoria auxiliar, o secundaria. Esta memoria no goza de la misma velocidad que la principal, pero sí logra ofrecer cantidades enormes de espacio donde almacenar datos de forma masiva. Ejemplos de esta memoria son el disco de una computadora, un DVD o un CD, o las cintas magnéticas.

La capacidad de almacenamiento de una computadora (o de cualquier soporte de información) se mide por el número de bytes de que dispone. De forma habitual se toman múltiplos de byte para esa cuantificación. Entendemos por…

Kilobyte: 2^{10} bytes, es decir, 1.024 bytes, que es un valor cercano a 10^3 bytes.

Megabyte: 2^{20} bytes, es decir 1.048.576 bytes, que es un del orden de los 10^6 bytes.

Gigabyte: 2^{30} bytes, es decir 1.073.741.824 bytes, que es un valor del orden de los 10^9 bytes.

Terabyte: 2^{40} bytes, es decir 1.099.511.627.776 bytes, que es un valor del orden de los 10^{12} bytes.

Petabyte: 2^{50} bytes, es decir 1.125.899.906.842.624 bytes, que es un valor del orden de los 10^{15} bytes.

Exabyte: 2^{60} bytes, es decir 1.152.921.504.606.846.976 bytes, que es un valor del orden de los 10^{18} bytes.

Otro componente, (cfr. Figura 1.2.) es la **Unidad de Control** (**UC**) de la computadora. Es la encargada de interpretar cada instrucción del programa (que está cargado en memoria), y de controlar su ejecución una vez interpretada. Capta también las señales de estado que proceden de las distintas unidades de la computadora y que le informan (a la UC) de la situación o condición actual de funcionamiento (v.gr., informan de si un determinado periférico está listo para ser usado, o de si un dato concreto está ya disponible). Y a partir de las instrucciones interpretadas y de las señales de estado recibidas, genera las señales de control que permitirán la ejecución de todo el programa.

Junto a la UC, vemos un segundo elemento que llamamos **ALU** (**Unidad Aritmético Lógica**). Este bloque de nuestro procesador está formado por una serie de circuitos electrónicos capaces de realizar una serie de operaciones: así, con esa electrónica digital, se definen una colección de operadores aritméticos y operadores lógicos que producen resultados en su salida a partir de la información almacenada en sus entradas. La ALU, como los demás elementos de la computadora, trabaja a las órdenes de las señales de control generadas en la UC.

Como ya ha quedado dicho, al conjunto de la UC y la ALU se le llama **CPU** (**Unidad Central de Proceso**). Ambos elementos han quedado agrupados por conveniencia de la tecnología: no formaban una unidad en la arquitectura Von Neumann inicial.

Existen algunos elementos más que han quedado recogidos en las Figuras 1.1 y 1.2. Uno es el conjunto de los llamados **controladores** de

entrada y salida, que permiten la correcta comunicación entre el procesador y los diferentes dispositivos de entrada y salida. El otro elemento es el formado por los **buses del sistema**, que comunican todas las unidades, y permiten el trasiego de datos (**bus de datos**), de direcciones de memoria (**bus de direcciones**) donde deben leerse o escribirse los datos, y de las diferentes sentencias de control generadas por la UC (**bus de control**).

Para terminar esta rápida presentación del procesador, conviene referir algunos elementos básicos que componen la UC, la ALU, y la memoria principal.

La Unidad de Control dispone, entre otros, de los siguientes registros de memoria de uso particular:

- **Registro de instrucción**, que contiene la instrucción que se está ejecutando.
- **Contador de programa**, que contiene permanentemente la dirección de memoria de la siguiente instrucción a ejecutar y la envía por el bus de direcciones.
- **Decodificador**, que se encarga de extraer el código de operación de la instrucción en curso.
- **Secuenciador**, que genera las micro-órdenes necesarias para ejecutar la instrucción decodificada.
- **Reloj**, que proporciona una sucesión de impulsos eléctricos que permiten sincronizar las operaciones de la computadora.

A su vez, la Unidad Aritmético Lógica, dispone de los siguientes registros de memoria y elementos:

- **Registros de Entrada**, que contienen los operandos (valores que intervienen como extremos de una operación) de la instrucción que se va a ejecutar.
- **Registro acumulador**, donde se almacenan los resultados de las operaciones.

- **Registro de estado**, que registra las condiciones de la operación anterior.
- **Circuito Operacional**, que realiza las operaciones con los datos de los registros de entrada y del registro de estado, deja el resultado en el registro acumulador y registra, para próximas operaciones, en el registro de estado, las condiciones que ha dejado la operación realizada.

Y finalmente, la memoria dispone también de una serie de elementos, que se recogen a continuación:

- **Registro de direcciones**, que contiene la dirección de la posición de memoria a la que se va a acceder.
- **Registro de intercambio**, que recibe los datos en las operaciones de lectura y almacena los datos en las operaciones de escritura.
- **Selector de memoria**, que se activa cada vez que hay que leer o escribir conectando la celda de memoria a la que hay que acceder con el registro de intercambio.
- **Señal de control**, que indica si una operación es de lectura o de escritura.

Instrucciones, Lenguajes, Compiladores.

Una **instrucción** es un conjunto de símbolos que representa (que codifica) una orden para el computador, que indica una operación o tratamiento sobre datos. Y un programa es un conjunto ordenado de instrucciones que se le dan a la computadora y que realizan, todas ellas, un determinado proceso.

Tanto las instrucciones, como los datos a manipular con esas instrucciones, se almacenan en la memoria principal, en lugares distintos de esa memoria.

Las instrucciones son solicitadas por la UC, y se envían desde la memoria hacia la Unidad de Control donde son interpretadas y donde se

generan las señales de control que gobiernan los restantes elementos del sistema.

Los datos pueden intervenir como operandos en un procedimiento (información inicial) o ser el resultado de una secuencia determinada de instrucciones (información calculada). En el primer caso, esos datos van de la memoria a la Unidad Aritmético Lógica, donde se realiza la operación indicada por la presente instrucción en ejecución. Si el dato es el resultado de un proceso, entonces el camino es el inverso, y los nuevos datos obtenidos son escritos en la memoria.

El tipo de instrucciones que puede interpretar o ejecutar la UC depende, entre otros factores de la ALU de que dispone el procesador en el que se trabaja. De forma general esas instrucciones se pueden clasificar en los siguientes tipos: de **transferencia** de información (por ejemplo, copiar un valor de una posición de la memoria a otra posición); **aritméticas**, que básicamente se reducen a la suma y la resta; **lógicas**, como operaciones relacionales (comparar valores de distintos datos) o funciones lógicas (AND, OR y XOR); **de salto**, con o sin comprobación o verificación de condición de salto, etc.

Las instrucciones a ejecutar deben ser codificadas de forma que la máquina las "entienda". Ya hemos visto que todo en una computadora se codifica con bits, a base de ceros y unos. Con ceros y unos se debe construir toda información o sentencia inteligible para la computadora. Lograr comunicarse con la máquina en ese lenguaje de instrucciones que ella entiende (código máquina) es una tarea difícil y compleja. A este galimatías hay que añadirle también el hecho de que los datos, evidentemente, también se codifican en binario, y que a éstos los encontramos a veces en zonas de memoria distintas a la de las instrucciones, pero en otras ocasiones se ubican intercalados con las instrucciones.

Un lenguaje así está sujeto a frecuentes errores de trascripción. Y resulta inexpresivo. Además, otro problema, no pequeño, de trabajar en

el lenguaje propio de una máquina es que un programa sólo resulta válido para esa máquina determinada, puesto que ante máquinas con distinta colección de microinstrucciones y diferente codificación de éstas, se tendrá lógicamente lenguajes diferentes. El único lenguaje que entiende directamente una máquina es su propio lenguaje máquina.

Resulta mucho más sencillo expresar las instrucciones que debe ejecutar la computadora en un lenguaje semejante al utilizado habitualmente por el hombre en su comunicación. Ésa es la finalidad de los lenguajes de programación. Pero un lenguaje así, desde luego, no lo entiende una máquina que sólo sabe codificar con ceros y unos.

Un lenguaje de programación no puede tener la complejidad de un lenguaje natural de comunicación entre personas. Un buen lenguaje de programación debe permitir describir de forma sencilla los diferentes datos y estructuras de datos. Debe lograr expresar, de forma sencilla y precisa, las distintas instrucciones que se deben ejecutar para resolver un determinado problema. Ha de resultar fácil escribir programas con él.

Así han surgido los distintos lenguajes de programación, capaces de expresar instrucciones en unas sentencias que quedan a mitad de camino entre el lenguaje habitual y el código máquina.

Dependiendo del grado de semejanza con el lenguaje natural, o de la cercanía con el lenguaje de la máquina, los lenguajes pueden clasificarse en distintas categorías:

1. El lenguaje de bajo nivel, o ensamblador, muy cercano y parecido al lenguaje máquina. Cada instrucción máquina se corresponde con una instrucción del lenguaje ensamblador, codificada, en lugar de con ceros y unos, con una agrupación de tres o cuatro letras que representan, abreviadamente, la palabra (habitualmente inglesa) que realiza la operación propia de esa instrucción.
2. Lenguajes de alto nivel, que disponen de instrucciones diferentes a las que la máquina es capaz de interpretar. Habitualmente, de una

instrucción del lenguaje de alto nivel se derivan varias del lenguaje máquina, y la ejecución de una instrucción de alto nivel supone la ejecución de muchas instrucciones en código máquina. Normalmente esas instrucciones se pueden expresar de una forma cómoda y comprensible.

Para resolver este problema de comunicación entre la máquina y su lenguaje máquina y el programador y su lenguaje de programación, se emplean programas que traducen del lenguaje de programación al lenguaje propio de la máquina. Ese programa traductor va tomando las instrucciones escritas en el lenguaje de programación y las va convirtiendo en instrucciones de código máquina.

Gracias a la capacidad de traducir un programa escrito en lenguaje de alto nivel al lenguaje máquina, se puede hablar de portabilidad en los lenguajes de alto nivel: la posibilidad de que un mismo programa pueda ser ejecutado en computadoras diferentes gracias a que cada una de ellas dispone de su correspondiente traductor desde el lenguaje en que va escrito el programa hacia el propio lenguaje máquina.

Se disponen de dos diferentes tipos de traductores. Unos, llamados **intérpretes**, van traduciendo el programa a medida que éste se ejecuta. Cada vez que un usuario quiera ejecutar ese programa deberá disponer del intérprete que vaya dictando a la computadora las instrucciones en código máquina que logran ejecutar las sentencias escritas en el lenguaje de alto nivel. Un intérprete hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el ordenador. No se crea un archivo o programa en código máquina. La ejecución del programa debe hacerse siempre supervisada por el intérprete.

Otro tipo de traductor se conoce como **compilador**: un compilador traduce todo el programa antes de ejecutarlo, y crea un programa en código máquina, que puede ser ejecutado tantas veces como se quiera, sin necesidad de disponer del código en el lenguaje de alto nivel y sin

necesidad tampoco de tener el compilador, que una vez ha creado el nuevo programa en lenguaje máquina ya no resulta necesario para su ejecución. Una vez traducido el programa al correspondiente lenguaje o código máquina, su ejecución es independiente del compilador.

Soporte físico (hardware) y soporte lógico (software). Sistemas Operativos.

Se habla de **hardware** cuando nos referimos a cualquiera de los componentes físicos de una computadora: la CPU, la memoria, un dispositivo de entrada… Se habla de **software** para referirse a los diferentes programas que hacen posible el uso de la computadora.

El hardware de una computadora se puede clasificar en función de su capacidad y potencia. Muy extendidos están las computadoras personales (comúnmente llamados PC). Habitualmente trabajaremos en ellos; cuando queramos referirnos a una computadora pensaremos en un PC, al que llamaremos, sencillamente, ordenador.

Un **sistema operativo** es un programa o software que actúa de interfaz o conexión entre el usuario de un ordenador y el propio hardware del ordenador. Ofrece al usuario el entorno necesario para la ejecución de los distintos programas. Un sistema operativo facilita el manejo del sistema informático, logrando un uso eficiente del hardware del ordenador. Facilita a los distintos usuarios la correcta ejecución de los programas, las operaciones de entrada y salida de datos, la gestión de la memoria, la detección de errores,… Permite una racional y correcta asignación de los recursos del sistema.

Piense en su ordenador. El hecho de que al pulsar un carácter del teclado aparezca una representación de ese carácter en la pantalla no es cosa trivial. Ni el que al mover su ratón se desplace de forma proporcionada una flecha o cursor en esa pantalla. ¿Quién se encarga de que lo que usted ha creado con un programa se guarde correctamente

en el disco duro: por ejemplo, un escrito creado con un editor de texto? ¿Quién gestiona los archivos en carpetas, y los busca cuando usted no recuerda dónde estaban? ¿Cómo logra el ordenador lanzar varios documentos a la impresora y que éstos salgan uno tras otro, de forma ordenada, sin colapsar el servicio ni sobrescribirse? ¿Por qué al hacer doble click en un icono se ejecuta un programa? ¿Cómo elimino de mi ordenador un archivo que ya no necesito?: no crea que basta con arrastrar ese archivo a lo que todo el mundo llamamos "la papelera": nada de todo eso es trivial. Todos requerimos de nuestro ordenador muchas operaciones, que alguien ha tenido que diseñar y dejar especificadas. Ésa es la tarea del sistema operativo.

Sistemas operativos conocidos son Unix, o su versión para PC llamada Linux, y el comercialmente extendido Windows, de Microsoft.

Recapitulación.

Hemos introducido algunas nociones necesarias para poder luego enfrentarnos al estudio de la programación y, en concreto, de un lenguaje como el C: qué es un lenguaje de programación y cómo se logra que un ordenador sea capaz de interpretar correctamente las instrucciones que el programador le indica con un determinado lenguaje.

Y hemos presentado los conceptos básicos necesarios para conocer cómo está construido y cómo funciona internamente un ordenador, y cómo es posible que realice una serie de operaciones perfectamente definidas y en correcta secuencia.

En el Epígrafe 1.1. del Capítulo 1 titulado "*Introducción al desarrollo de programas en Lenguaje C*" del manual "*Prácticas para aprender a programar en lenguaje C*", editado en Agosto de 2012 por el Centro Universitario de la Defensa (CUD) de San Javier (Murcia), puede encontrar otra redacción que presenta los mismos conceptos introducidos en este manual en este capítulo.

CAPÍTULO 2

CODIFICACIÓN NUMÉRICA.

El objetivo de este capítulo es mostrar el concepto de código, y específicamente y más en concreto presentar unas nociones básicas sobre la forma en que se codifican las cantidades mediante números.

Concepto de código.

Si se busca en el diccionario de la Real Academia Española el significado de la palabra **Código**, se encuentra, entre otras acepciones, las siguientes:

- "*Combinación de signos que tiene un determinado valor dentro de un sistema establecido. El código de una tarjeta de crédito.*"
- "*Sistema de signos y de reglas que permite formular y comprender un mensaje.*"

No encontramos con un segundo concepto que tiene, en el Diccionario

de la RAE, hasta 10 acepciones distintas. Es el concepto de **Signo** La primera de ellas dice:

- "*Signo: Objeto, fenómeno o acción material que, por naturaleza o convención, representa o sustituye a otro.*"

Podríamos decir que un código es una relación más o menos arbitraria que se define entre un conjunto de mensajes o significados a codificar y un sistema de signos que significan esos mensajes de forma inequívoca. El código no es tan solo el conjunto de signos, sino también la relación que asigna a cada uno de esos signos un significado concreto.

Ejemplos de códigos hay muchos: desde el semáforo que codifica tres posibles mensajes con sus tres valores de código diferentes (rojo, ámbar y verde) hasta el sistema de signos que, para comunicarse, emplean las personas sordas. O el código de banderas, o el sistema Braille para los invidentes que quieren leer.

Para establecer un código es necesario cuidar que se verifiquen las siguientes propiedades:

1. Que quede bien definido el conjunto de significados o mensajes que se quieren codificar. En el ejemplo del semáforo, queda claro que hay tres mensajes nítidos: adelante / alto / precaución. No hay confusión, ni posibilidad de equívoco en estos tres mensajes.
2. Que quede bien definido el conjunto de signos que van a codificar o significar esos mensajes. En el caso del semáforo queda claro este conjunto está formado por tres colores: rojo, ámbar y verde. No hay espacio para la confusión; excepto para quien tenga algún tipo de limitación con la vista.
3. Que quede meridianamente clara cuál es la relación entre cada signo y cada significado. En el caso del semáforo todo el mundo conoce que el signo color rojo significa el mensaje "alto"; que al ámbar le corresponde el mensaje "precaución"; y que al signo color verde le corresponde el mensaje "adelante".

4. Que no haya más significados que signos porque entonces ese código no es válido: tendrá mensajes que no están codificados, o tendrá signos que signifiquen varias cosas diferentes, lo que será causa de equívocos
5. También es deseable que no haya más signos que significados o mensajes a codificar. Un código con exceso de signos es válido, pero o tendrá redundancias (significados codificados con más de un signo) o tendrá signos que no signifiquen nada.

Lo mejor es siempre que un código se formule mediante una **aplicación biyectiva**, que establezca una relación entre significados y signos, que asigne a cada significado un signo y sólo uno, y que todo signo signifique un significado y sólo uno.

Los números como sistema de codificación de cantidades.

Para significar cantidades se han ideado muchos sistemas de representación, o códigos.

Todos conocemos el sistema romano, que codifica cantidades mediante letras. Ese sistema logra asignar a cada cantidad una única combinación de letras. Pero, por desgracia, no es un sistema que ayude en las tareas algebraicas. ¿Quién es capaz de resolver con agilidad la suma siguiente: CMXLVI + DCCLXIX?

El sistema de numeración arábigo, o indio, es en el que nosotros estamos habituados a trabajar. Gracias a él codificamos cantidades. Decir CMXLVI es lo mismo que decir 946; o decir DCCLXIX es lo mismo que decir 769. Son los mismos significados o cantidades codificados según dos códigos diferentes.

Un **sistema de numeración** es un código que permite codificar cantidades mediante números. Las cantidades se codifican de una

manera u otra en función del sistema de numeración elegido. Un número codifica una cantidad u otra en función del sistema de numeración que se haya seleccionado.

Un sistema de numeración está formado por un conjunto finito de símbolos y una serie de reglas de generación que permiten construir todos los números válidos en el sistema. Con un sistema de numeración (conjunto finito de símbolos y de reglas) se puede codificar una cantidad infinita de números.

Y hemos introducido ahora un nuevo concepto: el de símbolo. Esta vez el diccionario de la RAE no ayuda mucho. Nos quedamos con que un símbolo, en el ámbito del álgebra (**símbolo algebraico**, podemos llamarlo), es un signo: una letra que significa una cantidad con respecto a la unidad.

Un número es un elemento de un código: del código creado mediante un sistema de numeración. ¿Qué codifica un número?: Un número codifica una cantidad.

En la línea inmediatamente anterior hemos dibujado una serie de puntos negros.

Cuántos son esos puntos es una cuestión que en nada depende del sistema de numeración. La cantidad es la que es. Al margen de códigos.

En nuestro sistema habitual de codificación numérica (llamado sistema en base 10 o sistema decimal) diremos que tenemos 7 puntos. Pero si trabajamos en el sistema binario de numeración, diremos que tenemos 111 puntos. Lo importante es que tanto la codificación 7 (en base diez) como la codificación 111 (en base dos) significan la misma cantidad.

Y es que trabajar en base 10 no es la única manera de codificar cantidades. Ni tampoco es necesariamente la mejor.

Fundamentos matemáticos para un sistema de numeración. Bases, dígitos y cifras.

Todo número viene expresado dentro de un sistema de numeración. Todo sistema de numeración tiene un conjunto finito de símbolos. Este conjunto se llama base del sistema de numeración.

Una **base** es un conjunto finito y ordenado de símbolos algebraicos.

$B=\{\alpha_i \,/\, \alpha_0 = 0 \,;\, \alpha_{i+1} = \alpha_i + 1,\ \forall\ i=0, 1, \ldots, B-1\}$.

Sus propiedades pueden resumirse en las tres siguientes:

- El primer elemento de la base es el **cero** (este dígito es imprescindible en los sistemas de numeración posicionales: concepto que veremos más adelante en este capítulo).
- El segundo elemento de la base es la **unidad**.
- Los sucesivos elementos ordenados de la base son tales que cualquier elemento es igual a su inmediato anterior más la unidad.
- El máximo valor de la base es igual al cardinal de la base menos uno. Esta propiedad, en realidad, es consecuencia inmediata de la otras tres.

Se deduce que estas propiedades exigidas que toda base debe tener, al menos, dos elementos: el cero y la unidad.

La base B=10, por ejemplo, está formada por los siguientes elementos:

B= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Como ya hemos dicho, en los sistemas de numeración, además del conjunto de símbolos existe una colección de reglas que permiten, con ese conjunto finito de símbolos, codificar una cantidad infinita de números. Según una de estas reglas, todo número entero a > 0 puede ser escrito de modo único, para cada base B, en la forma

$$a = a_k \cdot B^k + a_{k-1} \cdot B^{k-1} + a_{k-2} \cdot B^{k-2} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0. \qquad (2.1.)$$

Donde k > 0 y cada uno de los a_i son elementos de la base y que, por tanto, verifican la siguiente relación:

$0 \leq a_i \leq B - 1$, para $i = 1, 2, 3, \ldots, k$, y $a_k \neq 0$ (2.2.)

A los elementos a_i se les llama, dentro de cualquier número, los **dígitos del número** a. Como se ve, los dígitos de cada número son siempre elementos o símbolos de la base de numeración A la Expresión 2.1. se la llama **expansión del número**.

El número habitualmente se representa como

$a = \left(a_k\, a_{k-1}\, a_{k-2} \ldots a_1\, a_0\right)_B$. (2.3.)

Cualquier número viene representado, en una base determinada, por una serie única de coeficientes a_i (ver 2.3.). A cada serie de coeficientes, que codifica un número de modo único en una determinada base, se le llama **cifra**. Número y cifra son conceptos equivalentes.

En una cifra importa tanto la posición relativa de cada dígito dentro de ella, como el valor de cada uno de esos dígitos. A este tipo de sistemas de numeración, en los que importa la posición del dígito dentro de la cifra, se llaman **sistemas de numeración posicionales**. No es lo mismo el número 567 que el 675, aunque ambos empleen la misma cantidad de y los mismos dígitos.

Cuanto más larga pueda ser la serie de dígitos que se emplean para codificar, mayor será el rango de números que podrán ser representados. Por ejemplo, en base B = 10, si disponemos de tres dígitos podremos codificar 1000 valores diferentes (desde el 000 hasta el 999); si disponemos de cinco dígitos podremos codificar 100.000 valores (desde el 00000 hasta el 99999). Desde luego, en un sistema de numeración como el que conocemos y usamos nosotros normalmente, no existen límites en la cantidad de dígitos, y con esta sencilla regla de la expansión se pueden codificar infinitas cantidades enteras.

Como se sabe, y como se desprende de esta forma de codificación, todo cero a la izquierda de estos dígitos supone un nuevo dígito que no aporta valor alguno a la cantidad codificada.

La expansión del número recoge el valor de cada dígito y su peso dentro de la cifra. El dígito a_0 del número a puede tomar cualquier valor comprendido entre 0 y B – 1. Cuando se necesita codificar un número mayor o igual que el cardinal de la base (B) se requiere un segundo dígito a_1, que también puede tomar sucesivamente todos los valores comprendidos entre 0 y B – 1. Cada vez que el dígito a_0 debiera superar el valor B – 1 vuelve a tomar el valor inicial 0 y se incrementa en uno el dígito a_1. Cuando el dígito a_1 necesita superar el valor B – 1 se hace necesario introducir un tercer dígito a_2 en la cifra, que también podrá tomar sucesivamente todos los valores comprendidos entre 0 y B – 1 incrementándose en uno cada vez que el dígito a_1 debiera superar el valor B – 1. El dígito a_1 "contabiliza" el número de veces que a_0 alcanza en sus incrementos el valor superior a B – 1. El dígito a_2 "contabiliza" el número de veces que a_1 alcanza en sus incrementos el valor superior a B – 1. Por tanto, el incremento en uno del dígito a_1 supone B incrementos del dígito a_0. El incremento en uno del dígito a_2 supone B incrementos del dígito a_1, lo que a su vez supone B^2 incrementos de a_0 Y así, sucesivamente, el incremento del dígito a_j exige B^j incrementos en a_0.

Todos los dígitos posteriores a la posición j codifican el número de veces que el dígito j ha recorrido de forma completa todos los valores comprendidos entre 0 y B – 1.

De todo lo dicho ahora se deduce que toda base es, en su sistema de numeración, base 10: Dos, en base binaria se codifica como 10; tres, en base 3, se codifica como 10; cuatro, en base 4, se codifica como 10… Es curioso: toda la vida hemos dicho que trabajamos ordinariamente en base 10. Pero… ¿qué significa realmente base 10?

Sistemas de numeración posicionales y bases más habituales en el mundo de la informática.

El sistema de numeración más habitual en nuestro mundo es el sistema decimal. Si buscamos un porqué a nuestra base 10 quizá deduzcamos que su motivo descansa en el número de dedos de nuestras dos manos.

Pero un ordenador no tiene manos. Ni dedos.

Como hemos visto en el capítulo anterior, el circuito electrónico básico de la memoria de los ordenadores, tal como hoy se conciben, está formado por una gran cantidad de circuitos electrónicos que tiene dos estados estables posibles.

¿Cuántos estados posibles?...: DOS.

Por eso, porque los ordenadores "sólo tienen dos dedos", es por lo que ellos trabajan mejor en base dos. Es decir, sólo disponen de dos elementos para codificar cantidades. El primer elemento, por definición de base, es el valor cero. El segundo (y último) es igual al cardinal de la base menos uno y es igual al primer elemento más uno. Esa base está formada, por tanto, por dos elementos que llamaremos: B = {0, 1}

Otras bases muy utilizadas en programación son la base octal (o base ocho) y la base hexadecimal (o base dieciséis).

Lo de la base hexadecimal puede llevar a una inicial confusión porque no nos imaginamos qué dígitos podemos emplear más allá del dígito nueve. Para esa base se extiende el conjunto de dígitos haciendo uso del abecedario: B= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}.

Sistema Binario.

Aprender a trabajar en una base nueva no está exento de cierta dificultad. Habría que echar mano de nuestra memoria, de cuando éramos infantes y no sabíamos contar. No nos resultaba sencillo saber qué número viene (en base diez) después del noventa y nueve.

Noventa y ocho,… Noventa y nueve,… Noventa y diez.

¡No!: cien.

Trabajemos en base diez:

```
 0   1   2   3   4   5   6   7   8   9
10  11  12  13  14  15  16  17  18  19
20  21  22  23  24  25  26  27  28  29
```

y… ¿en base dos?: después de cero el uno. Y después del uno… ¡el diez!

```
    0     1    10    11   100   101   110   111  1000  1001
 1010  1011  1100  1101  1110  1111 10000 10001 10010 10011
10100 10101 10110 10111 11000 11001 11010 11011 11100 11101
```

En ambos cuadros están codificadas las mismas cantidades. En base diez el primero, en base dos o base binaria el segundo.

Además de contar, es necesario aprender las operaciones matemáticas básicas. Al menos sumar y restar. De nuevo hay que volver a la infancia y aprender la aritmética básica de los clásicos cuadernos de sumas. ¿Se acuerda de los famosos cuadernos Rubio: http://www.rubio.net/?

Las reglas básicas para esas dos operaciones (suma y resta) son:

```
0 + 0 = 0                0 – 0 = 0
0 + 1 = 1                0 – 1 = 1 “y debo 1”
1 + 0 = 1                1 – 0 = 1
1 + 1 = 0 “y llevo 1”    1 – 1 = 0
```

Y así, se puede practicar con sumas de enteros de más o menos dígitos:

```
 10100     11101    1011011011     1010000110     100001001
 +1110    +10111     +1100011    +1100001110    +11101001
------    ------    ----------    ----------    ----------
100010    110100    1100111110    10110010100    111110010
```

Para las restas haremos lo mismo que cuando restamos en base diez: el minuendo siempre mayor que el sustrayendo: en caso contrario se invierten los valores y al resultado le cambiamos el signo:

```
 10100     11101    1011011011     1100001110     100001001
 -1110    -10111     -1100011    -1010000110    -11101001
------    ------    ----------    ----------    ----------
 00110     00110    1001111000     0010001000     000100000
```

El mejor modo de aprender es tomar papel y bolígrafo y plantearse

ejercicios hasta adquirir soltura y destreza suficiente para sentirse seguro en el manejo de estos números del en un sistema de numeración binario.

Al final del capítulo se recoge una sugerencia útil para practicar las operaciones aritméticas: realizarlas en la calculadora de Windows y probar luego a realizar esas mismas operaciones a mano.

Cambio de Base.

Paso de base dos a base diez: Para este cambio de base es suficiente con desarrollar la expansión del número. Por ejemplo, $(10011101)_2 = 1\cdot2^7+0\cdot2^6+0\cdot2^5+1\cdot2^4+1\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0=128+16+8+4+1=(157)_{10}$.

Paso de base diez a base dos: Para este cambio se divide el entero por dos (división entera), y se repite sucesivamente esta división hasta llegar a un cociente menor que la base. Simplemente vamos dividiendo por la base el número original y vamos repitiendo el procedimiento para los cocientes que vamos obteniendo.

Los restos de estas divisiones, y el último cociente, son los dígitos buscados (siempre serán valores entre 0 y B – 1). El último cociente es el dígito más significativo, y el primer resto el menos significativo. Por ejemplo, en el cálculo recogido en la Figura 2.1. vemos que el valor 157 expresado en base diez es, en base dos, 10011101.

Las bases octal y hexadecimal, a las que antes hemos hecho referencia, facilita el manejo de las cifras codificadas en base dos, que enseguida acumulan gran cantidad de dígitos, todos ellos ceros o unos. Para pasar de base dos a base dieciséis es suficiente con separar la cifra binaria en bloques de cuatro en cuatro dígitos, comenzando por el dígito menos significativo. Al último bloque, si no tiene cuatro dígitos, se le añaden tantos ceros a la izquierda como sean necesarios.

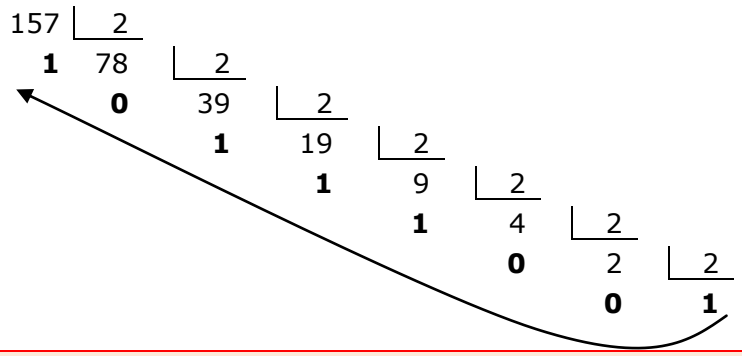


Figura 2.1. De base diez a base dos: $(157)_{10} = (10011101)_2$

La equivalencia entre la base dos y la base dieciséis (Pueden verse en la Tabla 2.1.) es inmediata sabiendo que dieciséis es dos a la cuarta.

Por ejemplo, la cantidad 10011101 expresada en base dos, queda, en base diez, 157 y, en base hexadecimal, 9D: los cuatro últimos dígitos binarios son 1101 que equivale al dígito D hexadecimal. Y los otros cuatro dígitos binarios son 1001, que equivalen al 9 hexadecimal.

No es necesario, para pasar de decimal a hexadecimal, hacer el paso intermedio por la base binaria. **Para pasar de cualquier base a la base decimal basta con la expansión del número** (Expresión 2.1.).

binario	decimal	hexadec.	binario	decimal	hexadec.
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Tabla 2.1. Equivalencias binario – hexadecimal.

Por ejemplo, el número 4E8, expresado en base hexadecimal, sería, en base diez, el siguiente:

$(4E8)_{16}=4 \cdot 16^2+ 14 \cdot 16^1+ 8 \cdot 16^0=4{\cdot}64+14{\cdot}16+8{\cdot}1= (1256)_{10}$.

donde, como se ve, se cambian los valores de los dígitos mayores que nueve por su equivalente decimal.

El cambio a la base octal es muy semejante a todo lo que se ha visto para el cambio a la base hexadecimal. De la Tabla 2.1. basta tener en consideración la columna de la izquierda. Los dígitos de la base octal son los mismos que para la base decimal, excluidos el 8 y el 9.

Quizá sea más complicado (quizá necesitamos una base de referencia) pasar de una base cualquiera a otra sin pasar por la base decimal.

Complementos a la Base.

Vamos a introducir dos conceptos nuevos, muy sencillos: los de complemento a la base y complemento a la base menos uno.

Supongamos el número N expresado en una base B determinada. Y supongamos que ese número tiene k dígitos. Llamamos **Complemento a la base** de ese número expresado en esa base determinada, a la cantidad que le falta para llegar a la cifra de (k + 1) dígitos, en el que el dígito más significativo es el uno y los demás son todos ellos iguales a cero.

De una forma más precisa, definiremos el complemento a la base de un número N codificado con k cifras en base B como

$$C_B^k\,(N)= B^k - N. \qquad (2.4.)$$

Por ejemplo, en base diez, el complemento a la base del número $(279)_{10}$ es la cantidad que hace falta para llegar a $(1000)_{10}$, que es $(721)_{10}$.

En esta definición hay que destacar que, si el número N viene expresado de forma que a su izquierda se tienen algunos dígitos iguales a cero,

entonces el complemento a la base es diferente que si estuviese sin esos dígitos, aunque la cantidad codificada sería la misma. Siguiendo con el ejemplo anterior, el complemento a la base del número codificado como $(0279)_{10}$ ya no es $(721)_{10}$, sino $(9721)_{10}$, porque ahora no se trata de calcular lo que falta para llegar a $(1000)_{10}$, sino para llegar a $(10000)_{10}$, puesto que ahora la cantidad numérica está codificado con cuatro dígitos.

El concepto de **Complemento a la Base menos uno** es muy semejante: es la cantidad que dista entre el número N, codificado con k dígitos en la base B, y el número formado por k dígitos, todos ellos con el valor del mayor elemento de la base B en la que se trabaja.

De una forma más precisa, definiremos el complemento a la base menos uno de un número N codificado con k cifras en base B como

$$C^{k}_{B-1}(N) = B^{k} - N - 1. \qquad (2.5.)$$

Por ejemplo el complemento a la base menos uno del número $(541)_{10}$ expresado en base diez es la cantidad que hace falta para llegar a $(999)_{10}$, que es $(458)_{10}$.

Igual que antes, cambia el complemento a la base menos uno según el número de ceros a su izquierda con que se codifique el número.

Es inmediato también deducir que la relación entre los dos complementos es que la diferencia entre ambos es igual a uno. De hecho se puede definir el Complemento a la Base menos uno de un número N codificado con k dígitos en la base B, como el Complemento a la Base de un número N codificado con k dígitos en la base B, menos 1.

$$C^{k}_{B-1}(N) = C^{k}_{B}(N) - 1. \qquad (2.6.)$$

Una curiosidad de los complementos es que, en cierta medida, facilitan el cálculo de las restas. Se pueden ver algunos ejemplos en base diez. Se cumple que la resta de dos enteros se puede también calcular

haciendo la suma entre el minuendo y el complemento a la base del sustrayendo, despreciando el posible acarreo final. Por ejemplo:

619 — 492 127	El complemento a la base de 492 es 508	619 + 508 1127

Donde si, como se ha dicho, despreciamos el último acarreo, tenemos que se llega al mismo resultado: 127.

También se puede realizar un procedimiento similar si se trabaja con el complemento menos uno. En ese caso, lo que se hace con el último acarreo, si se tiene, es eliminarlo del resultado intermedio y sumarlo para llegar al resultado final. Por ejemplo:

619 — 492 127	El complemento a la base menos uno de 492 es 507 Sumamos el acarreo a la cantidad obtenida sin acarreo	619 + 507 1126 +1 127

Hasta ahora todo lo expuesto puede aparecer como un enredo algo inútil porque, de hecho, para el cálculo del complemento a la base es necesario realizar ya una resta, y no parece que sea mejor hacer la resta mediante uno de los complementos que hacerla directamente.

Pero las cosas cambian cuando se trabaja en base dos. Vea en la Tabla 2.2. algunos ejemplos de cálculo de los complementos en esa base.

Si se compara la codificación de cada número N con su correspondiente complemento a la base menos uno, se descubre que todos los dígitos están cambiados: allí donde en N corresponde el dígito 1, en $C_1(N)$ se tiene un 0; y viceversa.

Es decir, que calcular el complemento a la base menos uno de cualquier número codificado en base dos es tan sencillo como cambiar el valor de cada uno de los dígitos de la cifra.

La ventaja clara que aportan los complementos es que no es necesario incorporar un restador en la ALU, puesto que con el sumador y un inversor se pueden realizar restas.

N	$C_2(N)$	$C_1(N)$
10110	01010	01001
11001111001	00110000111	00110000110
101	011	010

Tabla 2.2. Complemento a la base y a la base menos 1 en binario.

Recapitulación.

En este capítulo hemos visto que las cantidades pueden ser expresadas con distintos sistemas de codificación numérico. Y hemos estudiado las equivalencias entre los números codificados en diferentes bases.

Es conveniente que los conceptos introducidos en este capítulo estén bien trabajados. Es conveniente practicar diferentes cambios de base, y acostumbrarse a operar en base dos y en base hexadecimal. También es conveniente practicar en la búsqueda de complementos: de hecho, como ya veremos en el próximo capítulo, el ordenador echa mano de los complementos a la base para codificar y almacenar la información.

Ejercicios.

2.1. Cambio de base: Expresar $(810)_{10}$ en hexadecimal y otros cambios.

Podemos emplear dos caminos: o pasarlo a base 2 (por divisiones sucesivas por 2) y obtener a partir de ahí la expresión hexadecimal; o hacer el cambio a hexadecimal de forma directa, mediante divisiones sucesivas por 16. Mostramos esa última vía. El resultado es $(810)_{10} = (32A)_{16}$: el dígito más significativo, el último cociente (aquel

que ya es menor que el divisor, que es la base); y luego, uno detrás de otro, todos los restos, desde el último hasta el primero. El valor 10 se codifica, en hexadecimal, como A (cfr. Tabla 2.1.).

```
810 | 16
 10   50 | 16
       2    3
```

Podemos expresar ese número en cualquier otra base: por ejemplo, en base octal $(810)_{10} = (1452)_8$. Por último, como tercer ejemplo, lo pasamos a base 5: $(810)_{10} = (11220)_5$. Se puede verificar prontamente calculando la expansión (expresión 2.1.) del número en base 5, para pasarlo a la base decimal: $(11220)_5 = 1\cdot5^4+1\cdot5^3+2\cdot5^2+2\cdot5^1+0\cdot5^0 = 625+125+50+10 = (810)_{10}$

Las divisiones realizadas para estas dos conversiones son las siguientes:

```
810 |  8
  2   101 |  8
        5    12 |  8
             4     1
```

```
810 |  5
  0   162 |  5
        2    32 |  5
             2     6 |  5
                   1     1
```

2.2. ***Ensaye la resta de dos números expresados en base binaria. Primero puede realizar la resta en la forma habitual (minuendo menos sustrayendo) y luego repetirla sumando al minuendo el complemento a la base del sustrayendo.***

La Tabla 2.3. recoge algunos ejemplos. Intente obtener esos resultados.

2.3. ***Calcular los complementos a la base de los números de la Tabla 2.4., expresados en base 10.***

N_1	N_2	$C_2(N_2)$	$N_1 - N_2$
111 1011	100 1011	011 0101	11 0000
101 1101 1100	10 1110 1110	1 0001 0010	10 1110 1110
1 0100 1101	1101 1110	10 0010	110 1111
11 0001 0101	10 1000 1110	1 0111 0010	1000 0111

Tabla 2.3. Ejemplos de resta realizadas con complementos.

Número	Complemento a la base	Complemento a la base menos 1
0193	9807	9806
00710	99290	99289
6481	3519	3518
009999	990001	990000
98	2	1

Tabla 2.4. Ejemplos de complementos a la base en base 10.

Sugerencia

Un modo cómodo y rápido de lograr muchos ejemplos de cambio de base y de codificación de enteros es utilizar la calculadora de Windows. Si en el menú Ver se selecciona la calculadora Científica, se obtiene una como la mostrada en la Figura 2.2.

Debajo del visor, a su izquierda, se puede seleccionar la base en que se ha de trabajar. Si se introduce un valor en base decimal (Opción "Dec"), luego se puede ver esa misma cantidad codificada en hexadecimal, octal ó binario (opciones "Hex", "Oct" o "Bin"). Dependiendo de la opción, se activan los botones necesarios para trabajar con la base elegida.

Cuando se trabaja en base hexadecimal, octal o binaria, en la parte derecha y debajo del visor (cfr. Figura 2.3.), aparece un selector de cuatro posibilidades: "*Byte*", "*Word*", "*DWord*" ó "*QWord*" según se quiera trabajar con enteros de 8, 16, 32 ó 64 bits (1, 2, 4, u 8 bytes).

Figuras 2.2. Calculadora de Windows.

Al trabajar en binario se puede seleccionar, en el menú Ver, la opción "Número de dígitos en grupo". Así, los dígitos vendrán agrupados de cuatro en cuatro, mostrando con facilidad la equivalencia entre esa expresión binaria y la misma utilizando dígitos hexadecimales.

Figura 2.3. $(2103)_{16}$, o $(8451)_{10}$, aquí en base binaria.

CAPÍTULO 3

CODIFICACIÓN INTERNA DE LA INFORMACIÓN.

El objetivo de este capítulo es mostrar algunas formas habituales en que un ordenador codifica la información. Es conveniente conocer esta codificación: cómo fluye la información de los periféricos hacia el procesador o al revés; y cómo codifica el procesador la información en sus registros, o en la memoria principal, o en los circuitos de la ALU. Y resulta además de utilidad en las tareas del programador, que puede obtener muchas ventajas en el proceso de la información si conoce el modo en que esa información se encuentra disponible en las entrañas del ordenador.

Introducción.

La información, en un ordenador, se almacena mediante datos codificados con ceros y unos. Ya lo hemos visto en los capítulos

precedentes. Ahora, en este capítulo, queremos ver cómo son esos códigos de ceros y unos. No nos vamos a entretener en la codificación de todos los posibles tipos de dato. Hemos centrado principalmente la presentación de este capítulo en el modo cómo se codifican los enteros. Y eso por dos motivos: porque es un código muy sencillo; y porque resulta de gran utilidad conocer esa codificación: como veremos, el lenguaje C ofrece herramientas para poder manipular ese código y obtener, si se sabe, resultados interesantes y ventajosos.

Al tratar de los datos a codificar, deberemos distinguir entre la codificación que se emplea para la entrada y salida de datos, y la que el ordenador usa para su almacenamiento en memoria. Por ejemplo, si el usuario desea introducir el valor 412, deberá pulsar primero el cuatro, posteriormente la tecla del uno, y finalmente la del dos. El modo en que el teclado codifica e informa a la CPU de la introducción de cada uno de estos tres caracteres será diferente al modo en que finalmente el ordenador guardará en memoria el valor numérico 412.

Así las cosas, el modo en que un usuario puede suministrar información por teclado a la máquina es mediante caracteres, uno detrás de otro. Estos caracteres podemos clasificarlos en distintos grupos:

1. **De Texto**:
    a. **Alfanuméricos**:
        i. **Alfabéticos**: de la ‘a’ a la ‘z’ y de la ‘A’ a la ‘Z’.
        ii. **Numéricos**: del ‘0’ al ‘9’.
    b. **Especiales**: por ejemplo, ‘(‘, ‘)’. ‘+’. ‘?’. ‘@’,…
2. **De control**: por ejemplo, fin de línea, tabulador, avance de página, etc.
3. **Gráficos**: por ejemplo, ‘☺’, ‘♥’, ‘♣’,…

Como el ordenador sólo dispone, para codificar la información, de ceros y de unos, deberemos establecer una correspondencia definida entre el

conjunto de todos los caracteres y un conjunto formado por todas las posibles secuencias de ceros y de unos de una determinada longitud. A esa correspondencia la llamamos código de Entrada/Salida. Existen muchos distintos **códigos de E/S**, algunos de ellos normalizados y reconocidos en la comunidad internacional. Desde luego, cualquiera de estas codificaciones de E/S son arbitrarias, asignando a cada carácter codificado, una secuencia de bits, sin ninguna lógica intrínseca, aunque con lógica en su conjunto, como veremos. Estos códigos requieren de la existencia de tablas de equivalencia uno a uno, entre el carácter codificado y el código asignado para ese carácter.

Y, como acabamos de decir, esta codificación es distinta de la que, una vez introducido el dato, empleará el ordenador para codificar y almacenar en su memoria el valor introducido. Especialmente, si ese valor es un valor numérico. En ese caso especialmente, tiene poco sentido almacenar la información como una cadena de caracteres, todos ellos numéricos, y resulta mucho más conveniente, de cara también a posibles operaciones aritméticas, almacenar ese valor con una codificación numérica binaria. En ese caso, estamos hablando de la **representación o codificación interna** de los números, donde ya no se sigue un criterio arbitrario o aleatorio, sino que se toman en consideración reglas basadas en los sistemas de numeración posicional en base dos.

Códigos de Entrada/Salida.

Ya hemos quedado que esta codificación es arbitraria, asignando a cada carácter del teclado un valor numérico que queda codificado en las entrañas del ordenador mediante una cifra en base binaria de una longitud determinada de bits.

La cantidad de bits necesaria para codificar todos los caracteres dependerá, lógicamente, del número de caracteres que se deseen

codificar. Por ejemplo, con un bit, tan solo se pueden codificar dos caracteres: a uno le correspondería el código 0 y al otro el código 1. No tenemos más valores de código posibles y, por tanto, no podemos codificar un conjunto mayor de caracteres. Con dos bits, podríamos codificar cuatro caracteres; tantos como combinaciones posibles hay con esos dos dígitos binarios: 00, 01, 10 y 11.

En general diremos que con *n* bits seremos capaces de codificar hasta un total de 2^n caracteres. Y, al revés, si necesitamos codificar un conjunto de β caracteres, necesitaremos una secuencia de bits de longitud $n \geq \log_2 \beta$. Habitualmente, para un código de representación de caracteres se tomará el menor n que verifique esta desigualdad.

Una vez decidido el cardinal del conjunto de caracteres que se desea codificar, y tomado por tanto como **longitud del código** el menor número de bits necesarios para lograr asignar un valor de código a cada carácter, el resto del trabajo de creación del código será asignar a cada carácter codificado un valor numérico binario codificado con tantos ceros o unos como indique la longitud del código; evidentemente, como en cualquier código, deberemos asignar a cada carácter un valor numérico diferente.

Desde luego, se hace necesario lograr universalizar los códigos, y que el mayor números de máquinas y dispositivos trabajen con la misma codificación, para lograr un mínimo de entendimiento entre ellas y entre máquinas y periféricos. Para eso surgen los códigos normalizados de ámbito internacional. De entre los diferentes códigos normalizados válidos, señalamos aquí el **Código ASCII** (*American Standard Code for Information Interchange*). Es un código ampliamente utilizado. Está definido para una longitud de código n = 7 (es decir, puede codificar hasta 128 caracteres distintos), aunque existe una versión del ASCII de longitud n = 8 que dobla el número de caracteres que se pueden codificar (hasta 256) y que ha permitido introducir un gran número de caracteres gráficos.

En el código ASCII el carácter 'A' tiene el valor decimal 65 (en hexadecimal 41; 0100 0001 en binario), y consecutivamente, hasta el carácter 'Z' (valor decimal 90 en hexadecimal 5A; 0101 1010 en binario), van ordenados alfabéticamente, todas las letras mayúsculas. El alfabeto en minúsculas comienza un poco más adelante, con el código decimal 97 (61 en hexadecimal; 0110 0001 en binario) para la 'a' minúscula. Las letras 'ñ' y 'Ñ' tienen su código fuera de esta secuencia ordenada. Esta circunstancia trae no pocos problemas en la programación de aplicaciones de ordenación o de manejo de texto. Los caracteres numéricos comienzan con el valor decimal 48 (30 en hexadecimal) para el carácter '0', y luego, consecutivos, están codificados los restantes nueve guarismos de la base diez.

Es fácil encontrar una tabla con los valores del código ASCII. Quizá puede usted hacer la sencillo tarea de buscar esa tabla y comparar las codificaciones de las letras mayúsculas y minúsculas. ¿Advierte alguna relación entre ellas? ¿En qué se diferencian las mayúsculas y sus correspondientes minúsculas?

Representación o Codificación Interna de la Información.

La codificación de Entrada/Salida no es útil para realizar operaciones aritméticas. En ese caso resulta mucho más conveniente que los valores numéricos queden almacenados en su valor codificado en base dos.

Por ejemplo, utilizando el código ASCII, el valor numérico 491 queda codificado como 0110100 0111001 0110001. (34 39 31, en hexadecimal: puede buscarlos en una tabla ASCII) Ese mismo valor, almacenado con 16 bits, en su valor numérico, toma el código 0000000111101011 (desarrolle la expansión de este número, pasándolo a base 10, si quiere comprobarlo). No resulta mejor código únicamente porque requiere menos dígitos (se adquiere mayor compactación), sino

también porque esa codificación tiene una significación inmediatamente relacionada con el valor codificado. Y porque un valor así codificado es más fácilmente operable desde la ALU que si lo tomamos como una secuencia de caracteres de código arbitrario. Es decir, se logra una mayor adecuación con la aritmética.

Vamos a ver aquí la forma en que un ordenador codifica los valores numéricos enteros, con signo o sin signo. Desde luego, existe también una definición y normativa para la codificación de valores numéricos con decimales, también llamados de coma flotante (por ejemplo, la normativa IEEE 754), pero no nos vamos a detener en ella.

Enteros sin signo.

Para un entero sin signo, tomamos como código el valor de ese entero expresado en base binaria. Sin más.

Por ejemplo, para codificar el valor numérico N = 175 con ocho bits tomamos el código 10101111.

Además de saber cómo se codifica el entero, será necesario conocer el rango de valores codificables. Y eso estará en función del número de bits que se emplearán para la codificación.

Si tomáramos un byte para codificar valores enteros sin signo, entonces podríamos codificar hasta un total de 256 valores. Tal y como se ha definido el código en este epígrafe, es inmediato ver que los valores codificados son los comprendidos entre el 0 (código 00000000) y el 255 (código 11111111), ambos incluidos.

Si tomáramos dos bytes para codificar (16 bits), el rango de valores codificados iría desde el valor 0 hasta el valor 65.535 (en hexadecimal FFFF). Y si tomáramos cuatro bytes (32 bits) el valor máximo posible a codificar sería, en hexadecimal, el FFFFFFFF que, en base 10 es el número 4.294.967.295.

Evidentemente, cuantos más bytes se empleen, mayor cantidad de enteros se podrán codificar y más alto será el valor numérico codificado. En general, el rango de enteros sin signo codificados con n bits será el comprendido entre 0 y $+2^n - 1$.

Enteros con signo.

Hay diversas formas de codificar un valor numérico con signo. Vamos a ver aquí una de ellas, la que se emplea en los PC's de uso habitual.

El modo de codificar enteros con signo es el siguiente:

1. El bit más significativo no es un dígito numérico, sino el signo. Se pone a 0 si el entero codificado es positivo; se pone a 1 si el entero codificado es negativo.
2. Los restantes bits se emplean para la codificación del valor numérico.

La cuestión es cómo se codifica ese valor numérico (valor absoluto, sin consideración del signo, que ha quedado codificado en el bit más significativo) del entero que queremos codificar. Otra cuestión será determinar el rango de valores que se puede codificar cuando hablamos de enteros con signo.

Respecto al rango de valores la solución más cómoda y evidente es codificar los una cantidad de valores parejo entre negativos y positivos. No puede ser de otra manera si hemos decidido separar positivos y negativos por el valor del bit más significativo. Así, pues, para un entero de n bits, los valores que se pueden codificar son desde -2^{n-1} hasta $+2^{n-1} - 1$. Por ejemplo, para un entero de 1 bits, los valores posibles a codificar van desde -32.768 (-2^{15}) hasta 32.767 $(2^{15} - 1)$. Alguien puede extrañarse de que el cardinal de los positivos codificados es uno menos que el cardinal de los negativos; pero es que el cero es un valor que también hay que codificar.

Y así, con un byte se codifican los enteros comprendidos entre –128 y +127. Con dos bytes se pueden codificar los enteros comprendidos entre –32.768 y +32.767. Y con cuatro bytes el rango de valores codificados va desde el –2.147.483.648 hasta el +2.147.483.647.

En general, el rango de valores codificados con n bits será el comprendido entre -2^{n-1} y $+2^{n-1}-1$.

Más compleja es la decisión sobre cómo codificar esos valores. Ya ha quedado claro que el bit más significativo llevará la información del signo. Queda decidir cómo codificamos el valor absoluto del entero en los restantes n – 1 bits.

Una posibilidad sería codificar, sin más, el valor absoluto del número, en su código binario. Eso sería muy cómodo, pero trae un inconveniente no pequeño: suponiendo un entero de 16 bits, el código hexadecimal 0x8000 sería interpretado como –0 (menos cero), y el código 0x0000 como +0 (más cero). Así pues, el cero, tendría dos códigos posibles (ya sabe que el cero no tiene signo). Es, pues, una solución con un severo problema.

En realidad, la forma adoptada para codificar esos enteros es aparentemente más compleja, pero, como irá advirtiendo, llena de ventajas para el ordenador. El criterio de codificación es el siguiente:

- Si el entero es positivo, se codifica en esos n bits ese valor numérico en base binaria. Si, por ejemplo, el valor es el cero, y tenemos n = 16, entonces el código hexadecimal será 0000. El valor máximo positivo a codificar será 7FFF, que es el numero +32.767. Como puede comprobar, todos esos valores positivos en el rango marcado, requieren tan sólo n – 1 bits para ser codificados, y el bit más significativo queda siempre a cero.
- Si el entero es negativo, entonces lo que se codifica en esos bits es el complemento a la base del valor absoluto del valor codificado. Si, por ejemplo, tenemos n = 16, entonces el código del valor –1 será

FFFF; y el código hexadecimal del valor –32.768 (entero mínimo a codificar: el más negativo) será 8000.

Deberá aprender a buscar esos códigos para los valores numéricos; y deberá aprender a interpretar qué valor codifica cada código binario de representación.

Si, por ejemplo, queremos saber cómo queda codificado, con un byte, el valor numérico –75, debemos hacer los siguientes cálculos: El bit más significativo será 1, porque el entero a codificar es negativo. El código binario del valor absoluto del número es 1001011 (siete dígitos, que son los que nos quedan disponibles). El complemento a la base menos uno de ese valor es 0110100 (se calcula invirtiendo todos los dígitos: de 0 a 1 y de 1 a 0), y el complemento a la base será entonces 0110101 (recuérdese la igualdad 2.6.). Por tanto la representación interna de ese valor será 10110101, que en base hexadecimal queda B5.

Si hubiésemos codificado el entero con dos bytes entonces el resultado final del código sería FFB5.

Recapitulación.

En este capítulo hemos visto cómo se codifica la información dentro del ordenador. Hemos conocido primero cómo se trasfiere la información desde los periféricos hacia la CPU (código de E/S). Y cómo se codifica la información una vez ésta ya ha sido introducida en el ordenador: cómo se almacena en la memoria principal o en los registros internos del ordenador; especialmente nos hemos centrado en la codificación de los enteros.

En el epígrafe siguiente se muestran muchos valores negativos codificados tal y como los almacena el ordenador. Después de terminar el estudio de este capítulo es conveniente practicar y obtener la codificación interna de diferentes valores tomados de forma aleatoria.

Ejercicios.

El mejor modo para llegar a manejar la técnica de codificación de los enteros es la práctica. Queda recogida, en la Tabla 3.1. en este último epígrafe, la codificación de diferentes valores numéricos, para que se pueda practicar y verificar los resultados obtenidos. Todos ellos son valores negativos, y todos ellos codificados con dos bytes (16 bits, 4 dígitos hexadecimales).

−128	FF80	−127	FF81	−126	FF82	−125	FF83
−124	FF84	−123	FF85	−122	FF86	−121	FF87
−120	FF88	−119	FF89	−118	FF8A	−117	FF8B
−116	FF8C	−115	FF8D	−114	FF8E	−113	FF8F
−112	FF90	−111	FF91	−110	FF92	−109	FF93
−108	FF94	−107	FF95	−106	FF96	−105	FF97
−104	FF98	−103	FF99	−102	FF9A	−101	FF9B
−100	FF9C	−99	FF9D	−98	FF9E	−97	FF9F
−96	FFA0	−95	FFA1	−94	FFA2	−93	FFA3
−92	FFA4	−91	FFA5	−90	FFA6	−89	FFA7
−88	FFA8	−87	FFA9	−86	FFAA	−85	FFAB
−84	FFAC	−83	FFAD	−82	FFAE	−81	FFAF
−80	FFB0	−79	FFB1	−78	FFB2	−77	FFB3
−76	FFB4	−75	FFB5	−74	FFB6	−73	FFB7
−72	FFB8	−71	FFB9	−70	FFBA	−69	FFBB
−68	FFBC	−67	FFBD	−66	FFBE	−65	FFBF
−64	FFC0	−63	FFC1	−62	FFC2	−61	FFC3
−60	FFC4	−59	FFC5	−58	FFC6	−57	FFC7
−56	FFC8	−55	FFC9	−54	FFCA	−53	FFCB
−52	FFCC	−51	FFCD	−50	FFCE	−49	FFCF
−48	FFD0	−47	FFD1	−46	FFD2	−45	FFD3
−44	FFD4	−43	FFD5	−42	FFD6	−41	FFD7
−40	FFD8	−39	FFD9	−38	FFDA	−37	FFDB
−36	FFDC	−35	FFDD	−34	FFDE	−33	FFDF
−32	FFE0	−31	FFE1	−30	FFE2	−29	FFE3
−28	FFE4	−27	FFE5	−26	FFE6	−25	FFE7

Tabla 3.1. Codificación, con 2 bytes, de los enteros entre −128 y +127.

-24	FFE8	-23	FFE9	-22	FFEA	-21	FFEB
-20	FFEC	-19	FFED	-18	FFEE	-17	FFEF
-16	FFF0	-15	FFF1	-14	FFF2	-13	FFF3
-12	FFF4	-11	FFF5	-10	FFF6	-9	FFF7
-8	FFF8	-7	FFF9	-6	FFFA	-5	FFFB
-4	FFFC	-3	FFFD	-2	FFFE	-1	FFFF

Tabla 3.1. (Cont.).

A modo de ejemplo, mostramos los pasos a seguir para llegar a la codificación interna, en hexadecimal, a partir del valor entero.

3.1. *Codificación, con 2 bytes, del valor entero* $(-47)_{10}$.

El bit más significativo estará a 1, porque el entero es negativo. Los restantes 15 bits codifican el valor absoluto del entero. Su valor en binario (con 15 dígitos binarios) es $(47)_{10} = (000\ 0000\ 0010\ 1111)_2$. Su complemento a la base menos uno: $C_1^{15}(000\ 0000\ 0010\ 1111) = 111\ 1111\ 1101\ 0000$ y su complemento a la base: $C_2^{15}(N) = C_1^{15}(N) + 1 = 111\ 1111\ 1101\ 0001$. El código del entero en la representación interna del ordenador será, al añadirle delante el bit del signo, 1111 1111 1101 0001, que en hexadecimal será FFD1.

3.2. *Codificación, con 2 bytes, del valor entero* $(-1)_{10}$

El bit más significativo estará a 1, porque el entero es negativo. Los restantes 15 bits codifican el valor absoluto del entero. Su valor en binario (con 15 dígitos binarios) es: $(1)_{10} = (000\ 0000\ 0000\ 0001)_2$. Su

complemento a la base menos uno: $C_1^{15}(000\ 0000\ 0000\ 0001) = 111\ 1111\ 1111\ 1110$ y su complemento a la base: $C_2^{15}(N) = C_1^{15}(N) + 1 = 111\ 1111\ 1111\ 1111$. El código del entero en la representación interna del ordenador será, en hexadecimal, al añadirle el bit del signo, FFFF.

3.3. Otras formulaciones que ayudan a practicar y así entender la codificación interna de los enteros:

- Indicar el valor numérico que queda codificado con 89D4 (codificación en 16 bits)
- Si un entero queda codificado con AB (suponiendo una codificación en 16 bits), indique cómo quedaría codificado su valor cambiado de signo.
- Supongamos una codificación en 16 bits. Indique cuáles de las siguientes codificaciones de enteros con signo corresponden a enteros positivos, y cuáles a enteros negativos: ABCD, 7FFF, A0, FFF0.
- Ordene los números codificados en la pregunta anterior de menor a mayor.

CAPÍTULO 4

LENGUAJE C.

Presentamos en este capítulo una primera vista de la programación en lenguaje C. El objetivo ahora es ofrecer una breve introducción del Lenguaje C: no se trata ahora de aprender a usarlo: para esa meta tenemos todos los capítulos posteriores a éste. Ahora se ofrece una breve descripción de la evolución del Lenguaje C. También se muestran los conceptos básicos de un entorno de programación. Se procederá a redactar, con el entorno que cada uno quiera, un primer programa en C, que nos servirá para conocer las partes principales de un programa.

La lectura de este capítulo se complementa y completa la lectura del Capítulo 1, "*Introducción al desarrollo de programas en lenguaje C*", el Manual de Prácticas de la asignatura, "*Prácticas para aprender a programar en lenguaje C*". Es casi preceptivo, al trabajar con el manual de prácticas, realizar todos los trabajos que en él se van indicando.

Introducción.

Los lenguajes de programación están especialmente diseñados para programar computadoras. Sus características fundamentales son:

1. Son **independientes de la arquitectura física del ordenador**. Los lenguajes están, además, normalizados, de forma que queda garantizada la portabilidad de los programas escritos en esos lenguajes: un programa escrito en una máquina puede utilizarse en otra máquina distinta.
2. Normalmente **un mandato** o **sentencia** en un lenguaje de alto nivel da lugar, al ser introducido, a **varias instrucciones** en lenguaje máquina.
3. Utilizan **notaciones cercanas a las habituales**, con sentencias y frases semejantes al lenguaje matemático o al lenguaje natural.

El lenguaje C se diseñó en 1969. El lenguaje, su sintaxis y su semántica, así como el primer compilador de C fueron diseñados y creados por **Dennis M. Ritchie** en los laboratorios Bell. Juntamente con **Brian Kernighan** escribieron, en 1978, el libro "*The C Programming Language*". Esta primera versión del lenguaje C, presentada en ese libro, es conocida como **K&R** (nombre tomado de las iniciales de sus dos autores). Más tarde, en 1983, se definió el primer estándar del Lenguaje: **ANSI C**, que es el estándar sobre el que este manual trabaja. Más adelante, en este mismo capítulo, se relacionan los sucesivos estándares aparecidos: el último con fecha de 2011. Cuando, a lo largo del manual, se hable del Lenguaje C nos referimos habitualmente al estándar ANSI C. Cuando no sea así, expresamente se indicará de qué estándar se está hablando.

El lenguaje ANSI C tiene muy pocas reglas sintácticas, sencillas de aprender. Su léxico es muy reducido: tan solo **32 palabras**.

A menudo se le llama **lenguaje de medio nivel**, más próximo al código máquina que muchos lenguajes de más alto nivel. Es un lenguaje

apreciado en la comunidad científica por su probada eficiencia. Es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para implementar aplicaciones. Permite el uso del lenguaje ensamblador en partes del código, trabaja a nivel de bit, y permite modificar los datos con operadores que manipulan bit a bit la información. También se puede acceder a las diferentes posiciones de memoria conociendo su dirección.

El lenguaje C es un lenguaje del paradigma imperativo, estructurado. Permite con facilidad la programación **modular**, creando unidades que pueden compilarse de forma independiente, que pueden posteriormente enlazarse. Así, se crean funciones o procedimientos que se pueden compilar y almacenar, creando bibliotecas de código ya editado y compilado que resuelve distintas operaciones. Cada programador puede diseñar sus propias bibliotecas, que simplifican luego considerablemente el trabajo futuro. El ANSI C posee una amplia colección de bibliotecas de funciones estándar y normalizadas.

Entorno de programación.

Para escribir el código de una aplicación en un determinado lenguaje, y poder luego compilar y obtener un programa que realice la tarea planteada, se dispone de lo que se denomina un entorno de programación.

Un **entorno de programación** es un conjunto de programas necesarios para construir, a su vez, otros programas. Un entorno de programación incluye **editores de texto**, **compiladores**, **archivos de biblioteca**, **enlazadores** y **depuradores** (para una explicación más detallada de los entornos de programación puede consultar el manual de prácticas de la asignatura, "*Prácticas para aprender a programar en lenguaje C*", en el Capítulo 1, "*Introducción al desarrollo de programas en lenguaje C*", en los epígrafes 1.2. y 1.3.). Gracias a Dios existen entornos de

programación integrados (genéricamente llamados **IDE**, acrónimo en inglés de *Integrated Development Environment*), de forma que en una sola aplicación quedan reunidos todos estos programas. Muchos de esos entornos pueden obtenerse a través de internet. Por ejemplo, el entorno de programación **Dev-C++** (disponible en múltiples enlaces), ó **CodeLite** (http://codelite.org/). En el desarrollo de las clases de la asignatura se usará unos de esos entornos de libre distribución. Para conocer el uso de una herramienta de programación concreta, lo mejor es consultar la documentación que, para cada una de ellas, suele haber disponible también de forma gratuita, a través de Internet. No se presentará en este manual ningún IDE concreto.

Un **editor** es un programa que permite construir ficheros de caracteres, que el programador introduce a través del teclado. Un programa no es más que archivo de texto. El programa editado en el lenguaje de programación se llama **fichero fuente**. Algunos de los editores facilitan el correcto empleo de un determinado lenguaje de programación, y advierten de inmediato la inserción de una palabra clave, o de la presencia de un error sintáctico, marcando el texto de distintas formas.

Un **compilador** es un programa que compila, es decir, genera **ficheros objeto** que "entiende" el ordenador. Un archivo objeto todavía no es una archivo ejecutable.

El entorno ofrece también al programador un conjunto de archivos para incluir o **archivos de cabecera**. Esos archivos suelen incluir abundantes parámetros que hacen referencia a diferentes características de la máquina sobre la que se está trabajando. Así, el mismo programa en lenguaje de alto nivel, compilado en máquinas diferentes, logra archivos ejecutables distintos. Es decir, el mismo código fuente es así portable y válido para máquinas diferentes.

Otros archivos son los **archivos de biblioteca**. Son programas previamente compilados que realizan funciones específicas. Suele suceder que determinados bloques de código se deben escribir en

diferentes programas. Ciertas partes que son ya conocidas porque son comunes a la mayor parte de los programas están ya escritas y vienen recogidas y agrupadas en archivos que llamamos **bibliotecas**. Ejemplos de estas funciones son muchas matemáticas (trigonométricas, o numéricas,…) o funciones de entrada de datos desde teclado o de salida de la información del programa por pantalla (cfr. Capítulo 8 de este manual; también cfr. Capítulo 2 del manual de Prácticas "*Prácticas para aprender a programar en lenguaje C*"). Desde luego, para hacer uso de una función predefinida, es necesario conocer su existencia y tener localizada la biblioteca donde está pre-compilada; eso es parte del aprendizaje de un lenguaje de programación, aunque también se disponen de grandes índices de funciones, de fácil acceso para su consulta.

Al compilar un programa generamos un archivo objeto. Habitualmente los programas que compilemos harán uso de algunas funciones de biblioteca; en ese caso, el archivo objeto no es aún un fichero ejecutable, puesto que le falta añadir el código de esas funciones. Un entorno de programación que tenga definidas bibliotecas necesitará también un **enlazador** o *linkador* (perdón por esa palabra tan horrible) que realice la tarea de "juntar" el archivo objeto con las bibliotecas empleadas y llegar, así, al **código ejecutable**.

La creación e implementación de un programa no suele terminar con este último paso descrito. Con frecuencia se encontrarán **errores**, bien **de compilación** porque haya algún error sintáctico; bien **de ejecución**, porque el programa no haga exactamente lo que se deseaba. No siempre es sencillo encontrar los errores de nuestros programas; un buen entorno de programación ofrece al programador algunas herramientas llamadas **depuradores**, que facilitan esta tarea.

Podríamos escribir el algoritmo que define la tarea de crear un programa. Ese algoritmo podría tener el aspecto del recogido en el flujograma de la Figura 4.1.

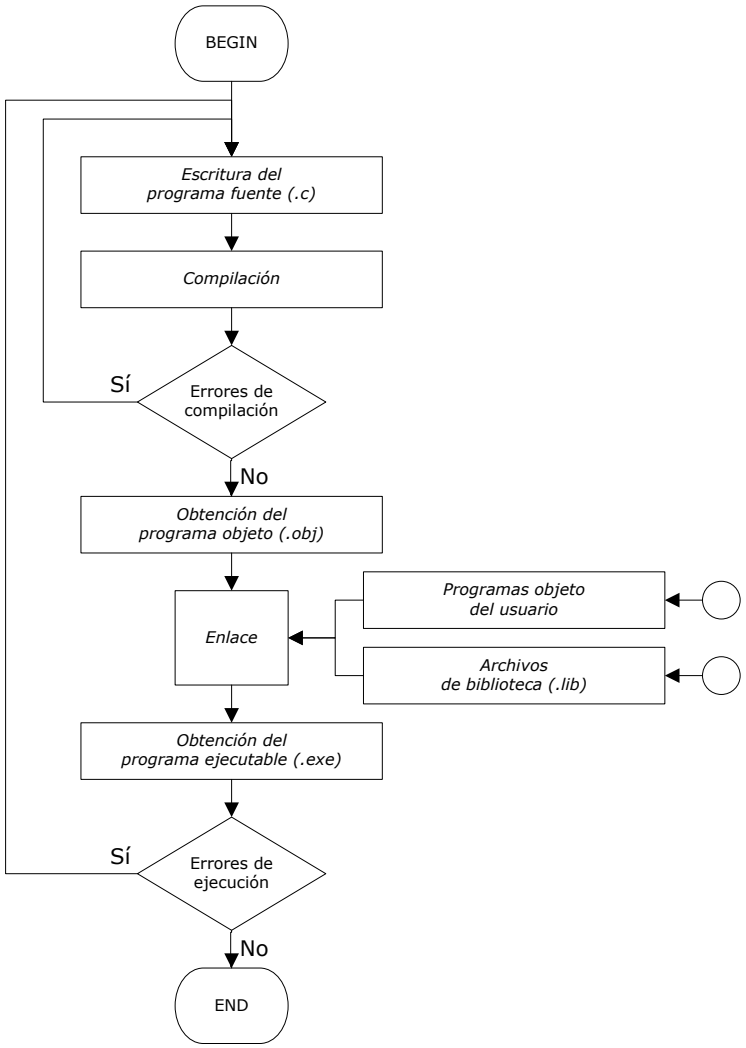


Figura 4.1. Fases en el desarrollo de un programa.

En el caso del lenguaje C, el archivo de texto donde se almacena el código tendrá un *nombre* (el que se quiera) y la extensión **.cpp** (si trabajamos con un entorno de programación de C++), o **.c**. Al compilar el fichero fuente (*nombre.cpp*) se llega al código máquina, con el mismo nombre que el archivo donde está el código fuente, y con la extensión **.obj**. Casi con toda probabilidad en código fuente hará uso de funciones que están ya definidas y pre-compiladas en las bibliotecas. Ese código

pre-compilado está en archivos con la extensión **.lib**. Con el archivo .obj y los necesarios .lib que se deseen emplear, se procede al "*linkado*" o enlazado que genera un fichero ejecutable con la extensión **.exe**.

Estructura básica de un programa en C.

Aquí viene escrito un sencillo programa en C (cfr. Código 4.1.). Quizá convenga ponerse ahora delante del ordenador y, con el editor de C en la pantalla, escribir estas líneas y ejecutarlas.

Código 4.1. Primer programa en C.

```
#include <stdio.h>
/* Este es un programa en C. */
// Imprime un mensaje en la pantalla del ordenador
int main(void)
{
      printf("mi primer programa en C.");
      return 0;
}
```

Todos los programas en C deben tener ciertos componentes fijos. Vamos a ver los que se han empleado en este primer programa:

1. `#include <stdio.h>`: Los archivos **.h** son los archivos de cabecera en C. Con esta línea de código se indica al compilador que se desea emplear, en el programa redactado, alguna función que está declarada en el archivo de biblioteca `stdio.h`. Esta archivo contiene las declaraciones de una colección de programas de entrada y salida por consola (pantalla y teclado).

   Esta instrucción nos permite utilizar cualquiera de las funciones declaradas en el archivo. Esta línea de código recoge el nombre del archivo `stdio.h`, donde están recogidos todos los prototipos de las

funciones de entrada y salida estándar. Todo archivo de cabecera contiene identificadores, constantes, variables globales, macros, prototipos de funciones, etc.

Toda línea que comience por # se llama **directiva de preprocesador**. A lo largo del libro se irán viendo diferentes directivas.

2. `main`: Es el nombre de una función. Es la **función principal** y establece el punto donde comienza la ejecución del programa. La función `main` es necesaria en cualquier programa de C que desee ejecutar instrucciones. **Un código será ejecutable si y sólo si dispone de la función `main`**.

3. **`int`** `main(`**`void`**`)`: Los paréntesis se encuentran siempre después de un identificador de función. Entre ellos se recogen los parámetros que se pasan a la función al ser llamada. En este caso, no se recoge ningún parámetro, y entre paréntesis se indica el tipo **`void`**. Ya se verá más adelante qué significa esta palabra. Delante del nombre de la función principal (`main`) también viene la palabra **`int`**, porque la función principal que hemos implementado devuelve un valor de tipo entero con signo: en concreto, en nuestro ejemplo, devuelve el valor 0 (instrucción **`return 0;`**)

4. `/* comentarios */`: Símbolos opcionales. Todo lo que se encuentre entre estos dos símbolos son comentarios al programa fuente y no serán leídos por el compilador.

   Los comentarios no se compilan, y por tanto no son parte del programa; pero son muy necesarios para lograr unos códigos inteligibles, fácilmente interpretables tiempo después de que hayan sido redactados y compilados. Es muy conveniente, cuando se realizan tareas de programación, insertar comentarios con frecuencia que vayan explicando el proceso que se está llevando en cada momento. Un programa bien documentado es un programa que

luego se podrá entender con facilidad y será, por tanto, más fácilmente modificado y mejorado.

También se pueden incluir comentarios precediéndolos de la doble barra `//`. En ese caso, el compilador no toma en consideración lo que esté escrito desde la doble barra hasta el final de la línea.

5. `;`: Toda sentencia en C termina con el **punto y coma**. En C, se entiende por sentencia todo lo que tenga, al final, un punto y coma. La línea antes comentada (`#include <stdio.h>`) no termina con un punto y coma porque no es una sentencia: es (ya lo hemos dicho) una directiva de preprocesador.
6. `{}`: Indican el principio y el final de todo bloque de programa. Cualquier conjunto de sentencias que se deseen agrupar, para formar entre ellas una sentencia compuesta o bloque, irán marcadas por un par de llaves: una antes de la primera sentencia a agrupar; la otra, de cierre, después de la última sentencia. Una función es un bloque de programa y debe, por tanto, llevarlas a su inicio y a su fin.
7. La sentencia **`return 0;`** Como acabamos de definir, la función main devuelve un valor de tipo **`int`**: por eso hemos escrito, delante del nombre de la función, esa palabra. La función main es tal que antes de terminar devolverá el valor 0 (así lo indica esta sentencia o instrucción). Aún es demasiado pronto para saber a quién le es "devuelto" ese valor. Por ahora hay que aprender a hacerlo así. La tarea de aprender a programar exige, en sus primeros pasos, saber fiarse de los manuales y de quien pueda enseñarnos. No se puede explicar todo el primer día.

Elementos léxicos.

Entendemos por **elemento léxico** cualquier palabra válida en el lenguaje C. Serán elementos léxicos, o palabras válidas, todas aquellas palabras que formen parte de las palabras reservadas del lenguaje, y

todas aquellas palabras que necesitemos generar para la redacción del programa, de acuerdo con una normativa sencilla.

Para crear un **identificador** (un identificador es un símbolo empleado para representar un objeto dentro de un programa) en el lenguaje C se usa cualquier secuencia de una o más **letras** (de la 'A' a la 'Z', y de la 'a' a la 'z', excluida las letras 'Ñ' y 'ñ'), **dígitos** (del '0' al '9') o **símbolo subrayado** ('_'). Los identificadores creados serán palabras válidas en nuestro programa en C. Con ellos podemos dar nombre a variables, constantes, tipos de dato, nombres de funciones o procedimientos, etc. También las palabras propias del lenguaje C son identificadores; estas palabras se llaman **palabras clave** o **palabras reservadas**.

Además de la restricción en el uso de caracteres válidos para crear identificadores, existen otras reglas básicas para su creación en el lenguaje C. Estas reglas básicas (algunas ya han quedado dichas, pero las repetimos para mostrar en este elenco todas las reglas juntas) son:

1. Están formadas por los caracteres de tipo alfabético (de la 'A' a la 'Z', y de la 'a' a la 'z'), caracteres de tipo dígito (del '0' al '9') y el signo subrayado (algunos lo llaman guión bajo: '_'). No se admite nuestra 'ñ' (ni la 'Ñ' mayúscula), y tampoco se aceptan aquellos caracteres acentuados, con diéresis, o con cualquier otra marca.
2. Debe comenzar por una letra del alfabeto o por el carácter subrayado. Un identificador no puede comenzar por un dígito.
3. El compilador sólo reconoce los primeros 32 caracteres de un identificador, pero éste puede tener cualquier otro tamaño mayor. Aunque no es nada habitual generar identificadores tan largos, si alguna vez así se hace hay que evitar que dos de ellos tengan iguales los 32 primeros caracteres, porque entonces para el compilador ambos identificadores serán el mismo.
4. Las letras de los identificadores pueden ser mayúsculas y minúsculas. El compilador distingue entre unas y otras, y dos

identificadores que se lean igual y que se diferencien únicamente en que una de sus letras es mayúscula en uno y minúscula en otro, son distintos.

5. Un identificador no puede deletrearse igual y tener el mismo tipo de letra (mayúscula o minúscula) que una palabra reservada o que una función definida en una librería que se haya incluido en el programa mediante una directiva `include`.

Las palabras reservadas, o palabras clave, son identificadores predefinidos que tienen un significado especial para el compilador de C. Sólo se pueden usar en la forma en que han sido definidos. En la Tabla 4.1. se muestra el conjunto de palabras clave o reservadas (que siempre van en minúscula) en ANSI C. Como puede comprobar, es un conjunto muy reducido: un total de 32 palabras.

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

Tabla 4.1. Palabras reservadas en C.

A lo largo del manual se verá el significado de cada una de ellas.

La palabra **goto** viene recogida entre paréntesis porque, aunque es una palabra reservada en C y su uso es sintácticamente correcto, de hecho no es una palabra permitida en un paradigma de programación estructurado como es el paradigma del lenguaje C. Esta palabra ha quedado como reliquia de las primeras versiones del C.

Sentencias simples y sentencias compuestas.

Una **sentencia simple** es cualquier expresión válida en la sintaxis de C que termine con el carácter de punto y coma. **Sentencia compuesta** es una sentencia formada por una o varias sentencias simples. Un punto y coma es una sentencia simple. Una sentencia compuesta está formada por una o varias simples (varios puntos y comas); se inicia con una llave de apertura ({) y se termina con una llave de clausura (}).

Errores y depuración.

No es extraño que, al terminar de redactar el código de un programa, al iniciar la compilación, el compilador deba abortar su proceso y avisar de que existen errores. El compilador ofrece algunos mensajes que clarifican frecuentemente el motivo del error, y la corrección de esos errores no comporta habitualmente demasiada dificultad. A esos errores sintácticos los llamamos **errores de compilación**. Ejemplo de estos errores pueden ser que haya olvidado el punto y coma de una sentencia, o que falte la llave de cierre de bloque de sentencias compuestas, o que sobre un paréntesis, o que emplee un identificador mal construido…

Otras veces, el compilador no halla error sintáctico alguno, y compila correctamente el programa, pero luego, en la ejecución, se producen errores que acaban por abortar el proceso. A esos errores los llamamos **errores de ejecución**. Un clásico ejemplo de este tipo de errores es forzar al ordenador a realizar una división por cero, o acceder a un espacio de memoria para el que no estamos autorizados. Esos errores también suelen ser sencillos de encontrar, aunque a veces, como no son debidos a fallos sintácticos ni de codificación del programa sino que pueden estar ocasionados por el valor que en un momento concreto adquiera una variable, no siempre son fácilmente identificables, y en esos casos puede ser necesario utilizar los depuradores que muchos entornos de programación ofrecen.

Y puede ocurrir también que el código no tenga errores sintácticos, y por tanto el compilador termine su tarea y genere un ejecutable; que el programa se ejecute sin contratiempo alguno, porque en ningún caso se llega a un error de ejecución; pero que el resultado final no sea el esperado. Todo está sintácticamente bien escrito, sin errores de compilación ni de ejecución, pero hay errores en el algoritmo que pretende resolver el problema que nos ocupa. Esos errores pueden ser ocasionados sencillamente por una errata a la hora de escribir el código, que no genera un error sintáctico, ni aborta la ejecución: por ejemplo, teclear indebidamente el operador suma (+) cuando el que correspondía era el operador resta (–). A veces, sin embargo, el gazapo no es fácil de encontrar. No hemos avanzado lo suficiente como para poner algún ejemplo. Cada uno descubrirá los suyos propios en su itinerario de aprendizaje del lenguaje: es cuestión de tiempo encontrarse con esas trampas.

Todo error requiere una modificación del programa, y una nueva compilación. No todos los errores aparecen de inmediato, y no es extraño que surjan después de muchas ejecuciones.

Evolución del lenguaje C. Historia de sus estándares.

La primera versión del Lenguaje C es la presentada por los autores del manual "*The C Programming Language*": Brian Kernighan y Dennis Ritchie. Este libro, considerado como la Biblia del C, aparecido en 1978, y esa primera versión del lenguaje es conocida como **K&R C**. Puede consultar en Internet la referencia a cualquiera de estos dos autores, especialmente del segundo, Dennis Ritchie, verdadero padre del lenguaje C, y galardonadísimo científico en el ámbito de la computación.

Años más tarde, en 1988, el Instituto Nacional Estadounidense de Estándares (*American National Standards Institute*, **ANSI**) estableció la

especificación estándar del lenguaje C. El nombre que se le dio a este estándar fue el ANSI X3.159-1989. También es conocido como **C89**, y también Standard C.

Un año más tarde, la Organización Internacional de Normalización (*International Organization for Standardization*, **ISO**) adoptó el estándar del ANSI. El nombre que se le dio a este nuevo estándar fue el ISO/IEC 9899:1990, y más comúnmente se le llama **C90**.

A efectos prácticos, ambos estándares, C89 y C90 establecen las mismas especificaciones: las modificaciones introducidas por C90 respecto a C89 son mínimas.

En el año 2000 se adoptó un tercer y nuevo estándar: el ISO/IEC 9899:1999. A este estándar se le conoce como **C99**. Casi todos los compiladores que se encuentran actualmente en el mercado compilan para C99, aunque todos ellos permiten hacer restricciones para que únicamente trabaje con las especificaciones del C89 ó del C90.

Entre otras muchas cosas, el lenguaje C99 introduce algunas nuevas palabras clave en el lenguaje. Estas palabras son: **`restrict`**, **`_Bool`**, **`_Complex`** e **`_Imaginary`**. Ninguna de ellas es soportada por C++, y, desde luego, tampoco por C90. Además, incluye la nueva palabra **`inline`**, que sí está soportada en C++.

El octubre del año 2011 se establece un nuevo estándar: el ISO/IEC 9899:2011, comúnmente conocido como **C11**. La versión Draft de este nuevo estándar, publicada en abril de 2011, se conoce como N1570, y está disponible de libre distribución en Internet. El documento definitivo en versión pdf se vende por 238 francos suizos (en agosto de 2012): inalcanzable ahora en tiempos de crisis. Los pocos comentarios que aparecen en el manual sobre esta última definición de estándar se basan, por tanto, en el documento Draft publicado en abril.

A lo largo de este manual se presenta, salvo algunas excepciones, el estándar C90. En frecuentes ocasiones quedarán también indicadas en

el manual, con epígrafe aparte, algunas nuevas aportaciones del estándar C99 y del C11, pero no siempre. Téngase en cuenta que el lenguaje Estándar C90 es casi un subconjunto perfecto del lenguaje orientado a objetos C++, y que bastantes de las novedades que incorpora el estándar C99 pueden también transportarse en un programa que se desee compilar con un compilador de C++; pero hay otras nuevas aportaciones del C99 que no están soportadas por el C++: A éstas les he dedicado menor atención en este manual, o simplemente no han quedado citadas. Respecto al estándar C11 hay que tener en cuenta que al ser reciente es posible que muchos compiladores disponibles en la red no hayan incluido sus nuevas incorporaciones. Quizá, en una manual de introducción, no sea necesario ahondar en sus novedades.

Recapitulación.

En este capítulo hemos introducido los conceptos básicos iniciales para poder comenzar a trabajar en la programación con el lenguaje C. Hemos presentado el entorno habitual de programación y hemos visto un primer programa en C (sencillo, desde luego) que nos ha permitido mostrar las partes básicas del código de un programa: las directivas de preprocesador, los comentarios, la función principal, las sentencias (simples o compuestas) y las llaves que agrupan sentencias. Y hemos aprendido las reglas básicas de creación de identificadores.

CAPÍTULO 5

ALGORITMIA.
DIAGRAMAS DE FLUJO.
PSEUDOCÓDIGO.

De lo que se va a tratar aquí es de intentar explicar cómo construir un programa que resuelva un problema concreto. No es tarea sencilla, y las técnicas que aquí se van a presentar requieren, por parte de quien quiera aprenderlas, empaparse de cierta lógica que se construye con muy pocos elementos y que no necesariamente resulta trivial. Se trata de aprender a expresar, ante un problema que se pretende resolver mediante un programa informático, y en una lógica extremadamente simple, cualquier colección o lista ordenada de instrucciones, fácil luego de traducir como un conjunto de sentencias que debe ejecutar un ordenador. Pero al decir que la lógica es simple, nos referimos a que se define mediante un conjunto mínimo de reglas: no quiere decir que sea sencilla de aprender o de utilizar.

En este capítulo se intenta presentar el concepto de algoritmo y se muestran algunas herramientas para expresar esos algoritmos, como los flujogramas o el pseudocódigo. El capítulo ofrece suficientes ejercicios para ayudar a afianzar los conceptos introducidos.

Es importante comprender y asimilar bien los contenidos de este capítulo: se trata de ofrecer las herramientas básicas para lograr expresar un procedimiento que pueda entender un ordenador; aprender cómo resolver un problema concreto mediante una secuencia ordenada y finita de instrucciones sencillas y precisas. Si ante un problema planteado logramos expresar el camino de la solución de esta forma, entonces la tarea de aprender un lenguaje de programación se convierte en sencilla y, hasta cierto punto, trivial. Una vez se sabe qué se ha de decir al ordenador, sólo resta la tarea de expresarlo en un lenguaje cualquiera.

Las principales referencias utilizadas para la confección de este capítulo han sido:

- "*El arte de programar ordenadores*". Volumen I: "*Algoritmos Fundamentales*" · Donald E. Knuth · Editorial Reverté, S.A., 1985.
- "*Introducción a la Informática*". 3ª Ed. · Alberto Prieto E., Antonio Lloris R., Juan Carlos Torres C. · Editorial Mc Graw Hill, 2006.
- "*Flowchart Techniques*": http://www.hostkhiladi.com/it-tutorials/DETAILED/flow-charting/flowcharting.pdf

Concepto de Algoritmo.

La noción de **algoritmo** es básica en la programación de ordenadores. El diccionario de la Real Academia Española lo define como "conjunto ordenado y finito de operaciones que permite hallar la solución de un problema". Otra definición podría ser: "procedimiento no ambiguo que resuelve un problema", entendiendo por procedimiento o proceso

(informático) una secuencia de instrucciones (o sentencias u operaciones) bien definida, donde cada una de ellas requiere una cantidad finita de memoria y se realiza en un tiempo finito.

Hay que tener en cuenta que la arquitectura de un ordenador permite la realización de un limitado conjunto de operaciones, todas ellas muy sencillas, tales como sumar, restar, transferir datos, etc. O expresamos los procedimientos en forma de instrucciones sencillas (es decir, no complejas) y simples (es decir, no compuestas), o no lograremos luego "indicarle" al ordenador (programarlo) qué órdenes debe ejecutar para alcanzar una solución.

No todos los procedimientos capaces (al menos teóricamente capaces) de alcanzar la solución de un problema son válidos para ser utilizados en un ordenador. Para que un procedimiento pueda ser luego convertido en un programa ejecutable por una computadora, debe verificar las siguientes propiedades:

1. Debe finalizar tras un **número finito de pasos**. Vale la pena remarcar la idea de que los pasos deben ser, efectivamente, "muy" finitos.
2. Cada uno de sus pasos debe definirse de un modo **preciso**. Las acciones a realizar han de estar especificadas en cada caso de forma rigurosa y sin ambigüedad.
3. Puede tener varias entradas de datos, o ninguna. Sin embargo, al menos debe tener **una salida o resultado**: el que se pretende obtener. Al hablar de "entradas" o de "salidas" nos referimos a la información (en forma de datos) que se le debe suministrar al algoritmo para su ejecución, y la información que finalmente ofrece como resultado del proceso definido.
4. **Cada una de las operaciones** a realizar debe ser lo bastante **básica** como para poder ser efectuada por una persona con papel y lápiz, de modo **exacto** en un lapso de **tiempo finito**.

Cuando un procedimiento **no ambiguo** que **resuelve** un determinado problema verifica además estas cuatro propiedades o condiciones, entonces diremos, efectivamente, que ese procedimiento es un algoritmo.

De acuerdo con Knuth nos quedamos con la siguiente definición de algoritmo: **una secuencia finita de instrucciones, reglas o pasos que describen de forma precisa las operaciones que un ordenador debe realizar para llevar a cabo una tarea en un tiempo finito**. Esa tarea que debe llevar a cabo el algoritmo es, precisamente, la obtención de la salida o el resultado que se indicaba en la tercera propiedad arriba enunciada.

El algoritmo que ha de seguirse para alcanzar un resultado buscando no es único. Habitualmente habrá muchos métodos o procedimientos distintos para alcanzar la solución buscada. Cuál de ellos sea mejor que otros dependerá de muchos factores. En la práctica no sólo queremos algoritmos: queremos *buenos* algoritmos. Un criterio de bondad frecuentemente utilizado es el tiempo que toma la ejecución de las instrucciones del algoritmo. Otro criterio es la cantidad de recursos (principalmente de memoria) que demanda la ejecución del algoritmo.

Creación y expresión de algoritmos.

Si el problema que se pretende afrontar es sencillo, entonces la construcción del algoritmo y del programa que conduce a la solución es, con frecuencia, sencilla también. Ante muchos problemas, un programador avezado simplemente se pondrá delante de la pantalla y teclado de su ordenador y se pondrá a escribir código; y con frecuencia el resultado final será bueno y eficiente.

Pero cuando el programador es novel este modo de proceder no es siempre posible ni recomendable. O cuando el problema a resolver tiene cierta complejidad (cosa por otro lado habitual en un programa que

pretenda resolver un problema medianamente complejo), esa actitud de sentarse delante de la pantalla y, sin más, ponerse a redactar código, difícilmente logra un final feliz: en realidad así se llega fácilmente a unos códigos ininteligibles e indescifrables, imposibles de reinterpretar. Y si ese código tiene —cosa por otro lado nada extraña— algún error, éste logra esconderse entre las líneas enmarañadas. Y su localización se convierte en un trabajo que llega a ser tedioso y normalmente y finalmente estéril.

Ante muchos programas a implementar es conveniente y necesario primero plantear un diseño de lo que se desea hacer. Gracias a Dios existen diferentes métodos estructurados que ofrecen una herramienta eficaz para esta labor de diseño. (Aquí el término "estructurado" no viene como de casualidad, ni es uno más posible entre tantos otros: es un término acuñado que quedará más completamente definido en el este Capítulo y en el siguiente. Por ahora basta con lo que generalmente se entiende por estructurado.)

Cuando se trabaja con método y de forma estructurada, se logran notables beneficios:

- La detección de errores se convierte en una tarea asequible.
- Los programas creados son fácilmente modificables.
- Es posible crear una documentación clara y completa que explique el proceso que se ha diseñado e implementado.
- Se logra un diseño modular que fracciona el problema total en secciones más pequeñas.

El uso habitual de métodos de diseño estructurado aumenta grandemente la probabilidad de llegar a una solución definitiva; y eso con un coste de tiempo significativamente pequeño. Además, con estas metodologías, aumenta notablemente la probabilidad de localizar prontamente los posibles errores de diseño. No hay que minusvalorar esa ventaja: el coste de un programa se mide en gran medida en horas

de programación. Y ante un mal diseño previo, las horas de búsqueda de errores llegan a ser tan impredecibles como, a veces, desorbitadas.

El hecho de que estas metodologías permitan la clara y completa documentación del trabajo implementado también es una ventaja de gran valor e importancia. Y eso permite la posterior comprensión del código, en un futuro en el que se vea necesario hacer modificaciones, o mejoras, o ampliaciones al código inicial.

Y la posibilidad que permite la programación estructurada de fragmentar los problemas en módulos más sencillos (ya lo verá…) facilita grandemente el trabajo de implementación en equipo, de dar por terminadas distintas fases de desarrollo aún antes de terminar el producto final. Así, es posible implicar a varios equipos de desarrollo en la implementación final del programa. Y se asegura que cada equipo pueda comprender su tarea en el conjunto y pueda comprender la solución final global que todos los equipos persiguen alcanzar.

Diagramas de flujo (o flujogramas).

Un diagrama de flujo, o flujograma, es una representación gráfica de una secuencia de operaciones en un programa. Recoge de forma ordenada, según una secuencia, la colección de instrucciones que el programa o subprograma deberá ejecutar para alcanzar su objetivo final. Es un modo habitual de expresar algoritmos.

El flujograma emplea diferentes figuras sencillas para significar diferentes tipos de instrucciones o para representar algunos elementos necesarios que ayudan a determinar qué instrucciones deben ser ejecutadas. Las instrucciones se representan mediante cajas que se conectan entre sí mediante líneas con flecha que indican así el flujo de instrucciones de la operación diseñada. Iremos viendo los diferentes elementos o figuras a continuación. Toda la lógica de la programación puede expresarse mediante estos diagramas. Una vez expresada la

lógica mediante uno de esos gráficos, siempre resulta sencillo expresar esa secuencia de instrucciones con un lenguaje de programación. Una vez terminado el programa, el flujograma permanece como la mejor de las documentaciones para futuras revisiones del código.

Símbolos utilizados en un flujograma.

Son varios los símbolos que se han definido para expresar mediante flujograma la secuencia de instrucciones de un programa. No los vamos a ver aquí todos, sino simplemente aquellos que usaremos en este manual para crear flujogramas acordes con la llamada programación estructurada. Pueden verse esos símbolos en la Figura 5.1.

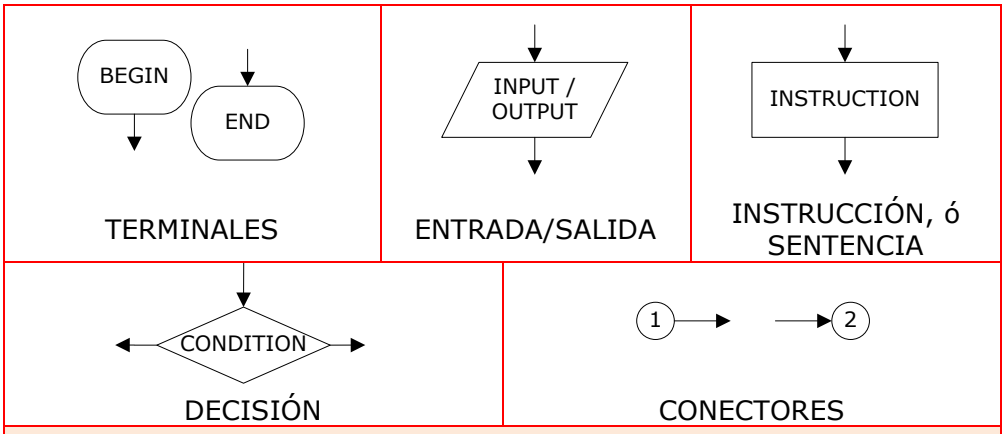


Figura 5.1. Símbolos básicos para los flujogramas

Estos elementos son los siguientes:

- **Terminales**: Se emplea para indicar el punto donde comienza y donde termina el flujograma. Cada flujograma debe tener uno y sólo un punto de arranque y uno y sólo un punto de término.
- **Entrada / Salida**: Indican aquellos puntos dentro de la secuencia de instrucciones donde se inserta una función de entrada o salida de

datos: entrada por teclado, o desde un archivo o desde internet; salida por pantalla, o hacia un archivo, etc.

- **Instrucción, Proceso o Sentencia**: Se emplea para las instrucciones aritméticas o binarias, e instrucciones de movimiento de datos. Cuáles son esas instrucciones lo iremos viendo a lo largo del capítulo mediante ejemplos, y luego a lo largo de todo el libro.
- **Decisión**: El diamante de decisión es pieza clave en la programación estructurada. Señala un punto donde el flujo se divide en dos caminos posibles no simultáneos: sólo uno de los dos será el que se tomará en el flujo de ejecución de instrucciones. La decisión viene expresada mediante una condición que construimos con operaciones de relación o lógicas y con valores literales o recogidos en variables. Esa expresión condicional sólo puede tener dos valores posibles: verdadero o falso. Más adelante verá ejemplos de estas expresiones.

  Para referirnos a un diamante de decisión, lo haremos indicando la condición que tiene inscrita. Diremos, pues, "el diamante C1", o "el diamante gobernado por la condición C1", indistintamente.
- **Conectores**: Con frecuencia un flujograma resulta demasiado extenso, y las líneas de ejecución se esparcen entre las páginas. Los conectores permiten extender el flujograma más allá de la página actual marcando cómo se conectan las líneas de ejecución de una página a otra.
- **Líneas de flujo**: Conectando todos los elementos anteriores, están las líneas de flujo. Sus cabezas de flecha indican el flujo de las operaciones, es decir, la exacta secuencia en que esas instrucciones deben ser ejecutadas. El flujo normal de un flujograma irá habitualmente de arriba hacia abajo, y de izquierda a derecha: las puntas de flecha son sólo necesarias cuando este criterio no se cumpla. De todas formas, es buena práctica dibujar siempre esas puntas de flecha, y así se hará en este manual.

Estructuras básicas de la programación estructurada.

Un flujograma estructurado será aquel que se ajuste perfectamente a una colección reducida de estructuras básicas que vamos a definir a continuación y que vienen recogidas en la Figura 5.2.

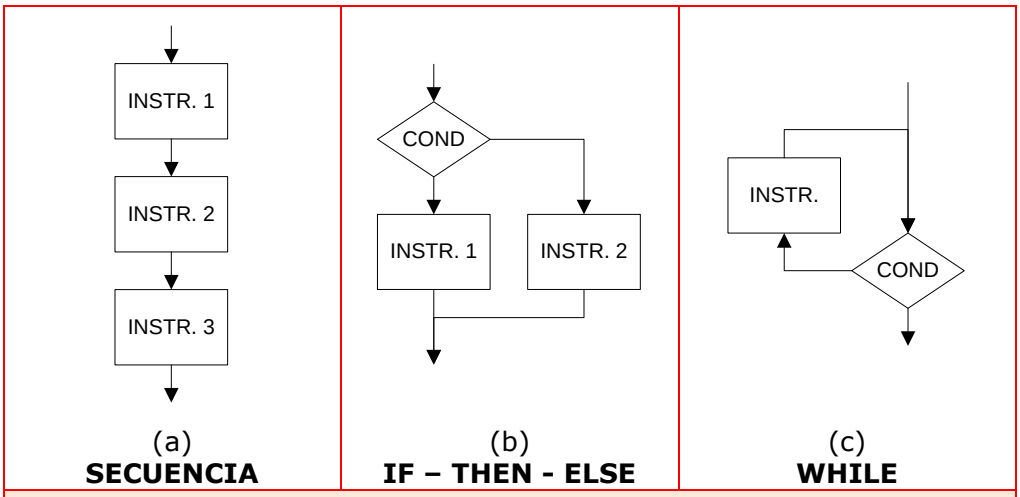


Figura 5.2. Estructuras básicas en un flujograma estructurado.

La **secuencia** está formada por una serie consecutiva de dos o más sentencias o instrucciones que se llevan a cabo una después de la otra y en el orden preciso en el que vienen indicadas por las líneas de flujo.

La estructura **condicional** o **de decisión** IF – THEN – ELSE separa dos caminos de ejecución posibles. Siempre se seguirá uno de ellos y nunca los dos. La decisión sobre qué camino tomar dependerá de la condición del diamante. Habitualmente pondremos a la izquierda el camino a recorrer si la condición se evalúa como verdadera; y el camino de la derecha si se evalúa como falsa. De todas formas, es conveniente indicar siempre con la leyenda "Sí", ó "V" (de Verdadero), ó "T" (de True) y "No" ó "F" (de Falso o False) ambos caminos.

La estructura de decisión tiene una versión simplificada, para el caso en que el camino del ELSE (cuando la condición es falsa) no recoge ninguna sentencia o instrucción a ejecutar. Habitualmente también llamaremos a las estructuras condicionales **estructuras de bifurcación**: serán **cerradas** si disponen de procesos en ambos caminos; será **abierta** si el camino del ELSE no dispone de sentencia alguna.

La estructura de **iteración** o **de repetición** WHILE permite la construcción de una estructura de salto y repetición dentro de un programa, creando lo que solemos llamar un bucle o una iteración. La decisión de ejecutar el proceso en el bucle se realiza antes de la primera ejecución del proceso: se evalúa la condición del diamante, y si esta es evaluada como verdadera entonces se ejecuta el proceso y se vuelve a evaluar la condición del diamante; si de nuevo la condición se evalúa como verdadera, entonces se vuelve a ejecutar el proceso y de nuevo se vuelve a evaluar la condición del diamante. La secuencia del programa quedará así atrapada en un ciclo de ejecución de una colección de sentencias, hasta que la condición del diamante llegue a ser falsa.

La programación estructurada permite la **sustitución** de cualquier bloque de instrucción por una nueva estructura básica de secuencia, de decisión o de iteración. Así, es posible expresar cualquier proceso de ejecución.

Estructuras derivadas.

Aunque cualquier programa estructurado puede expresarse mediante una combinación de cualquiera de las estructuras presentada en la Figura 5.2., con relativa frecuencia se emplean otras tres estructuras adicionales. Éstas se recogen en la Figura 5.3.

La estructura **DO–WHILE** (Figura 5.3.(a)) es parecida a la estructura WHILE, con la diferencia de que ahora primero se ejecuta la sentencia y sólo después se evalúa la condición de diamante que, si resulta

verdadera, permitirá que, de nuevo, se ejecute la sentencia iterada y de nuevo se volverá a evaluar la condición.

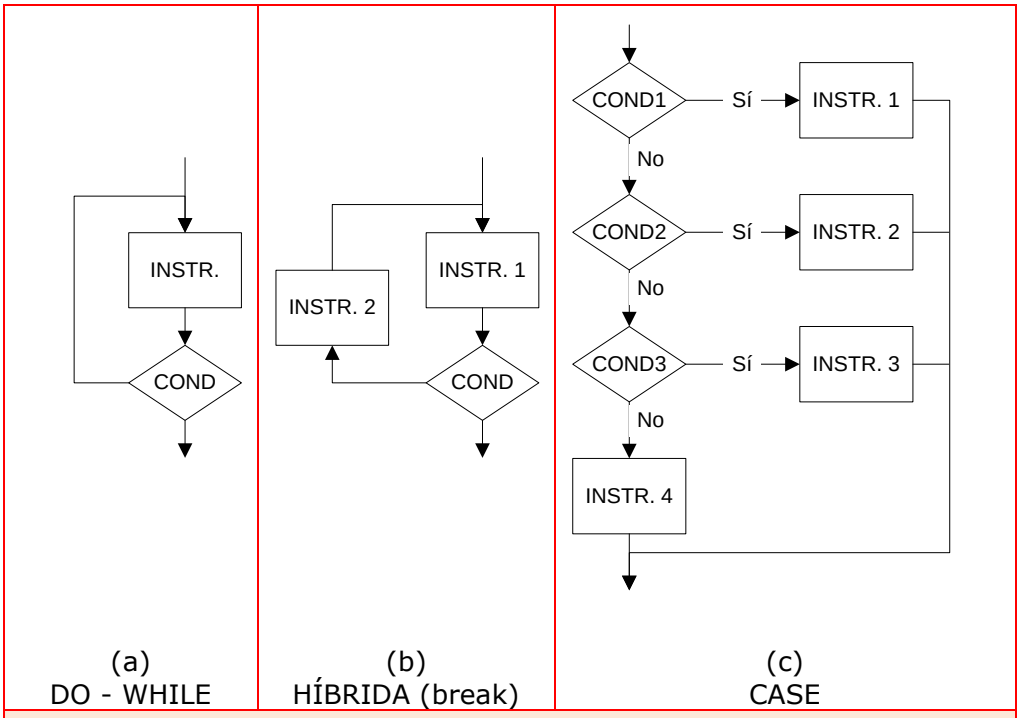
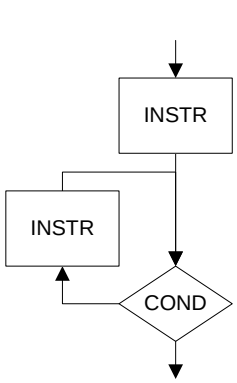


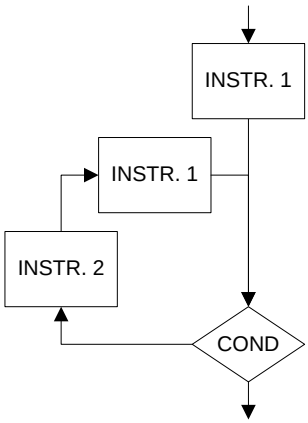
Figura 5.3. Estructuras derivadas básicas en un flujograma estructurado.

La estructura que hemos llamado HÍBRIDA (Figura 5.3.(b)) no está recogida en muchas referencias de programación estructurada. Pero la introducimos aquí porque, de hecho, el lenguaje C, como otros muchos lenguajes, tienen una sentencia válida y frecuentemente usada que permite escribir el código de esta estructura. Como se ve, es otra estructura de repetición, donde hay un proceso que se ejecuta antes de la evaluación de la sentencia (y en eso se comporta como una estructura DO-WHILE) y otro proceso que se ejecutará después de la evaluación de la condición de diamante si éste resulta verdadera (y en esto se comporta de forma semejante a la estructura WHILE).

Esta estructura híbrida no respeta, de hecho, las reglas básicas de la llamada programación estructurada. Pero bien utilizada sí permite simplificar el código estructurado y, de hecho —con prevención y sabiendo lo que se hace— se utiliza ordinariamente. De hecho, no es muy difícil comprobar que ambas estructuras derivadas de repetición se pueden expresar fácilmente con la concatenación de un bloque de proceso y una estructura WHILE (y por tanto de acuerdo con las reglas básicas de la programación estructurada).



(a)
Representación de la estructura derivada DO-WHILE mediante la secuencia de una sentencia y de una estructura WHILE.



(b)
Representación de la estructura HIBRIDA mediante la secuencia de una sentencia y de una estructura WHILE que itera dos sentencias.

Figura 5.4. Un modo de expresar las estructuras derivadas de repetición mediante la concatenación y la estructura WHILE.

Como se puede efectivamente ver en la Figura 5.4., estas nuevas estructuras no son estrictamente necesarias. Pero con frecuencia se acude a ellas porque permiten simplificar el código de un programa. Obsérvese que en ambos casos, al expresar la estructura derivada mediante la estructura básica WHILE se debe repetir uno de los procesos. Quizá no sea un problema si ese proceso a reescribir en el

código está formado por una sencilla sentencia simple. Pero también pudiera ser que ese proceso fuera una larga sucesión de sentencias y de estructuras de decisión y de repetición.

La tercera estructura derivada presentada en la Figura 5.3. es la estructura CASE. Ésta es útil cuando se necesita una estructura condicional donde la condición no sea evaluada simplemente como verdadero o falso, sino que admita una variada distribución de valores.

Por poner un ejemplo de los infinitos posibles, supóngase la luz de un semáforo, que puede tomar los colores Rojo, Naranja o Verde. Tomando el flujograma de la estructura CASE de la Figura 5.3., la primera condición sería "el semáforo está verde", y la primera sentencia sería "seguir adelante"; la segunda condición sería "el semáforo está naranja", y la segunda instrucción, "circule con cuidado"; la tercera condición sería "el semáforo está rojo" y la instrucción a ejecutar si esa condición fuera cierta sería "no avance". En el caso de que el valor del color del semáforo no fuera ninguno de esos tres, cabría pensar alguna interpretación posible: por ejemplo: "el semáforo está fuera de servicio".

Ventajas y limitaciones al trabajar con Flujogramas.

Por resumir algunas de las ventajas, señalamos las siguientes:

1. El flujograma es un sistema de representación que hace fácil al programador comprender la lógica de la aplicación que se desarrolla. Es más fácil también su explicación y permite documentar el algoritmo de una manera independiente al código concreto que lo implementa.
2. Un flujograma general de toda una aplicación (que pueden llegar a ser muy complejas y extensas) cartografía las principales líneas de la lógica del programa. Y llega a ser el modelo del sistema, que puede luego ser descompuesto y desarrollado en partes más detalladas para el estudio y un análisis más profundo de sistema.

3. Una vez un flujograma está terminado, es relativamente sencillo (sólo "relativamente") escribir el programa correspondiente, y el flujograma actúa como mapa de ruta de lo que el programador debe escribir. Hace de guía en el camino que va desde el inicio hasta el final de la aplicación, ayudando a que no quede omitido ningún paso.
4. Aún en los casos en que el programador haya trabajado con sumo cuidado y haya diseñado cuidadosamente la aplicación a desarrollar, es frecuente que se encuentren errores, quizá porque el diseñador no haya pensado en una casuística particular. Esos errores son detectados sólo cuando se inicia la ejecución del programa en el ordenador. Este tipo de errores se llaman "*bugs*", y el proceso de detección y corrección se denomina "*debugging*". En esta tarea el flujograma presta una ayuda valiosa en la labor de detección, localización y corrección sistemática de esos errores.

Y por resumir algunas de las principales limitaciones o inconvenientes que trae consigo trabajar con flujogramas, señalamos los siguientes:

1. La creación de un flujograma, y sobre todo la tarea de su dibujo exige un consumo importante de tiempo.
2. Las modificaciones introducidas en las especificaciones de un programa exigen muchas veces la creación de un nuevo flujograma. Con frecuencia no es posible trabajar de forma eficiente sobre los flujogramas de las versiones anteriores.
3. No existe un estándar que determine la información y el detalle que debe recoger un flujograma.

Flujogramas estructurados y no estructurados.

Es frecuente, sobre todo en los primeros pasos de aprendizaje en la construcción de flujogramas, cometer el error de diseñar uno que no cumpla con las reglas de la programación estructurada. Estas reglas ya

han quedado recogidas en los epígrafes anteriores, aunque convendrá, de nuevo, sumariarlos en los siguientes cuatro puntos:

1. Toda sentencia simple puede ser sustituida por una estructura básica secuencial se sentencias simples.
2. Toda sentencia simple puede ser sustituida por una estructura básica de decisión, sea ésta abierta o cerrada, o por una estructura derivada CASE.
3. Toda sentencia simple puede ser sustituida por una estructura básica o derivada de iteración.
4. Las sustituciones indicadas en 1, 2 y 3 pueden realizarse tantas veces como sea necesario, llevando a anidar unas estructuras dentro de otras. No hay, teóricamente, límite en el número de anidaciones.

Existen infinidad de posibles flujogramas, que presentan una solución válida para un problema que se desea resolver mediante un algoritmo informático, pero que no cumplen las reglas de la programación estructurada. No es siempre fácil detectar las violaciones a esas reglas. Pero es necesario no dar por bueno un algoritmo expresado mediante flujograma hasta tener evidencia de que respeta las reglas del paradigma de la programación estructurada (para conocer el concepto de paradigma, ver Capítulo 6).

Vea por ejemplo el flujograma recogido en la Figura 5.5. Da igual cuáles sean las sentencias y las condiciones de nuestro algoritmo. Lo importante ahora es comprender que el flujo definido mediante las líneas de flujo está violando esas reglas de la programación estructurada. Una de las dos líneas apuntadas con una flecha blanca no debería estar allí: la que va desde el diamante C2 hasta la sentencia S3. El diamante con la condición C1 forma parte, claramente, de una estructura de iteración: es la que queda enmarcada sobre el fondo sombreado. Y esta estructura, como todas las iteradas, debería tener un único camino de iteración (señalado en la línea de flujo que asciende por

la izquierda) y un camino alternativo de salida de la iteración que es el que en este caso corre por la derecha. ¿Y no son aquí dos las líneas de salida del bloque de iteración?: vea sino las dos líneas señaladas con una flecha.

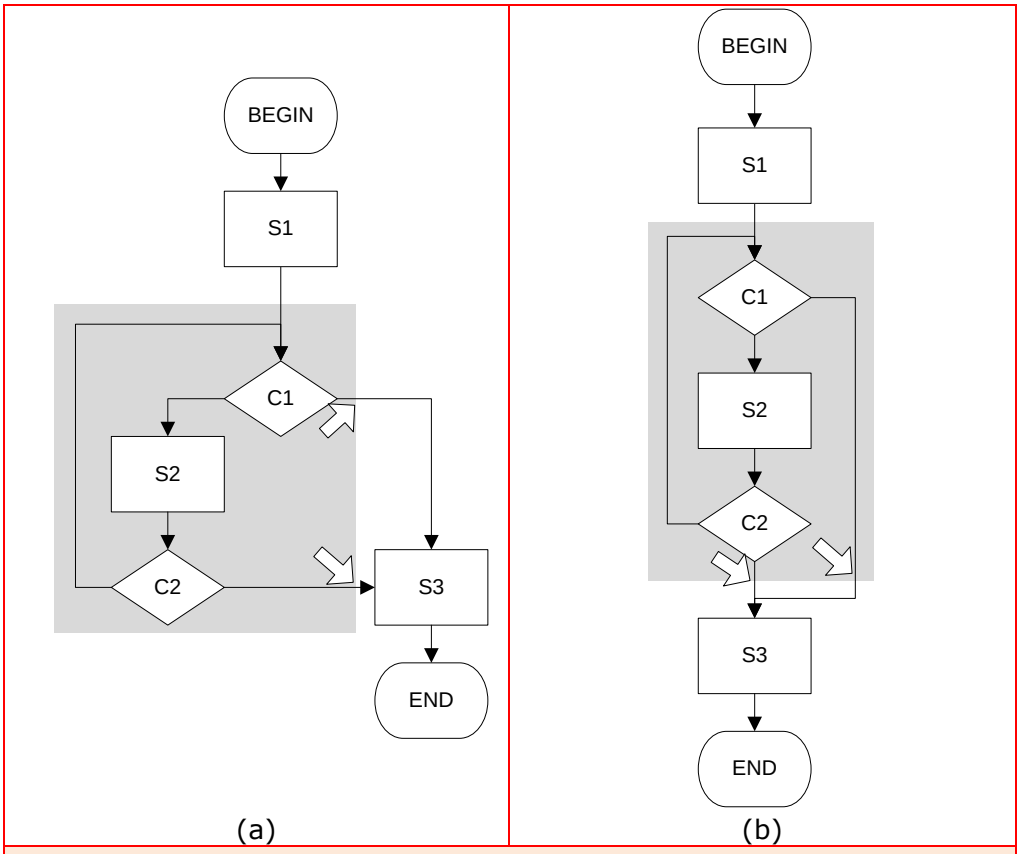


Figura 5.5. Ejemplo de un flujograma que no respeta las reglas de la programación estructurada. Se muestra de dos formas. La presentada en (b) tiene una forma completamente vertical, que ofrece, en opinión del autor de este manual, unas flujogramas más estilizados pero también menos claros.

Este tipo de errores no son, al menos al principio, fácilmente identificables. Menos aún a veces cuando los flujogramas suelen dibujarse en un formato más bien vertical, como el mostrado en la

Figura 5.5.b. Y muchos lenguajes, incluido el C, son capaces de implementar una aplicación como la diseñada con el flujograma de la Figura 5.5.: pero ese tipo de código (llamado vulgarmente código 'spaghetti') resulta con frecuencia notablemente más difícil de entender, de implementar, de depurar y de mantener que el código de una aplicación diseñada de forma coherente con las cuatro reglas indicadas de la programación estructurada.

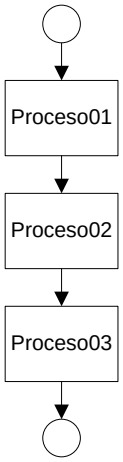
Pseudocódigo.

El pseudocódigo es otra herramienta habitual y útil para expresar la lógica de un programa. El prefijo "*pseudo*", modifica el significado de la palabra a la que precede y le otorga el carácter de "falso", o de "imitación de". El pseudocódigo es una imitación del código que puede escribirse con un lenguaje de programación. Cada "pseudo instrucción" es una frase escrita en un lenguaje natural (castellano, por ejemplo; o inglés, más frecuentemente). En el pseudocódigo, en lugar de utilizar un conjunto de símbolos para expresar una lógica, como en el caso de los flujogramas, lo que se hace es hacer uso de una colección de estructuras sintácticas que imitan la que se utilizan, de hecho, en el código escrito en un lenguaje de programación cualquiera.

Esas estructuras de las que hace uso el pseudocódigo son asombrosamente pocas. Sólo tres son suficientes para lograr expresar cualquier programa estructurado escrito en un ordenador. Estas estructuras, que ya hemos visto al presentar las estructuras básicas de un flujograma y que vienen también recogidas en las reglas básicas de la programación estructurada, son:

1. La **secuencia**. La secuencia lógica se usa para indicar las sentencias o instrucciones que, una detrás de la otra, configuran el camino a recorrer para alcanzar un resultado buscado. Las instrucciones de pseudocódigo se escriben en un orden, o secuencia, que es en el que

éstas deberán ser ejecutadas. El orden lógico de ejecución de estas instrucciones va de arriba hacia abajo, aunque es frecuente enumerarlas para remarcar ese orden de ejecución. La Figura 5.6. recoge la equivalencia entre ambas representaciones (pseudocódigo y flujograma) de la secuencia de instrucciones.




```
.
.
Proceso01
Proceso02
Proceso03
.
.
```

Figura 5.6. Equivalencia entre la secuencia de instrucciones representada con un flujograma y con pseudocódigo.

2. La **selección**. La selección o decisión lógica se emplea en aquellos puntos del algoritmo en los que hay que tomar una decisión para determinar si una determinada sentencia o grupo de sentencias se deben ejecutar o no, o para determinar cuáles, entre dos o más bloques, se han de ejecutar finalmente. La lógica de la selección se expresa con las estructuras que ofrecen las siguientes expresiones:

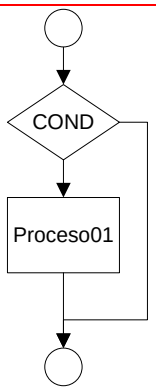
IF COND
THEN
PROCEDIMIENTO 1
ELSE
PROCEDIMIENTO 2
END IF
	IF COND
	THEN
	PROCEDIMIENTO
	END IF.

Bifurcación Abierta.	Bifurcación Cerrada.

La Figura 5.7. recoge un ejemplo de estas estructuras de pseudocódigo, expresadas también con el flujograma equivalente. Si la condición expresada en la estructura de selección se evalúa como verdadera, entonces se ejecuta el proceso Proceso01. Si no (si la condición se evalúa como falsa), en el caso recogido en la Figura 5.7.(a), no se ejecuta ninguna instrucción o sentencia; en el ejemplo de la Figura 5.7.(b) se ejecutará el proceso Proceso02.

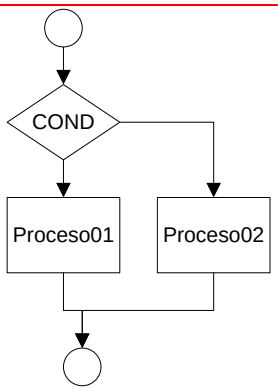
En ambos modelos (bifurcación abierta o cerrada) los procesos a ejecutar estarán compuestos por una o más sentencias independientes. No hay limitación alguna en el número de instrucciones o sentencias que se van a poder ejecutar dentro de cada uno de los dos caminos de la bifurcación.

La marca END IF señala el final de los caminos de las sentencias condicionadas. Más allá de esta marca, las sentencias que le sigan no estarán bajo el control de esta estructura.



(a)

IF COND is TRUE
THEN Proceso01
END IF

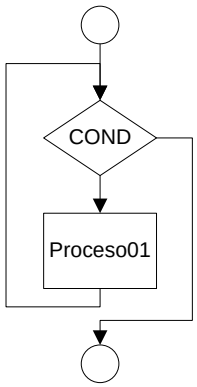


(b)

IF COND is TRUE
THEN Proceso01
ELSE Proceso02
END IF

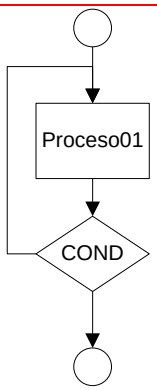
Figura 5.7. Equivalencia entre la selección expresada con un bloque de un flujograma y mediante la estructura de selección de un bloque de pseudocódigo. Se muestra la forma de expresarse una bifurcación abierta y una bifurcación cerrada.

3. La **iteración**. La iteración lógica se usa cuando una o más instrucciones deben ser ejecutadas un número indeterminado de veces. La decisión sobre cuántas veces deban ser ejecutadas esas instrucciones dependerá de cómo se evalúe una condición que podemos llamar condición de permanencia, que se recoge mediante una expresión lógica y que se evalúa como verdadera o falsa.



(a)

WHILE COND is TRUE
Proceso01
END WHILE



(b)

DO
Proceso01
WHILE COND is TRUE
END DO

Figura 5.8. Equivalencia entre la iteración expresada con un bloque de un flujograma y mediante la estructura de iteración de un bloque de pseudocódigo.

Hay dos formas habituales de expresar la iteración lógica. La Figura 5.8. muestra ambas posibilidades. La primera se corresponde con la iteración presentada como estructura básica de iteración de un flujograma (cfr. Figura 5.2.(c)); la segunda se corresponde con la iteración presentada como estructura derivada de iteración (cfr. Figura 5.3.(a)).

Es importante darse cuenta de que con estas estructuras de repetición, las instrucciones iteradas serán ejecutadas una y otra vez mientras la condición de permanencia siga evaluándose como verdadera. Por lo

tanto es necesario que entre esas instrucciones haya alguna o algunas sentencias que puedan hacer cambiar el valor de esa condición de control de permanencia.

Ventajas y limitaciones al trabajar con Pseudocódigo.

Antes hemos señalado algunas ventajas e inconvenientes de trabajar con flujogramas; ahora mostramos las de trabajar con el pseudocódigo. Podemos destacar las siguientes:

1. Transformar el pseudocódigo en código escrito en un lenguaje determinado es siempre una tarea trivial. El proceso de traducción de pseudocódigo a código es definitivamente más sencillo que el que se sigue cuando se parte del flujograma.
2. El pseudocódigo facilita la modificación de la lógica del programa, la introducción o eliminación de sentencias y estructuras.
3. El trabajo de redacción del pseudocódigo es siempre más sencillo que la creación de los gráficos de los flujogramas.
4. La lógica del pseudocódigo es muy sencilla: únicamente tres estructuras posibles. El programador puede concentrar todo su esfuerzo en la lógica del algoritmo.

También podemos sumariar algunos inconvenientes o limitaciones:

1. No se dispone de una representación gráfica de la lógica del programa que, sin duda, ayuda enormemente a su comprensión.
2. No hay reglas estándares para el uso del pseudocódigo. Al final, cada programador termina por establecer su propio estilo.
3. Habitualmente, al programador novel le resulta más complicado seguir la lógica del algoritmo expresada mediante pseudocódigo que aquella que se representa mediante flujogramas

Un primer ejemplo de construcción de algoritmos.

A partir de este momento en que ya ha quedado terminada la presentación teórica de cómo expresar los algoritmos, vamos a mostrar algunos ejemplos de cómo construir y expresar una serie de algoritmos sencillos.

Comenzamos por plantear el problema del cálculo del factorial de un entero cualquiera.

El factorial de un número se define como el producto de todos los enteros positivos igual o menores que ese número del que queremos calcular su factorial: $n!=n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 2\cdot 1$.

Un algoritmo válido para el cálculo del factorial de un entero podría representarse en pseudocódigo de la siguiente manera:

ENUNCIADO: Dado un entero positivo n, calcular su factorial.

1. Inicializar Variables:
   1.1. `Fact` ← 1.
   1.2. El usuario del programa introduce el valor de n.
2. **WHILE** $n \neq 0$
   2.1. `Fact` ← `Fact` · n.
   2.2. n ← n – 1.
   END WHILE
3. Resultado: `Fact`.

Cada sentencia del algoritmo se representa mediante una frase sencilla o con una expresión aritmética. Por ejemplo, La expresión `Fact` ← 1 indica que se asigna el valor 1 a la variable `Fact`. La expresión `Fact` ← `Fact` · n indica que se asigna a la variable `Fact` su valor actual multiplicado por el valor que en ese momento tiene la variable n.

Podemos probar si el algoritmo, tal y como está escrito, ofrece como resultado el valor del factorial del valor de entrada n. En Cuadro 5.1. se recoge la evolución de las variables n y `Fact` para un valor inicial de b igual a 4.

	Sentencia	Valor de la condición
1.1.	$Fact \leftarrow 1$.	
1.2.	$n = 4$.	($n = 4$: distinto de cero)
2.1.	$Fact \leftarrow Fact \cdot n$, es decir, $Fact = 4$.	
2.2.	$n \leftarrow n - 1$, es decir, $n = 3$.	($n = 3$: distinto de cero)
2.1.	$Fact \leftarrow Fact \cdot n$, es decir, $Fact = 12$.	
2.2.	$n \leftarrow n - 1$, es decir, $n = 2$.	($n = 2$: distinto de cero)
2.1.	$Fact \leftarrow Fact \cdot n$, es decir, $Fact = 24$.	
2.2.	$n \leftarrow n - 1$, es decir, $n = 1$.	($n = 1$: distinto de cero)
2.1.	$Fact \leftarrow Fact \cdot n$, es decir, $Fact = 24$.	
2.2.	$n \leftarrow n - 1$, es decir, $n = 0$.	($n = 0$: condición falsa)
3.	Resultado: $Fact = 24$.	

Cuadro 5.1. Evolución de los valores de las variables en el algoritmo de cálculo del Factorial de n, con valor inicial n = 4.

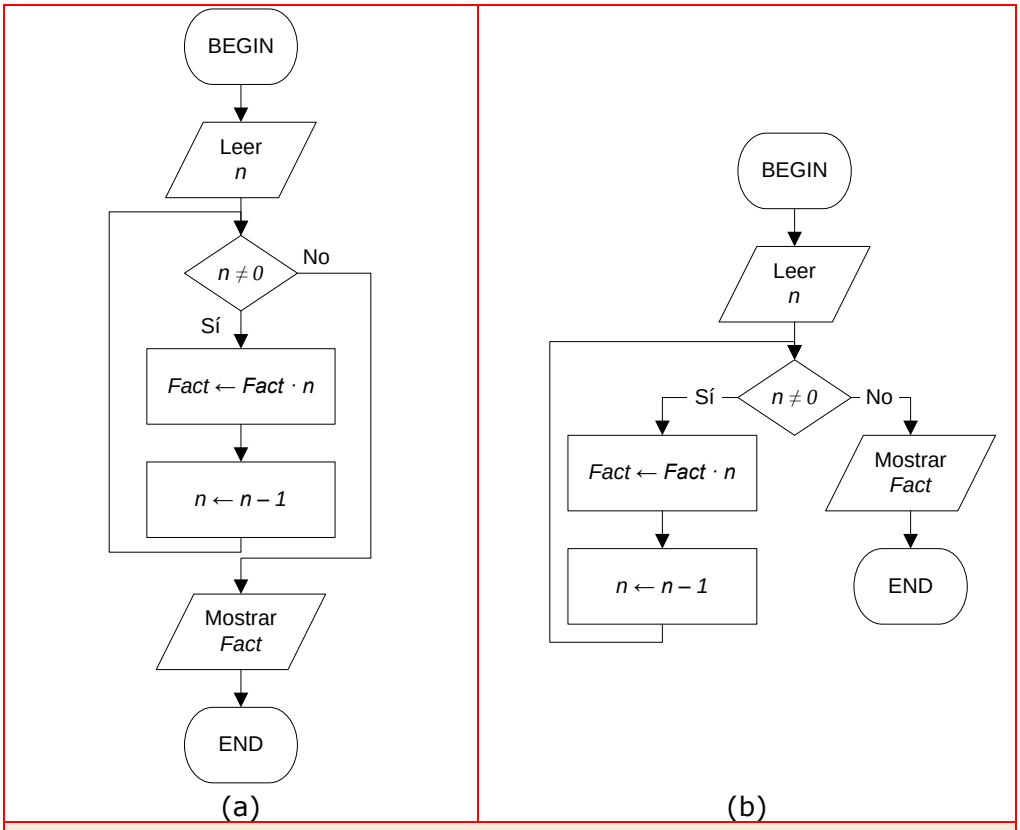


Figura 5.9. Flujograma que presenta un posible algoritmo para el cálculo del factorial de un entero codificado en la variable *n*.

El flujograma de este algoritmo queda recogido en la Figura 5.9. En este caso hemos presentado el flujograma en un diseño más estilizado o vertical y en otro menos vertical pero quizá visualmente más claro. Iremos utilizando ambas formas indistintamente.

Más ejemplos de construcción de algoritmos.

Aunque no se ha llegado todavía a explicar en este libro cómo se expresan en el lenguaje C las diferentes estructuras básicas de la programación estructurada; y aunque sólo se ha visto hasta el momento un capitulo introductorio sobre ese lenguaje; incluimos, al final del capítulo, el código, en C, que expresa los algoritmos que se proponen como solución a los enunciados.

5.1. Proponer un algoritmo que indique cuál es el menor de tres enteros recibidos.

ENUNCIADO: Dados tres valores a, b, y c, asignar a la variable m el menor de esos tres valores.

1. Inicializar Variables: El usuario introduce los valores a, b, y c
2. Asignación: m ← a
3. **IF** m > b
   THEN
   m ← b
   END IF
4. **IF** m > c
   THEN
   m ← c
   END IF
5. Resultado: m.

Inicialmente se asigna a la variable m el primero de los tres valores introducidos (valor de a). Se compara ese valor con el de la variable b: si resulta mayor, entonces se debe asignar a la variable m el nuevo

valor, menor que el primero introducido; si no es mayor no hay que cambiar el valor de m que en tal caso aguarda, efectivamente, un valor que es el menor entre a y b. Y a continuación se compara `m` con `c`, y se resulta mayor se guarda en `m` ese valor, menor que el menor entre los anteriores a y b. Así, al final, `m` guarda el menor de los tres valores.

El flujograma queda recogido en la Figura 5.10. En el podemos ver el sentido y uso de los conectores: más allá del final de la línea de ejecución de la izquierda, se continúa con el diagrama que viene en la línea de ejecución de la derecha: los puntos marcados con el número 1 son coincidentes.

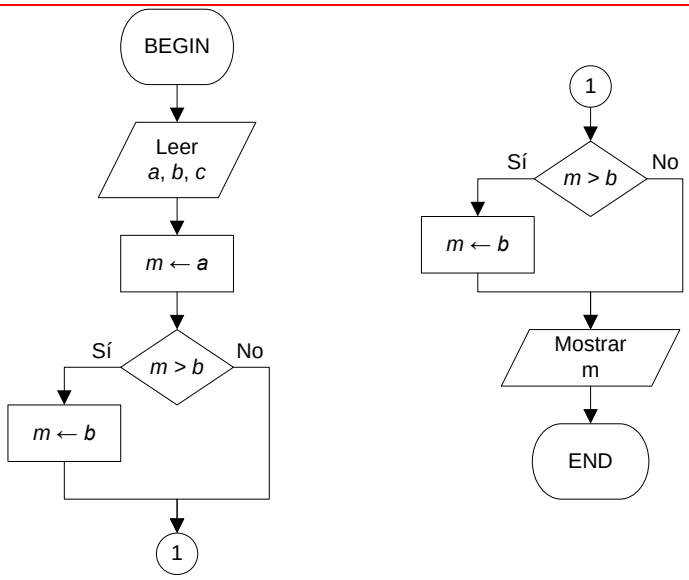


Figura 5.10. Flujograma que plantea una solución al ejercicio 5.1.

Este problema planteado pudiera tener otras soluciones. Veamos esta segunda posibilidad, bastante parecida a la primera, pero donde ahora tendremos anidamiento de bifurcaciones:

ENUNCIADO: Dados tres valores a, b, y c, asignar a la variable `m` el menor de esos tres valores.

1. Inicializar Variables: El usuario introduce los valores a, b, y c.
2. **IF** a < b
   THEN
      IF a < c
      THEN m ← a.
      ELSE m ← c.
      END IF
   ELSE
      IF b < c
      THEN m ← b
      ELSE m ← c
      END IF
   END IF
3. Resultado: m.

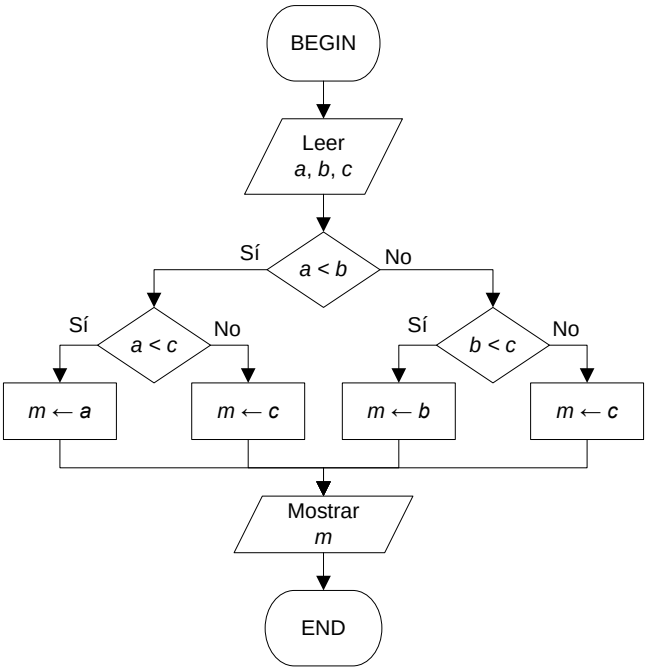


Figura 5.11. Flujograma que plantea una solución al ejercicio 5.1.

El flujograma queda recogido en la Figura 5.11. y un posible código en C de estos dos algoritmos se muestra, al final del Capítulo, en Código 5.1. y en Código 5.2.

5.2. Proponer un algoritmo que indique si un entero recibido es par o es impar

Una solución sencilla para saber si un entero es par o impar será dividir el número por 2 y comprobar si el resto de este cociente es cero. Si el resto es igual a 1, entonces el entero es impar; de lo contrario, el entero será par. A la operación de cálculo del resto de una división entera se le suele llamar operación módulo (a módulo b, ó a `mod` b).

Veamos el pseudocódigo:

ENUNCIADO: Dado un entero, determinar si éste es par o impar.

```
1.  Inicializar Variables: El usuario introduce el valor del entero n
2.  resto ← n mod 2
3.  IF resto es igual a cero
    THEN
        Mostrar que el entero es PAR.
    ELSE
        Mostrar que el entero es IMPAR
    END IF
```

Averiguar si un entero es par o impar es una tarea verdaderamente sencilla. Y es evidente que no hay un modo único o algoritmo para averiguarlo. Existen otros procedimientos o algoritmo posibles para resolver este problema. Por ejemplo, si el número se puede obtener como suma de doses, entonces el entero es par. Si en un momento determinado, a la suma de doses hay que añadirle un uno, entonces el número es impar. Un pseudocódigo que explique este algoritmo podría ser el siguiente:

ENUNCIADO: Dado un entero, determinar si éste es par o impar.

```
1.  Inicializar Variables: El usuario introduce el valor del entero n.
2.  WHILE n ≥ 2
        n ← n – 2
    END WHILE
3.  IF n = 2
    THEN
```

Mostrar que el entero es PAR.
ELSE
Mostrar que el entero es IMPAR
END IF

El Flujograma de ambos pseudocódigos puede verse en la Figura 5.12. Código 5.3 y 5.4. muestran una posible implementación en C.

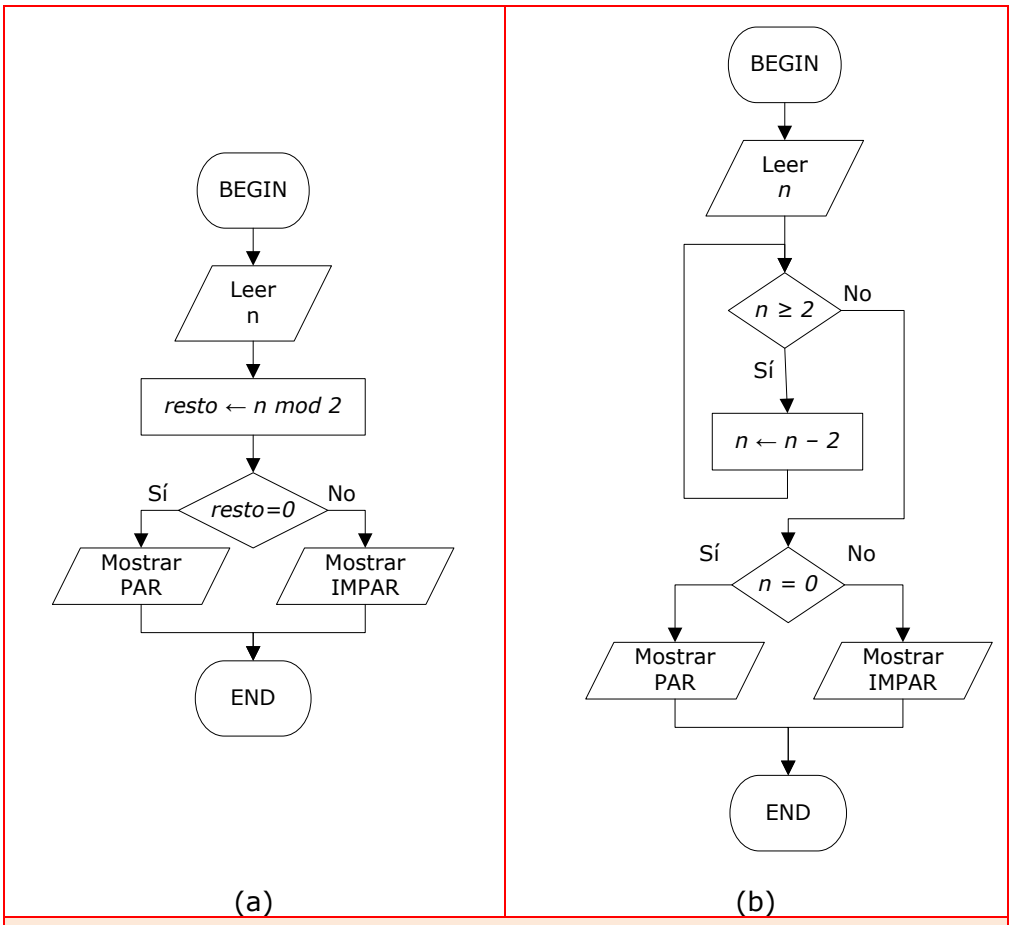


Figura 5.12. Dos formas de resolver el problema 5.2.

5.3. ***Proponer un algoritmo que muestre el resultado de una potencia de la que se recibe la base y el exponente. El exponente debe ser un entero mayor o igual que cero.***

Un modo de calcular la potencia es realizar el producto de la base tantas veces como indique el exponente. Hay que tener la precaución de que si el exponente es igual a cero, entonces nuestro algoritmo ofrezca como salida el valor uno.

El flujograma del algoritmo se muestra en la Figura 5.13. Su forma con el pseudocódigo sería de la siguiente manera:

ENUNCIADO: Dados dos enteros (b y e ≥ 0) calcular el valor que r que resulta del cálculo de la potencia de b elevado a e.

1. Inicializar Variables:
   1.1. El usuario introduce los valores b y e
   1.2. r ← 1
2. **WHILE** e ≠ 0
   2.1. r ← r·b
   2.2. e ← e – 1
   END WHILE
3. Resultado: r.

En el caso de que la condición evaluada en el paso 2 sea verdadera, ocurrirá que se ejecutarán una serie de instrucciones y, de nuevo se volverá a evaluar la condición del paso 2. Vemos por tanto que mediante la evaluación de expresiones se puede determinar que se ejecute una instrucción u otra de nuestro algoritmo (diamante en una estructura de decisión); y también que se ejecute una instrucción o un bloque de instrucciones o una sola vez, o varias veces, o ninguna vez (diamante en una estructura de iteración), dependiendo de que se siga cumpliendo o no la condición de permanencia.

¿Ha comparado el flujograma mostrado en la Figura 5.13. con el de la Figura 5.9.(b)? ¿Ve alguna coincidencia? Es conveniente que se dé cuenta de que con frecuencia el procedimiento a seguir en ejercicios similares es prácticamente el mismo. Calcular una determinada potencia de un número, o calcular su factorial se reduce, en ambos casos, ha acumular un determinado número de veces un producto. Es lógico que los algoritmos sean muy similares. El código en C de este algoritmo se muestra en Código 5.5.

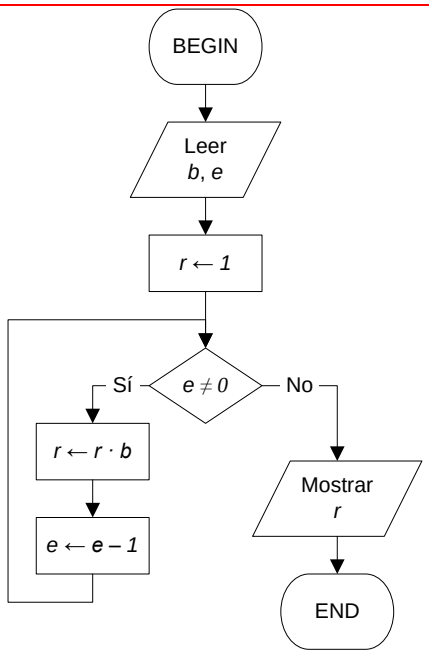


Figura 5.13. Flujograma que representa el algoritmo de cálculo de la potencia de un entero elevado a otro entero positivo.

5.4. *Algoritmo de Euclides para el cálculo de máximo común divisor de dos enteros positivos.*

Dados dos enteros positivos `m` y `n`, el algoritmo debe devolver el mayor entero positivo que divide a la vez a `m` y a `n`. Euclides demostró una propiedad del máximo común divisor de dos enteros que permite definir un procedimiento (algoritmo) sencillo para calcular ese valor. Según la propiedad demostrada por Euclides, se verifica que el máximo común divisor de dos enteros positivos cualesquiera `m` y `n` (`mcd(m, n)`) es igual al máximo común divisor del segundo (`n`) y el resto de dividir el primero por el segundo: `mcd(m, n) = mcd(n, m mod n)`. A la operación del cálculo del resto del cociente entre dos enteros, la llamamos operación módulo. Esta operación está explícitamente definida en el lenguaje C.

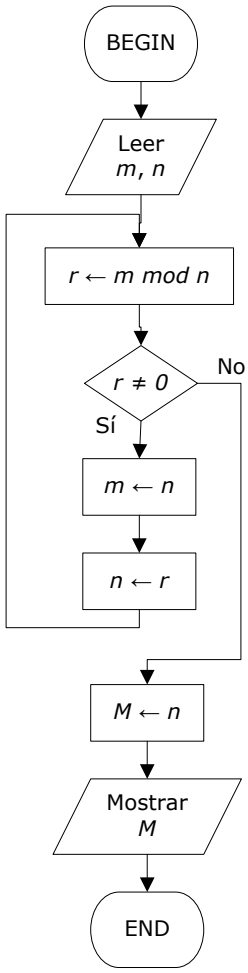


Figura 5.14. Flujograma correspondiente al algoritmo de Euclides.

Si tenemos en cuenta que el máximo común divisor de dos enteros donde el primero es múltiplo del segundo es igual a ese segundo entero, entonces ya tenemos un algoritmo sencillo de definir: Cuando lleguemos a un par de valores (`m, n`) tales que `m` sea múltiplo de `n`, tendremos que `m mod n = 0` y el máximo común divisor buscado será `n`.

Si ha comprendido el algoritmo de Euclides, entonces es sencillo ahora definir el algoritmo usando pseudocódigo:

ENUNCIADO: Dados dos enteros (m y n) calcular el valor del máximo común divisor, M, de m y n. (M = mcd(m, n))

1. Inicializar Variables: El usuario introduce los valores m y n.
2. r ← m mod n
3. **WHILE** r ≠ 0
   3.1. m ← n
   3.2. n ← r
   END WHILE
4. M = n
5. Resultado: M.

El flujograma de este algoritmo queda representado en la Figura 5.14. Y de nuevo, si lo compara con el flujograma de las Figuras 5.9. y 5.13. volverá a ver la misma estructura.

5.5. Algoritmo que muestra la tabla de multiplicar de un entero.

Este enunciado no requiere presentación. Simplemente se trata de un programa que solicita del usuario un valor entero, y entonces el programa muestra por pantalla la clásica tala de multiplicar. Veamos el pseudocódigo:

ENUNCIADO: Mostrar la tabla de multiplicar del entero n introducido por el usuario.

1. Inicializar Variables:
   1.1. El usuario introduce el valor n.
   1.2. i ← 0
2. **WHILE** i ≤ 10
   2.1. pr ← i · n
   2.2. Mostrar el valor de las tres variables: i , n , pr
   2.3. i ← i + 1
   END WHILE

Y el flujograma queda recogido en la Figura 5.15. El código en C podría ser el sugerido en Código 5.6.

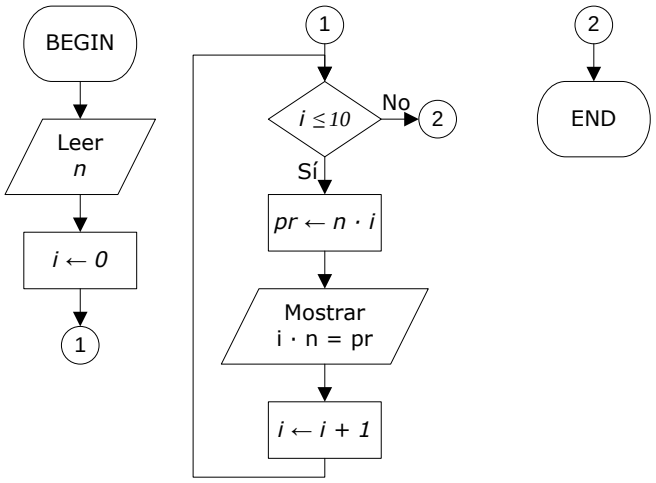


Figura 5.15. Flujograma que plantea una solución al ejercicio 5.5.

5.6. *Algoritmo que muestra un término cualquiera de la serie de Fibonacci.*

Fibonacci fue un matemático italiano del siglo XIII que definió la serie que lleva su nombre. Cada valor de esta serie es el resultado de la suma de los dos anteriores. Y los dos primeros elementos de la serie son unos.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Al margen de propiedades que pudieran facilitar el cálculo de un elemento cualquiera de la serie, el algoritmo que, con lo que sabemos de esta serie, resuelve nuestro problema, pudiera tener el siguiente pseudocódigo (el diagrama de flujo del algoritmo queda recogido en la figura 5.16.; en Código 5.7. se propone su implementación):

ENUNCIADO: Mostrar el término de la serie de Fibonacci que solicite el usuario del programa.

1. Inicializar Variables:
   1.1. El usuario introduce el valor n.

```
        1.2.   fib1 ← 1
        1.3.   fib2 ← 1
        1.4.   Fib ← 1
        1.5.   i ← 2
2.      WHILE i ≤ n
        2.1.   Fib ← fib1+fib2
        2.2.   fib1 ← fib2
        2.3.   fib2 ← Fib
        2.4.   i ← i + 1
        END WHILE
3.      Resultado: Fib.
```

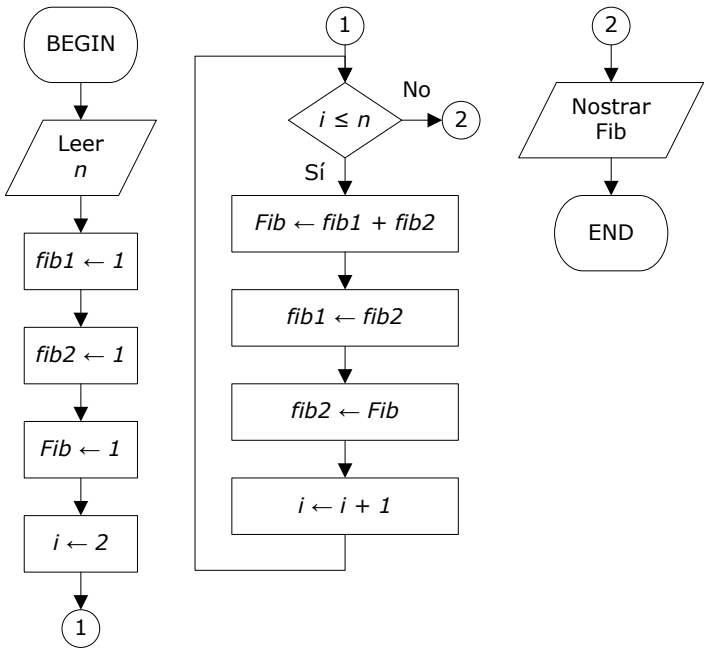


Figura 5.16. Flujograma correspondiente al enunciado 5.6. que solicita el algoritmo que muestre el elemento n-ésimo de la serie de Fibonacci.

Recapitulación.

En este capítulo hemos definido el concepto de algoritmo, y hemos presentado dos herramientas para su formulación: el pseudocódigo y los flujogramas o diagramas de flujo. Hemos distinguido entre sentencias o

instrucciones y estructuras de control, que permiten determinar qué instrucciones se deben ejecutar en cada momento y cuáles no. Hemos empleado estructuras de decisión y de iteración. Y hemos mostrado diferentes ejemplos de construcción de algoritmos.

Ya hemos dicho que la construcción de algoritmos requiere cierta dosis de oficio y experiencia. No se ganó Zamora en una hora. Se proponen a continuación algunos ejercicios más que pueden ayudar a afianzar los conceptos y herramientas recién estudiados.

Otros ejercicios propuestos.

5.7. ***Diseñe un algoritmo que resuelva una ecuación de segundo grado. Tendrá como entrada los coeficientes a, b y c de la ecuación y ofrecerá como resultado las dos soluciones reales.***

Ante este enunciado lo primero que necesitamos es saber cómo se resuelve una ecuación de segundo grado. Con un polinomio de grado 2 igualado a cero, las soluciones posibles para esa ecuación, como ya se sabe, son

$$x_{1,2} = \frac{-b \pm \sqrt[2]{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

Por lo cual, nuestro algoritmo deberá hacer lo siguiente: Dados los tres coeficientes a, b, y c, lo primero será comprobar que la ecuación es, efectivamente de segundo grado, es decir, que el coeficiente a es distinto de cero. Si no tenemos una ecuación de segundo grado, al menos podemos tener una de primer grado, pero para ello ahora hemos de verificar que el coeficiente b es distinto de cero: si no lo es, entonces sencillamente no tenemos ecuación; y si lo es, entonces su solución es $x \leftarrow -c/b$.

Si el coeficiente a es distinto de cero, entonces efectivamente tendremos una ecuación de segundo grado. Pero ahora lo siguiente que deberemos averiguar es si sus soluciones son reales o complejas. Para ello hay que ver el signo del discriminante. Si es negativo, entonces las raíces son complejas, con una parte real y una parte imaginaria.

El pseudocódigo podría ser el siguiente:

ENUNCIADO: Buscar las soluciones de una ecuación de segundo grado.

Inicializar Variables: entrada de valores a, b, y c.
IF a = 0
THEN
    IF b = 0
    THEN Mostrar Mensaje: "ERROR: NO HAY ECUACIÓN".
    ELSE
        resultado ← –c / b
        Mostrar resultado
    END IF
ELSE (es decir, tenemos que a es distinto de cero)
    discr ← b^2 – 4 · a · c
    IF discr ≥ 0
    THEN
        resultado1 ← (–b + $\sqrt[2]{\text{discr}}$/2·a)
        resultado2 ← (–b – $\sqrt[2]{\text{discr}}$/2·a)
        Mostrar resultado1 y resultado2.
    ELSE (es decir, tenemos soluciones imaginarias)
        real ← –b / 2 · a
        imag ← $\sqrt[2]{-\text{discr}}$ / 2 · a
        Mostrar real + imag * i
        Mostrar real - imag * i
    END IF
END IF

El flujograma podría ser el de la Figura 5.17. (suponemos que existe una forma sencilla de calcular la raíz cuadrada de un número.) El código de este algoritmo, escrito en lenguaje C, tiene la forma del Código 5.8. El archivo math.h define una colección de funciones de uso matemático. Entre ellas está la función `sqrt()`, que devuelve el valor de la raíz cuadrada del valor **`double`** que recibe como único parámetro de entrada.

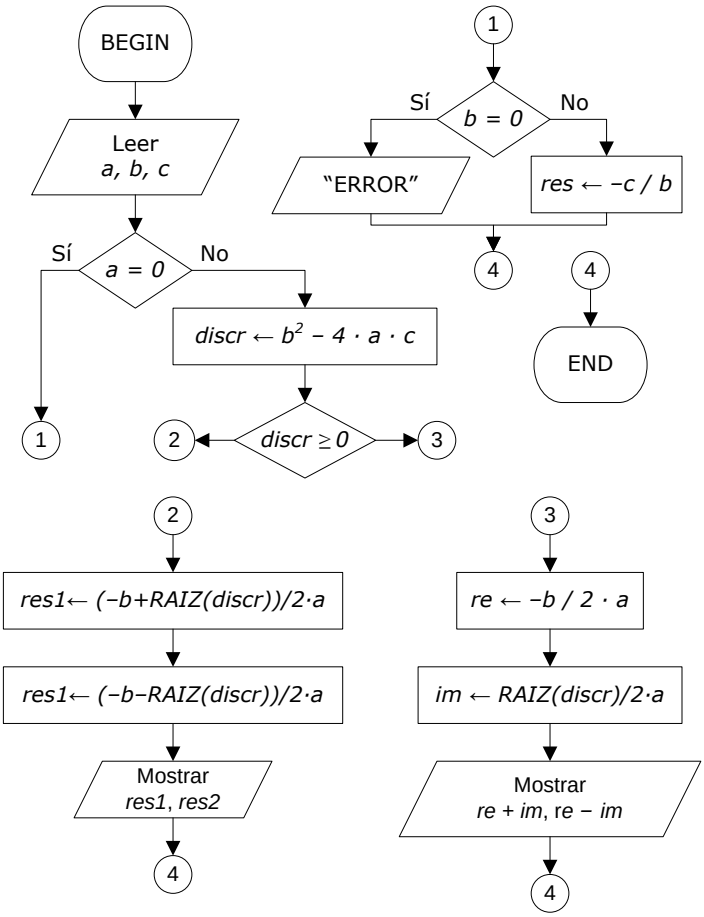


Figura 5.17. Algoritmo resolución de una ecuación polinómica de segundo grado.

5.8. *Diseñe un algoritmo que muestre todos los divisores de un entero que se recibe como entrada del algoritmo.*

Todos los divisores de un entero N son enteros comprendidos entre la unidad y la parte entera de la mitad de N, y el mismo N. Por ejemplo, los divisores de 18 son 2, 3, 6, y 9; a éstos se les añaden los dos divisores propios que son la unidad (el 1) y el mismo número (el 18). Nunca

habrá divisores entre la mitad y el mismo número, como en este caso no encontraremos ningún divisor entre 9 y 18. El diagrama de flujo podría ser el recogido en la Figura 5.18.

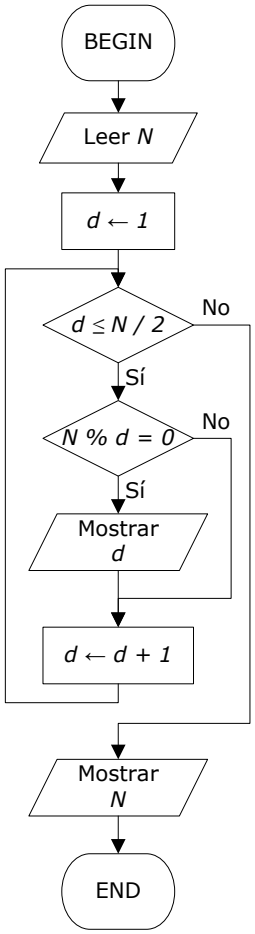


Figura 5.18. Flujograma del algoritmo que mostraría por pantalla los divisores del entero N introducido por el usuario.

5.9. ***Diseñe un algoritmo que recibe n enteros como entrada y muestra el mayor de ellos en la salida. Escriba el pseudocódigo y dibuje el flujograma.***

Intente resolverlo usted mismo, por su cuenta.

5.10. ***Diseñe un algoritmo que calcule el valor del número pi sabiendo que éste verifica la siguiente relación:***

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2 \cdot k + 1}$$

Escriba el pseudocódigo y dibuje el flujograma.

Intento resolverlo usted mismo, por su cuenta.

5.11. ***Haga lo mismo sabiendo que el número pi verifica:***

$$\frac{\pi^2}{6} = \sum_{k=1}^{\infty} \frac{1}{k^2}$$

Para solventar estos dos últimos ejercicios es evidente que no podemos realizar infinitas sumas: por más rápido que sume el ordenador, para realizar sumas sin fin nunca habrá tiempo suficiente.

Podemos solventar el problema planteado de una forma bastante simple sin más que limitando el número de operaciones a una cantidad alta pero finita: por ejemplo diez mil, o un millón de veces, por decir algo.

El pseudocódigo del segundo de estos dos últimos enunciados sería algo así como lo que a continuación se propone:

ENUNCIADO: Calcular el valor del número PI de acuerdo con la expresión del problema de Basilea, resuelto por Euler en 1735.

1. Inicializar Variables: entrada de valores
   1.1. `suma ← 0`
   1.2. `k ← 1`
2. **WHILE** $k \leq 10^6$

```
      2.1.  suma ← suma +  1/ k²
      2.2.  k ← k + 1
      END WHILE
3.    pi = RAIZ(6 · suma)
4.    Resultado: pi.
```

El código en C quedaría, por ejemplo, como se muestra en código 5.9.

5.12. ***Diseñe un algoritmo que calcule la media de todas los notas introducidas. Convenimos en que el proceso de entrada de datos termina cuando se introduzca una calificación negativa, ésta se considerará no válida e indicará el final de entrada de datos***

ENUNCIADO: Calcular la media de una serie de números introducidos por el usuario. El proceso terminará cuando éste introduzca un valor negativo, que no se considerará valor introducido a tener en cuenta.

```
1.    Inicializar Variables:
      1.1.  cont ← 0    (valores introducidos hasta el momento)
      1.2.  suma ← 0    (valor inicial de la suma de valores introd.)
2.    DO
      2.1.  Solicitar al usuario entrada
      2.2.  IF entrada ≥ 0
            THEN
            2.2.1.      suma ← suma  + entrada
            2.2.2.      cont ← cont + 1
      WHILE entrada ≠ 0
      END DO
3.    Media ← suma / cont
4.    Resultado: Media
```

Este pseudocódigo recoge la lógica del algoritmo que resuelve el problema del cálculo de la media de los valores introducidos por el usuario. Pero, quizás, no ha tenido en cuenta todas las posibilidades: ¿qué ocurriría si el usuario de nuestro programa introdujera, como primera entrada, un valor negativo?

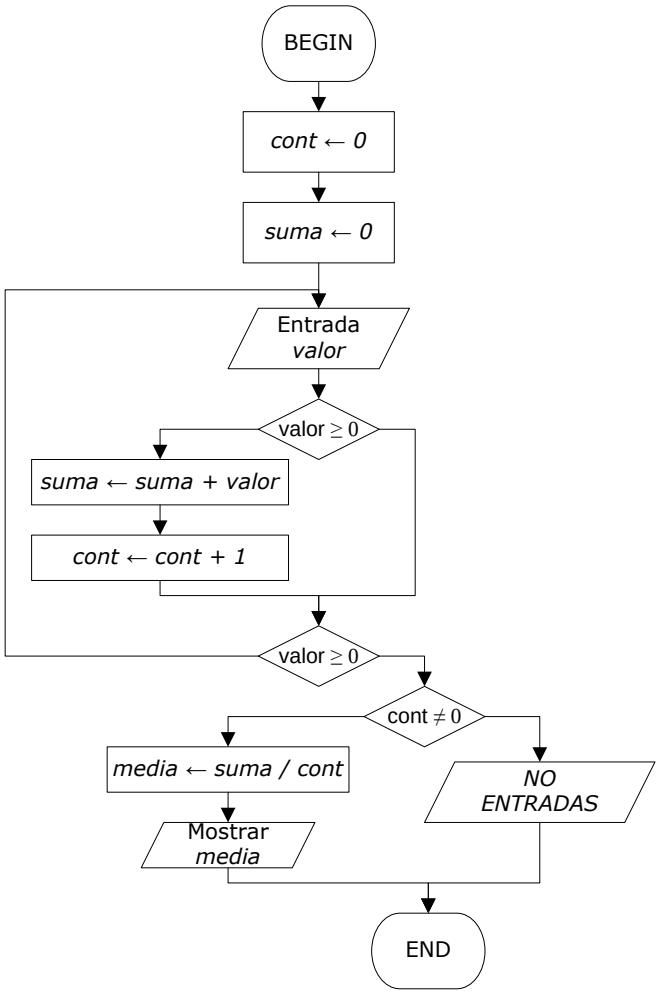


Figura 5.19. Flujograma del algoritmo que calcula la media de una serie de valores positivos introducidos por el usuario.

En ese caso el resultado que calcularíamos sería el de `0/0`, que como se sabe es una indeterminación. Habría que verificar que el usuario ha efectuado, al menos, una entrada distinta de cero. Para ello, el pseudocódigo de las sentencias 3 y 4 quedaría ligeramente modificado de la siguiente manera (Flujograma en la Figura 5.19.):

```
3.    IF cont ≠ 0
      THEN
      3.1.  Media ← suma / cont
```

3.2. Resultado: Media.
ELSE
3.1. Mostrar: "No se ha introducido valor alguno"
END IF

Implementación en C de algunos de los problemas planteados.

Código 5.1. Expresa en C el algoritmo del Flujograma 5.10.

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

      printf("Valor de a ... ");
      scanf(" %hd ", &a);
      printf("Valor de b ... ");
      scanf(" %hd ", &b);
      printf("Valor de c ... ");
      scanf(" %hd ", &c);

      m = a;            // Asignamos a m el valor de a.

      if(m > b)         // Si m es mayor que b ...
      {
            m = b;
      }

      if(m > c)         // Si m es mayor que c ...
      {
            m = c;
      }

      printf("\nEl menor valor introducido es %hd\n", m);

      return 0;
}
```

Código 5.2. Expresa en C el algoritmo del Flujograma 5.11.

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

      printf("Valor de a ... ");    scanf(" %hd ", &a);
      printf("Valor de b ... ");    scanf(" %hd ", &b);
      printf("Valor de c ... ");    scanf(" %hd ", &c);

      if(a < b)
      {
            if(a < c)
            {
                  m = a;
            }

            else
            {
                  m = c;
            }
      }

      else
      {
            if(b < c)
            {
                  m = b;
            }

            else
            {
                  m = c;
            }
      }
      printf("\nEl menor valor introducido es %hd\n", m);

      return 0;
}
```

Código 5.3. Expresa en C el algoritmo del Flujograma 5.12. (a)

```
#include <stdio.h>
int main(void)
{
      short int n , resto;
      printf("Valor de n ... ");    scanf(" %hd", &n);
      resto = n % 2;
      if(resto == 0)    printf("PAR");
      else              printf("IMPAR");
      return 0;
}
```

Código 5.4. Expresa en C el algoritmo del Flujograma 5.12. (b)

```
#include <stdio.h>
int main(void)
{
      short int n;
      printf("Valor de n ... ");    scanf(" %hd ", &n);

      while(n >= 2) n -= 2;

      if(n == 0) printf("PAR ");
      else printf("IMPAR ");

      return 0;
}
```

Código 5.5. Expresa en C el algoritmo del Flujograma 5.13.

```
#include <stdio.h>
int main(void)
{
      short b, e;
      long r;
```

Código 5.5. (Cont.)

```
      printf("Base ... ");          scanf(" %hd", &b);
      printf("Exponente ... ");     scanf(" %hd", &e);
      r = 1;
      while(e != 0)
      {
            r = r * b;
            e = e - 1;
      }
      printf("El valor de la potencia es %ld\n", r);
      return 0;
}
```

Código 5.6. Expresa en C el algoritmo del Flujograma 5.15.

```
#include <stdio.h>

int main(void)
{
      short i, N;
      printf("Valor de i ... ");    scanf(" %hd", &i);

      for(i = 0 ; i <= 10 ; i++)
      {
            printf("%hu * %hu = %hu\n", N, i, N * i);
      }
      return 0;
}
```

Código 5.7. Expresa en C el algoritmo del Flujograma 5.16.

```
int main(void)
{
      short int fib1 = 1 , fib2 = 1, Fib, i = 2 , n;
      printf("El valor de n ... ", n);    scanf(" %hd", &n);
```

Código 5.7. (Cont.)

```
      while(i <= n)
      {
            Fib  = fib1 + fib2;
            fib1 = Fib2;
            fib2 = Fib;
            i++;
      }

      printf("El elemento %hd es %hd.\n", n, Fib);
      return 0;
}
```

Código 5.8. Expresa en C el algoritmo del Flujograma 5.17.

```
#include <stdio.h>
#include <math.h>
int main (void)
{
      short a, b, c;
      double x1, x2, d;

      printf("COEFICIENTES ...\n");
      printf("Coeficiente a: ");    scanf(" %hd", &a);
      printf("Coeficiente b: ");    scanf(" %hd", &b);
      printf("Coeficiente c: ");    scanf(" %hd", &c);

      if(!a)
      {
            if(!b)
            {
                  printf("Ecuación Errónea.\n");
            }
            else
            {
                  printf("Solución: %lf\n", x1 = -c / b);
            }
      }
```

Código 5.8. (Cont.)

```
      else
      {
            d = (double)b * b – 4 * a * c;
            if(d >= 0)        // Soluciones Reales
            {
                  x1 = (-b + sqrt(d)) / ( 2 * a);
                  x2 = (-b - sqrt(d)) / ( 2 * a);
                  printf("x1: %lf\n", x1);
                  printf("x2: %lf\n", x2);
            }
            else              // Soluciones Imaginarias
            {
                  x1 = -b / (2 * a);
                  x2 = sqrt(-d) / (2 * a);
                  printf("sol1: %+lf %+lf * i\n", x1, x2)
                  printf("sol1: %+lf %+lf * i\n", x1, -x2)
            }
      }
      return 0;
}
```

Código 5.9. Solución, en C, del ejercicio 5.11.

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

int main (void)         {
      double suma = 0;
      long k = 1;

      while(k < 1000000)
      {           suma += 1 / ((double)k * k++);      }

      printf("El número PI es %8.6lf.\n" , sqrt(6 * suma));
      return 0;
}
```

CAPÍTULO 6

MODELO DE REPRESENTACIÓN.

En este capítulo queremos introducir una serie de conceptos necesarios para la comprensión del proceso de construcción de algoritmos y programas: la abstracción y la modularidad. También se incide en la noción de paradigma de programación, que ya se introdujo en el capítulo anterior.

El objetivo final del capítulo es lograr que se comprenda el modo en que se aborda un problema del que se busca una solución informática. Qué información del problema es importante y cómo se codifica, qué sentencias configuran los algoritmos que ofrecen solución al problema.

Introducción.

La creación de un programa no es una tarea lineal. Se trata de diseñar una solución, un algoritmo o conjunto de algoritmos, capaces de dar una solución al problema planteado. No existe habitualmente una única

solución, y muchas veces tampoco se puede destacar una de ellas como mejor que las demás. La solución adoptada debe ser eficiente, que logre hacer buen uso de los recursos disponibles. Uno de esos recursos es el tiempo: no todas las soluciones son igualmente rápidas. Hay muchos problemas para los que aún no se ha obtenido una solución aceptable.

Además, los programas necesitan con frecuencia modificaciones en sus instrucciones o en las definiciones de sus datos. Los problemas evolucionan y sus soluciones también. Poco a poco mejora la comprensión de los problemas que se abordan, y por tanto soluciones antes adoptadas necesitan pequeñas o no tan pequeñas modificaciones.

El primer paso cuando se pretende resolver un problema mediante medios informáticos consiste en la abstracción del problema en busca de un modelo que lo represente. Así, mediante la creación de una representación simplificada, se consideran sólo aquellos detalles que nos interesan para poder tratar el problema. Primeramente hay que determinar cuál es exactamente el resultado que se busca y se desea obtener; luego cuáles son los valores de entrada de los que disponemos, identificando aquellos que sean realmente necesarios para alcanzar de nuestro objetivo. Por último, hay que determinar con precisión los pasos que deberá seguir el proceso para alcanzar el resultado final buscado.

Al conjunto formado por el problema, con todos sus elementos y la solución buscada, y todo su entorno, es a lo que llamaremos **sistema**.

Abstracción.

La **abstracción** es la capacidad de identificar los elementos más significativos de un sistema que se está estudiando, y las relaciones entre esos elementos. La correcta abstracción del sistema que se aborda capacita para la construcción de modelos que permiten luego comprender la estructura del sistema estudiado y su comportamiento. Cada elemento que define a nuestro sistema tendrá una representación

en un tipo de dato y con un valor concreto tomado de un rango de valores posibles. Y cada relación entre elementos pude definirse mediante expresiones o procedimientos.

La abstracción es un paso previo en la construcción de cualquier programa. Fundamentalmente hablaremos de dos formas de abstracción:

1. Por un lado se deben determinar los **tipos de datos** que interviene en el sistema, es decir, cuál es el conjunto de parámetros que definen su estado en todo momento y su rango de valores posibles, y las operaciones que pueden realizarse con esos valores. También interesa determinar cuáles son los valores iniciales y los resultados finales que resumen los **estados inicial y final** del sistema.
2. Por otro lado, se debe también determinar las **funciones** o **procedimientos** del sistema: los procedimientos que definen su comportamiento.

Modularidad.

La **modularidad** es la capacidad de dividir el sistema sobre el que estamos trabajando en sus correspondientes partes diferenciadas (módulos), cada una de ellas con sus propias responsabilidades y sub-tareas. Para cada uno de los módulos deben quedar bien definidas sus relaciones con todos los demás módulos y su modo de comunicación con todo el resto del sistema.

Qué sea lo que se considera por módulo depende del paradigma de programación que se utilice. En el lenguaje C, que es un lenguaje del paradigma imperativo y estructurado (ya veremos más adelante en este capítulo estos conceptos) a cada módulo lo llamaremos **función** o **procedimiento**. En Java, que es un lenguaje de paradigma de programación orientado a objetos, un módulo puede ser una clase o cada uno de los métodos que implementa.

La modularidad permite convertir un problema en un conjunto de problemas menores, más fáciles de abordar. Así se logra la división del trabajo entre programadores o equipos de programadores, se aumenta la claridad del software que se desarrolla y se favorece la reutilización de parte del software desarrollado para problemas distintos para los que pudiera haber algún módulo semejante a los ya desarrollados. Además, en muchas ocasiones, este modo de trabajar reduce los costes de desarrollo del software y de su posterior mantenimiento.

Esta tarea, tan ventajosa, no es en absoluto trivial. Determinar correctamente los módulos que describen el funcionamiento del sistema y lograr definir las relaciones entre todos ellos puede llegar a ser muy complicado. De hecho esta modularización de un sistema se realiza siempre mediante técnicas de refinamientos sucesivos, pasando de un problema general a diferentes módulos que, a su vez, pueden considerarse como otro problema a modularizar; y así sucesivamente, hasta llegar a partes o módulos muy simples y sencillos de implementar.

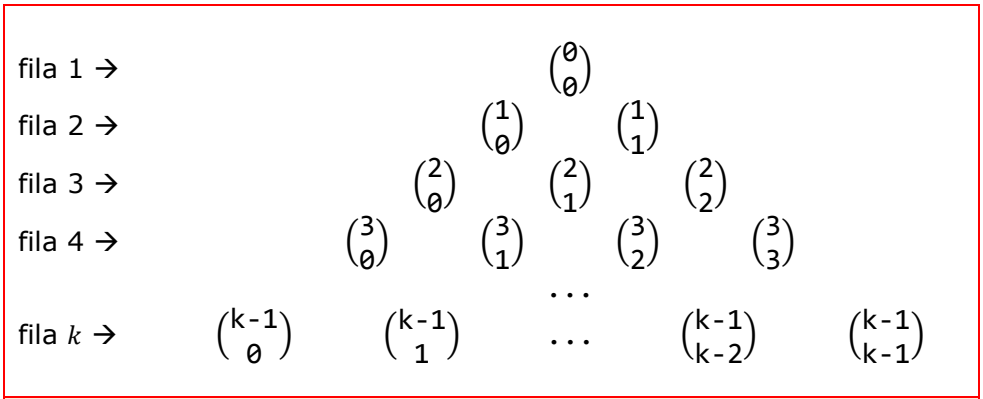


Figura 6.1. Distribución de valores en el Triángulo se Tartaglia.

Para mejor comprender este concepto de modularidad proponemos un ejemplo sencillo. Todos conocemos el **Triángulo de Tartaglia**: está formado por números combinatorios, ordenados de una forma concreta,

tal y como se muestra en la Figura 6.1., donde la expresión de los números combinatorios también es conocida:

$$\binom{m}{k} = \frac{m!}{k! \times (m-k)!}$$

Queremos hacer un programa que muestre una cualquiera de las filas del triángulo de Tartaglia. El programa preguntará al usuario qué fila desea que se muestre, y luego, mediante el algoritmo que definamos, calculará los valores de esa fila, que mostrará luego por pantalla.

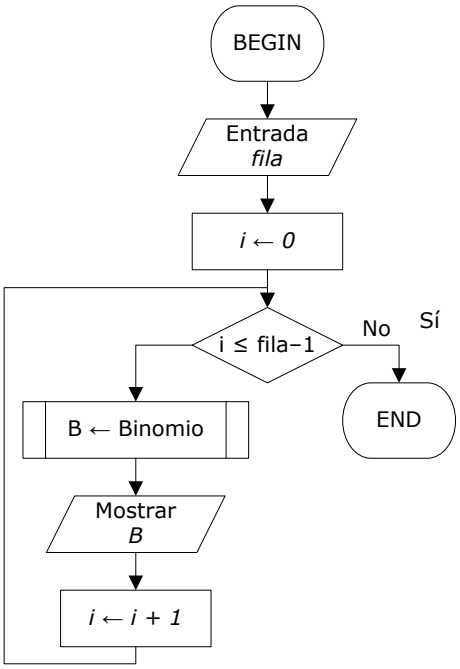


Figura 6.2. Algoritmo para cálculo del Triángulo de Tartaglia.

El algoritmo que logra resolver nuestro problema tendría la siguiente forma inicial:

ENUNCIADO: Hacer un programa que solicite al usuario un valor entero n y se muestren entonces los valores correspondientes a la fila n-ésima del triángulo de Tartaglia.

1. Inicializar Variables: El usuario introduce el valor de `fila`.
2. `i ← 0`
3. **WHILE** `i ≤ fila – 1`
   3.1. Mostrar valor `B ← Binomio(fila – 1 , i)`
   END WHILE

Donde vemos, en la línea 3.1., una llamada a procedimiento. El flujograma de este algoritmo quedaría como se recoge en la Figura 6.2., donde hemos utilizado un nuevo elemento para la construcción de flujogramas: la Llamada a **Procedimiento**. Este nuevo elemento tiene un comportamiento similar al símbolo básico de instrucción o sentencia: supone la definición de una colección de sentencias que, juntas, realizan un bloque del trabajo de la aplicación. Esta llamada a procedimiento significa y exige la creación de un nuevo programa o módulo. En nuestro ejemplo actual, exige la creación de un bloque de código que calcule el binomio o número combinatorio.

Hay, pues, que diseñar un nuevo algoritmo, que se habrá de emplear en nuestro primer algoritmo diseñado para mostrar una línea del triángulo de Tartaglia. Este segundo algoritmo podría tener la siguiente forma:

ENUNCIADO: Hacer un programa que recibe dos valores enteros positivos, el primero mayor o igual que el segundo, y calcula el binomio de newton del primero sobre el segundo.

1. Inicializar Variables: Se reciben los valores de `m` y de `k`.
2. `B ← Factorial(m)`
3. `B ← B/Factorial(k)`
4. `B ← B/Factorial(m – k)`
5. Resultado: `B`

El flujograma de este nuevo algoritmo queda recogido en la Figura 6.3., donde de nuevo nos encontramos con varias llamadas a un mismo procedimiento: la que se emplea para calcular el valor factorial de un entero. Y es que, como ya se dijo en el capítulo anterior, un algoritmo debe estar compuesto por operaciones simples: el ordenador no sabe calcular el valor factorial de un entero y hay, por tanto, que definir también un algoritmo que calcule ese valor para un entero dado (cfr. Figura 5.9., en el capítulo anterior.).

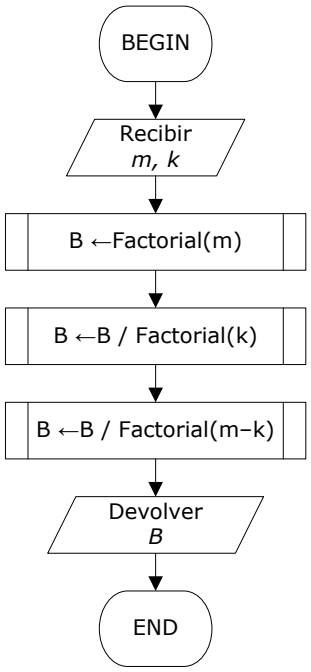


Figura 6.3. Algoritmo para cálculo del Binomio de Newton.

En el paso 5 de nuestro Algoritmo del Binomio obtenemos el valor del binomio de Newton. Qué cosa se vaya hacer con este valor es asunto que al procedimiento no le afecta. El procedimiento finaliza su cometido cuando logra obtener ese valor, sin hacer luego nada especial o concreto con él. En este caso es un cálculo intermedio necesario para lograr encontrar nuestra información verdaderamente buscada: la fila del Triángulo de Tartaglia indicada como entrada de todo el proceso que estamos afrontando.

Por lo mismo, tampoco es parte integrante del algoritmo del cálculo del Factorial presentado en el Capítulo anterior qué cosa se vaya a hacer con el resultado: si mostrarlo, u hacer con él otras operaciones, o lo que sea. En este caso, el procedimiento de cálculo de factorial facilita el valor del factorial del entero recibido al procedimiento que lo necesita, que en este caso es el procedimiento Binomio.

Con este ejemplo hemos visto que para afrontar y resolver satisfactoriamente un problema es conveniente descomponerlo en partes menores, independientes y más sencillas, que hacen el proceso más inteligible y fácil de resolver. Es a este proceso al que llamamos de creación de módulos, o modularización.

La apariencia del código para la resolución de este problema, en lenguaje C, es la que se muestra en Código 6.1. (no se preocupe si ahora le parece complicado: realmente no lo es tanto: ya lo aprenderá).

Código 6.1. Programa que muestra una fila del Triángulo de Tartagila.

```
#include <stdio.h>

/* Declaración de módulos o funciones definidas. -------- */
void Tartaglia(unsigned short);
unsigned long Binomio(unsigned short, unsigned short);
unsigned long Factorial(unsigned short);

/* Función Principal. ---------------------------------- */
int main(void)
{
      unsigned short fila;
      printf("Aplicación que muestra una fila");
      printf(" del triángulo de Tartaglia.\n\n");
      printf("Indique número de la fila a visualizar ... ");
      scanf(" %hu", &fila);

      Tartaglia(fila);
      return 0;
}
/* Función Tartaglia. ---------------------------------- */
void Tartaglia(unsigned short f)
{
      unsigned short i;

      for(i = 0 ; i < f ; i++)
            printf("%lu\t", Binomio(f - 1 , i ));
}
```

Código 6.1. (Cont.)

```
/* Función Binomio. ------------------------------------ */
unsigned long Binomio(unsigned short m, unsigned short k)
{
      return Factorial(m) / (Factorial(k)*Factorial(m - k));
}
/* Función Factorial. ---------------------------------- */
unsigned long Factorial(unsigned short n)
{
      unsigned long Fact = 1;

      while(n) Fact *= n--;
      return Fact;
}
```

Como puede comprobar en el código, la función `main` (función principal) se limita a recibir del usuario del programa el número de la fila que quiere visualizar, e invoca a la función `tartaglia`. A su vez, esta función simplemente invoca varias veces la función `binomio`: cada vez con un valor de `i` incrementado en uno, y hasta llegar a su valor máximo, con el valor de `i` igual a `f – 1`. Y a su vez la función `binomio` invoca por tres veces a la función `factorial`.

Otro ejemplo: supongamos que queremos hacer un programa que sirva para operar con números complejos. Queremos definir las operaciones de suma, de resta y de producto de dos complejos. Vamos a representar aquí un número complejo como un par de reales: `a + b·i = (a, b)` donde `a` y `b` son reales. Habrá que definir tres procedimientos para cada una de las tres operaciones. Cada uno de ellos recibe como datos de entrada dos números complejos (dos pares de números reales `a` y `b`); y ofrece como salida un número complejo.

La operación de cada uno de los tres módulos queda definida por los algoritmos representamos en los flujogramas de la Figura 6.4.

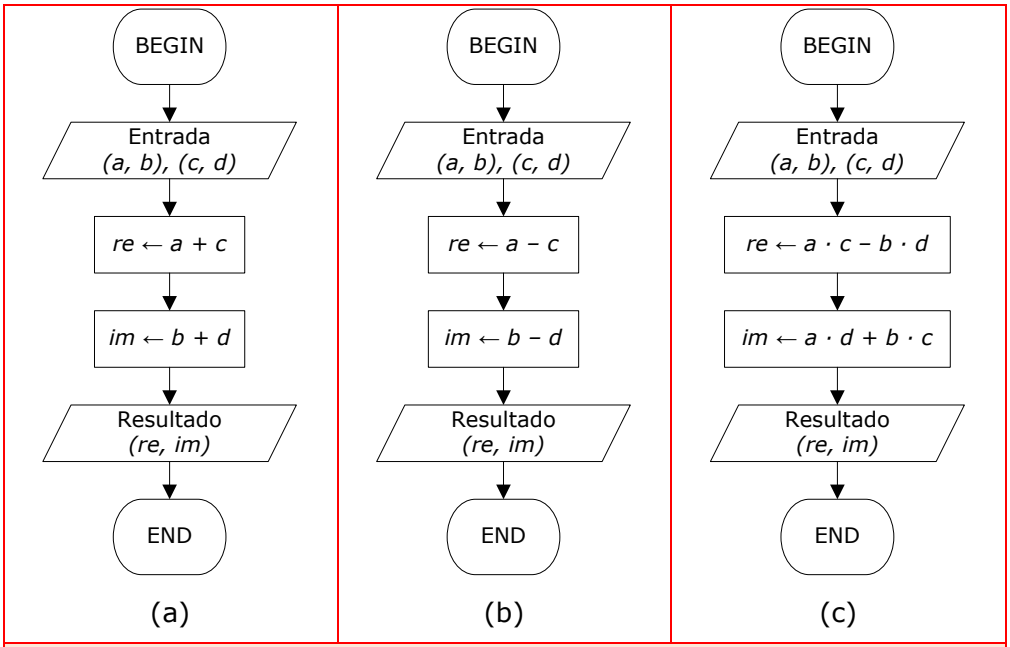


Figura 6.4. Operaciones con números complejos: (a) Suma; (b) Resta; (c) Producto.

El algoritmo de la aplicación (lo desarrollamos ahora en pseudocódigo) podría tener la siguiente definición. Suponemos que la aplicación recibe como entrada, además de los complejos a operar, un carácter que indica cuál es la operación que se desea realizar.

ENUNCIADO: Calculadora que realiza sumas, restas y productos de números complejos.

```
1.    Inicializar Variables: El usuario introduce:
      1.1.  v1 ←(a, b)
      1.2.  v2 ←(c, d)
      1.3.  op ← ( +|- | * )
2.    IF op = '+'
      THEN
            R ← Suma(v1, v2)
      ELSE
            IF op = '-'
            THEN
                  R ← Resta(v1, v2)
            ELSE
                  IF op = '*'
```

```
                THEN
                    R ← Producto(v1, v2)
                ELSE
                    Mensaje: "operador NO VÁLIDO"
                    R = "ERROR"
                END IF
            END IF
        END IF
```

3. Resultado: R

Si se solicita una operación no definida, el algoritmo lo adverte y no realizar nada.

Como ya hemos vislumbrado en ambos ejemplos, los módulos que se obtienen deben gozar de algunas propiedades, necesarias si se desea que la modularización resulte finalmente útil:

1. **Independencia funcional**. Cada módulo definido debe realizar una función concreta o un conjunto de funciones afines (alta cohesión), sin apenas ninguna relación con el resto (bajo acoplamiento).

   En el ejemplo de las operaciones con valores complejos, tenemos definidos tres módulos que en nada interfieren el uno con el otro.

   En el ejemplo del triángulo de Tartaglia tenemos dos niveles de independencia: el módulo `Binomio` necesita del módulo `Factorial`. Para que el módulo `Factorial` se ejecute correctamente no necesita de nada del resto del proceso definido excepto, claro está, de una valor de entrada. Y para que el módulo `Binomio` ejecute correctamente no necesita de nada del resto del proceso definido en su mismo nivel excepto, de nuevo, de dos valores de entrada.

2. **Comprensibilidad**. Es decir, cada módulo debe ser comprensible de forma aislada. Para lograr eso, desde luego se requiere la independencia funcional; y también establecer correctamente las relaciones entre cada módulo definido y los restantes.

   En nuestros dos ejemplos, cada uno de los módulos realiza una tarea concreta y bien definida. En el ejemplo del Triángulo de Tartaglia, se han implementado tres funciones útiles, cada una de las cuales

realiza una operación concreta y conocida: una construye la línea solicitada del triángulo; otra calcula el valor del binomio de Newton; y otra devuelve el valor del factorial de un entero recibido. Allá donde se recoja ese código, se podrán utilizar estas funciones

3. **Adaptabilidad**. Es decir, los módulos se deben definir de manera que permitan posteriores modificaciones o extensiones, realizadas tanto en ese módulo como en los módulos con los que se relaciona.

   En nuestro ejemplo del triángulo de Tartaglia, el módulo `Factorial`, una vez definido, ya es válido para todo proceso que necesite el cálculo de ese valor. De hecho ni siquiera ha sido necesario presentar este módulo en este capítulo porque ha bastado hacer referencia a la descripción que ya se hizo en el capítulo anterior: en nada importa el entorno concreto en el que queremos que se ejecute el algoritmo de cálculo del factorial.

4. Una advertencia importante para la correcta definición y construcción de un nuevo módulo: la correcta descripción del modo en que se utiliza: lo que podríamos llamar como una correcta definición de su **interfaz**. Las tres propiedades previas de la modularidad presentadas, exigen que cuando se defina un módulo o procedimiento quede perfectamente determinado el modo en que se hará uso de él: qué valores y en qué orden y forma espera recibir ese módulo como entrada para realizar correctamente el proceso descrito con su algoritmo. Y qué valores ofrece como resultado (y de nuevo en qué orden y forma) el módulo al finalizar las sentencias del algoritmo. Para que un módulo se considere debidamente definido debe ocurrir que, para cualquiera que desee usarlos, le sea suficiente conocer únicamente cuál es su interfaz. Cuál sea la definición del módulo, sus sentencias e instrucciones que ejecuta, y su orden, es cuestión que en nada deben importar al usuario de ese módulo.

   Allí donde se desee realizar una operación suma de complejos se podrá acudir al módulo definido con el algoritmo recogido en la

Figura 6.4. (a). Pero siempre que se haga uso de ese módulo será necesario que quien lo utilice facilite como entrada los dos pares de reales que definen los dos complejos, operandos de la operación suma que se desea realizar.

El proceso para la construcción de un programa pasa, por tanto, por estos pasos: abstracción → modularización → diseño de algoritmos → implementación. Entendemos por **implementación** el proceso de escribir, en un lenguaje de programación concreto, cada uno de los algoritmos diseñados.

La abstracción de la información: los datos.

En este proceso de abstracción de la realidad que hemos presentado, se ha seleccionado la información necesaria para resolver el problema planteado. Esta información viene recogida mediante unas entidades que llamamos datos. Entendemos por **dato** cualquier objeto manipulable por el ordenador: un carácter leído por el teclado, un valor numérico almacenado en disco o recibido a través de la red, etc. Estos datos pueden estar vinculados entre sí mediante un conjunto de relaciones.

Un dato puede ser tanto una **constante** definida dentro del programa y que no altera su valor durante su ejecución; o dato **variable**, que puede cambiar su valor a lo largo de la ejecución el programa.

El conjunto de valores posibles que puede tomar un dato variable se llama **rango** o **dominio**. Por ejemplo, podemos definir un rango que sea todos los enteros comprendidos entre 0 y $2^8 - 1 = 255$ (256 valores posibles). Los diferentes valores recogidos en un determinado rango se denominan literales. Un literal es la forma en que se codifica cada uno de los valores posibles de un rango o dominio determinado. Entendemos por **literal** cualquier símbolo que representa un valor. Por ejemplo, `3` representa el número 3 (literal numérico), y "`Este es un texto de siete palabras`" representa un texto (literal alfanumérico).

Los datos deberán ser codificados y ubicados en el programa mediante la reserva, para cada una de ellos, de un **espacio de memoria**. Los diferentes valores que pueden tomar esos datos variables quedarán codificados en los distintos estados que puede tomar esa memoria. El estado físico concreto que tome en un momento concreto un determinado espacio de memoria significará un valor u otro dependiendo de cuál sea el código empleado.

Cada espacio de memoria reservado para codificar y ubicar un dato variable (lo llamaremos simplemente variable) deberá ser identificado de forma inequívoca mediante un nombre. Para la generación de esos nombres hará falta echar mano de un alfabeto y de unas reglas de construcción (cada lenguaje tiene sus propias reglas). A estos nombres los llamamos identificadores. Los **identificadores** son símbolos empleados para representar objetos. Cada lenguaje debe tener definidas sus reglas de creación: en el capítulo 4, en el epígrafe "Elementos léxicos", están recogidas las reglas de creación de identificadores en el lenguaje C. Si un identificador representa un literal, es decir, si se define para un valor concreto, no variable, entonces el identificador representa un dato constante. Por ejemplo, se podría llamar PI a la constante que guarde el literal `3.14159`.

Tipo de dato.

Un **tipo de dato** define de forma explícita un conjunto de valores, denominado dominio (ya lo hemos definido antes), sobre el cual se pueden realizar un conjunto de operaciones.

Cada espacio de memoria reservado en un programa para almacenar un valor debe estar asociado a un tipo de dato. La principal motivación es la organización de nuestras ideas sobre los objetos que manipulamos.

Un lenguaje de programación proporciona un conjunto de tipos de datos simples o predefinidos (que se llaman los **tipos de dato primitivos**) y

además proporciona mecanismos para definir nuevos tipos de datos, llamados compuestos, combinando los anteriores.

Distintos valores pertenecientes a diferentes tipos de datos pueden tener la misma representación en la memoria. Por ejemplo, un byte con el estado 01000001 codificará el valor numérico 65 si este byte está empleado para almacenar valores de un tipo de dato entero; y codificará la letra 'A' si el tipo de dato es el de los caracteres y el código empleado es el ASCII.

Por eso, es muy importante, al reservar un espacio de memoria para almacenar valores concretos, indicar el tipo de dato para el que se ha reservado ese espacio.

Variable.

Una **variable** es un elemento o espacio de la memoria que sirve de almacenamiento de un valor, referenciada por un nombre, y perteneciente a un tipo de dato.

Podemos definir una variable como la cuádrupla

$$V = <N, T, R, K> \qquad (6.1.)$$

Donde N es el nombre de la variable (su identificador); T el tipo de dato para el que se creó esa variable (que le indica el dominio o rango de valores posibles); R es la referencia en memoria, es decir, la posición o dirección de la memoria reservada para esa variable (su ubicación); y K es el valor concreto que adopta esa variable en cada momento y que vendrá codificado mediante un estado físico de la memoria.

Por ejemplo, mediante <x, entero, Ref_x , 7> nos referimos a una variable que se ha llamado x, creada para reservar datos de tipo entero, que está posicionada en la dirección de memoria Ref_x (ya verá en su momento qué forma tienen esas referencias de memoria) y que en este

preciso instante tiene codificado el valor entero 7. Al hacer la asignación x ← 3x, se altera el estado de esa memoria, de forma que pase a codificar el valor entero 3.

En la mayoría de los lenguajes (y también en C) es preciso que toda variable se declare antes de su utilización. En la declaración se realiza la asociación o vinculación entre el tipo de dato (que explicita el dominio de valores) y el nombre que se le da a la variable. En algunos lenguajes es obligado, al declarar una variable, indicar su valor inicial; otros lenguajes asignan un valor por defecto en caso de que el programador no asigne ninguno; el lenguaje C no obliga a realizar esta operación de inicialización de valores en las variables declaradas: si no se le asigna un valor inicial, esas variables podrán tener cualquier valor de los definidos en el dominio (tipo de dato) en el que han sido declaradas. Ya verá como un error clásico de quien comienza a aprender a programar es descuidar esa operación de dar valor inicial a las variables.

En la declaración de la variable también queda definida la asociación entre el nombre y el espacio de memoria reservado: antes de la ejecución de un programa el ordenador ya conoce los requerimientos de espacio que ese programa lleva consigo: cantidad de bytes reservados y posición de la memoria del ordenador donde se ubica esa variable.

Variable - Tipo de dato – Valor.

Una definición, quizá un poco misteriosa, del concepto **valor** es: elemento perteneciente a un conjunto. ¿De qué conjunto estamos hablando?: del que queda explicitado mediante la declaración del tipo de dato. A este conjunto es al que hemos llamado rango o dominio.

Una vez introducidos estos tres conceptos (variable, tipo de dato, valor), es útil pensar en ellos conjuntamente.

Una variable ocupa un porción de memoria. Que esa porción sea más o menos extensa dependerá de para qué haya sido reservada esa

variable. Y eso lo determina el tipo de dato para el que se ha creado esa variable.

Qué valor codifica una variable en un momento determinado también depende del tipo de dato. Dos estados iguales pueden codificar valores distintos si se trata de espacios de memoria que codifican tipos de dato diferentes. Y qué valores puede codificar una determinada variable dependerá, de nuevo, del tipo de dato en el que ésta se haya definido.

Paradigmas de programación. Programación estructurada.

Un lenguaje de programación refleja cierto paradigma de programación. Un **paradigma** de programación es una colección de conceptos que guían el proceso de construcción de un programa y que determinan su estructura. Dichos conceptos controlan la forma en que se piensan y formulan los programas. Existen distintas formas de abordar un problema y darle solución mediante un programa informático; distintas enfoques que se pueden adoptar para resolver un problema de programación. A cada enfoque, a cada forma de actuar, se le llama paradigma.

La clasificación más común distingue tres tipos de paradigmas: imperativo, declarativo y orientado a objetos (que es una extensión del imperativo). Vamos a ceñirnos aquí a los lenguajes imperativos, que es el paradigma del lenguaje C.

Las principales **características** de los **lenguajes imperativos** son:

(1) el concepto de **variable** como medio para representar el estado del proceso computacional, y la instrucción de **asignación** como medio para cambiar el valor de una variable,

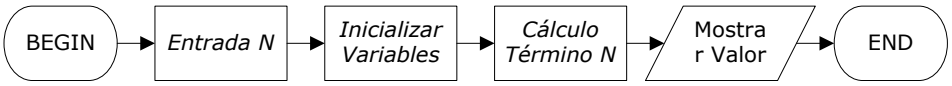
(2) las acciones y **funciones** como medio de descomposición de los programas, y

(3) las **instrucciones de control** que permiten establecer el orden en que se ejecutan las instrucciones.

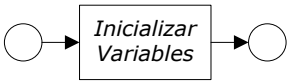
Dentro de estos lenguajes imperativos cabe distinguir a su vez los estructurados y los no estructurados. Los lenguajes estructurados incluyen estructuras de control que permiten determinar el orden de ejecución de las sentencias del algoritmo.

La programación estructurada establece las siguientes reglas:

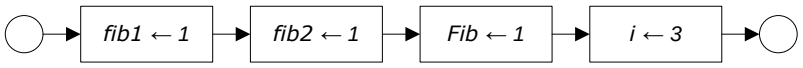
1. Todo programa consiste en una serie de acciones o **sentencias que se ejecutan en secuencia**, una detrás de otra. Si tomamos el ejemplo del capítulo anterior del cálculo del término N de la serie de Fibonacci, podemos esquematizar el proceso en estos pasos:



2. Cualquier acción puede ser sustituida por dos o más acciones en secuencia. Esta regla se conoce como la de **apilamiento**. Por ejemplo, el paso antes visto:

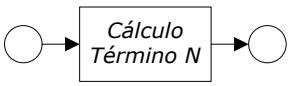


   puede representarse mejor como:

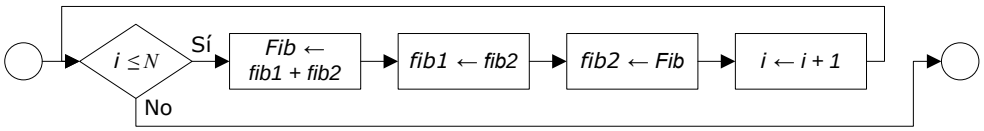


3. Cualquier acción puede ser sustituida por cualquier **estructura de control**; y sólo se consideran tres estructuras de control: la **secuencia** (ejecución secuencias de sentencias, una detrás de otra), la **selección** (o decisión) y la **repetición**. Esta regla se conoce como **regla de anidamiento**. Todas las estructuras de control de la programación estructurada tienen un solo punto de entrada y un solo

punto de salida. (Estas estructuras de control ya las hemos visto en el capítulo 5 del manual.) Por ejemplo, la sentencia:



puede representarse mejor como:



Las reglas de apilamiento y de anidamiento pueden aplicarse tantas veces como se desee y en cualquier orden.

El lenguaje C es un lenguaje imperativo y estructurado.

Recapitulación.

En este capítulo hemos presentado varios de conceptos muy necesarios para comprender el trabajo de la programación. La abstracción y la modularidad, como pasos previos e imprescindibles en todo trabajo de construcción de una aplicación: muchas horas de trabajo, sin teclados ni pantallas, previas a la selección o definición de algoritmos, y previas a la implementación a uno u otro lenguaje de programación.

Y el concepto de paradigma de programación, que determina distintos modos de trabajar y de formas de plantear soluciones. Los lenguajes de programación quedan clasificados en función de su paradigma. Para cada modo de plantear y resolver un problema existe un conjunto de lenguajes aptos para expresar el modelo abstraído y modularizado.

Ejemplo de lenguaje de paradigma imperativo y estructurado es el lenguaje C. El hecho de que el lenguaje C pertenezca a este paradigma implica un modo de trabajar y de programar. Ya ha quedado dicho en

este capítulo y en el anterior, pero no viene mal repetirlo: en lenguaje C no se deben emplear sentencias de salto: todo el control del flujo del programa debe organizarse mediante estructuras de control.

Ejemplos de lenguaje para programación orientada a objetos son los conocidos C++ y Java, entre otros. Aprender Java, por ejemplo, no es una tarea que se limite a conocer las reglas sintácticas de este lenguaje: requiere conocer a fondo el planteamiento que lleva consigo el paradigma de la programación orientada a objetos. Pretender programar en Java sin conocer su paradigma es como intentar que un mono hable.

CAPÍTULO 7

TIPOS DE DATO Y VARIABLES EN C.

Un **tipo de dato** define de forma explícita un conjunto de valores, denominado **dominio**, sobre el cual se pueden realizar una serie de operaciones. Un **valor** es un elemento del conjunto que hemos llamado dominio. Una **variable** es un espacio de la memoria destinada al almacenamiento de un valor de un tipo de dato concreto, referenciada por un nombre. Son conceptos sencillos, pero muy necesarios para saber exactamente qué se hace cuando se crea una variable en un programa.

Un tipo de dato puede ser tan complejo como se quiera. Puede necesitar un byte para almacenar cualquier valor de su dominio, o requerir de muchos bytes.

Cada lenguaje ofrece una colección de tipos de datos, que hemos llamado **primitivos**. También ofrece herramientas para crear tipos de dato distintos, más complejos que los primitivos y más acordes con el tipo de problema que se aborde en cada momento.

En este capítulo vamos a presentar los diferentes tipos de datos primitivos que ofrece el lenguaje C. Veremos cómo se crean y declaran las variables, qué operaciones se pueden realizar con cada una de ellas, y de qué manera se pueden relacionar unas variables con otras para formar expresiones. Veremos las limitaciones en el uso de las variables según su tipo de dato.

Ya hemos dicho que un tipo de dato especifica un dominio sobre el que una variable de ese tipo puede tomar sus valores; y unos operadores. A lo largo del capítulo iremos presentando los distintos operadores básicos asociados con los tipos de dato primitivos del lenguaje C. Es importante entender la operación que realiza cada operador y sobre qué dominio este operador está definido.

Declaración de variables.

Antes de ver los tipos de dato primitivos, conviene saber cómo se crea una variable en C.

Toda variable debe ser declarada previa a su uso. **Declarar una variable** es indicar al programa un **identificador** o nombre para esa variable, y el **tipo de dato** para la que se crea esa variable.

La declaración de variable tiene la siguiente sintaxis:

```
tipo var_1 [=valor1, var_2 = valor_2, ..., var_N = valor_N];
```

Donde `tipo` es el nombre del tipo de variable que se desea crear, y `var_1`, es el nombre o identificador de esa variable.

Aclaración a la notación: en las reglas sintácticas de un lenguaje de programación, es habitual colocar entre corchetes (`[]`) aquellas partes de la sintaxis que son optativas.

En este caso tenemos que en una declaración de variables se pueden declarar una o más variables del mismo tipo, todas ellas separadas por el **operador coma**. Al final de la sentencia de declaración de variables

se encuentra, como siempre ocurrirá en cualquier sentencia, el operador punto y coma.

En la declaración de una variable, es posible asignarle un valor de inicio. De lo contrario, la variable creada adquirirá un valor cualquiera entre todos los explicitados por el rango del tipo de dato, desconocido para el programador.

¿Qué ocurre si una variable no es inicializada? En ese caso, al declararla se dará orden de reservar un espacio de memoria (del tamaño que exija el tipo de dato indicado para la variable) para el almacenamiento de los valores que pueda ir tomando esa variable creada. Esa porción de memoria es un elemento físico y, como tal, deberá tener un estado físico. Cada uno de los bits de esta porción de memoria estará en el estado que se ha llamado 1, o en el estado que se ha llamado 0. Y un estado de memoria codifica una información concreta: la que corresponda al tipo de dato para el que está reservada esa memoria.

Es conveniente remarcar esta idea. No es necesario, y tampoco lo exige la sintaxis de C, dar valor inicial a una variable en el momento de su declaración. La casuística es siempre enorme, y se dan casos y circunstancias en las que realmente no sea conveniente asignar a la variable un valor inicial. Pero **habitualmente es muy recomendable inicializar las variables**. Otros lenguajes lo hacen por defecto en el momento de la declaración de variables; C no lo hace. Otros lenguajes detectan como error de compilación (errar sintáctico) el uso de una variable no inicializada; C acepta esta posibilidad.

A partir del momento en que se ha declarado esa variable, puede ya hacerse uso de ella. Tras la declaración ha quedado reservado un espacio de memoria para almacenar la información de esa variable.

Si declaramos `tipo variable = valor;` tendremos la variable `<variable, tipo, R, valor>`, de la que desconocemos su dirección de memoria. Cada vez que el programa trabaje con `variable` estará

haciendo referencia a esta posición de memoria R. Y estará refiriéndose a uno o más bytes, en función del tamaño del tipo de dato para el que se ha creado la variable.

Tipos de datos primitivos en C: sus dominios.

Los tipos de dato primitivos en C quedan recogidos en la Tabla 7.1.

Las variables de tipo de dato carácter ocupan 1 byte. Aunque están creadas para almacenar caracteres mediante una codificación como la ASCII (que asigna a cada carácter un valor numérico codificado con esos 8 bits), también pueden usarse como variables numéricas. En ese caso, el rango de valores es el recogido en la Tabla 7.1. En el caso de que se traten de variables con signo, entonces el rango va desde -2^7 hasta $+2^7-1$.

		RANGO DE VALORES	
SIGNO	TIPOS	MENOR	MAYOR
tipos de dato CARÁCTER: **char**			
signed	**char**	-128	+127
unsigned	**char**	0	+255
tipos de dato ENTERO: **int**			
signed	**short int**	-32.768	+32.767
unsigned	**short int**	0	+65.535
signed	**long int**	-2.147.483.648	+2.147.483.647
unsigned	**long int**	0	4.294.967.295
tipos de dato CON COMA FLOTANTE			
	float	-3.402923E+38	+3.402923E+38
	double	-1.7976931E+308	+1.7976931E+308
	long double	depende del compilador	

Tabla 7.1. Tipos de datos primitivos en C.

Para crear una variable de tipo carácter en C, utilizaremos la palabra clave **char**. Si la variable es con signo, entonces su tipo será **signed char**, y si no debe almacenar signo, entonces será **unsigned char**. Por defecto, si no se especifica si la variable es con o sin signo, el lenguaje C considera que se ha tomado la variable con signo, de forma que decir **char** es lo mismo que decir **signed char**.

Lo habitual será utilizar variables tipo **char** para el manejo de caracteres. Los caracteres simples del alfabeto latino se representan mediante este tipo de dato. El dominio de las variables **char** es un conjunto finito ordenado de caracteres, para el que se ha definido una correspondencia que asigna, a cada carácter del dominio, un código binario diferente de acuerdo con alguna normalización. El código más extendido es el código ASCII (American Standard Code for Information Interchange).

Las variables tipo entero, en C, se llaman **int**. Dependiendo de que esas variables sean de dos bytes o de cuatro bytes las llamaremos de tipo **short int** (16 bits) ó de tipo **long int** (32 bits). Y para cada una de ellas, se pueden crear con signo o sin signo: **signed short int** y **signed long int**, ó **unsigned short int** y **unsigned long int**. De nuevo, si no se especifica nada, C considera que la variable entera creada es con signo, de forma que la palabra **signed** vuelve a ser opcional. En general, se recomienda el uso de la palabra **signed**. Utilizar esa palabra al declarar enteros con signo facilita la compresión del código. Pero queda claro que no es necesario, en ningún caso, explicitarla en la declaración de una variable.

El tamaño de la variable **int** depende del concepto, no introducido hasta el momento, de **longitud de la palabra**. Habitualmente esta longitud se toma múltiplo de 8, que es el número de bits del byte. De hecho la longitud de la palabra viene definida por el máximo número de bits que puede manejar el procesador, de una sola vez, cuando hace cálculos con enteros.

Si se declara una variable en un PC como de tipo **int** (sin determinar si es **short** o **long**), el compilador de C considerará que esa variable es de la longitud de la palabra: de 16 ó de 32 bits. Es importante conocer ese dato (que depende del compilador), o a cambio es mejor especificar siempre en el programa si se desea una variable corta o larga, y no dejar esa decisión al tamaño de la palabra. Excepto en aquellos casos en que sea completamente indiferente cuál sea el tamaño de la variable entera declarada, se recomienda indicar siempre ese tamaño en el código del programa, al declarar esa variable.

Una variable declarada como de tipo **long** se entiende que es **long int**. Y una variable declarada como de tipo **short**, se entiende que es **short int**. Muchas veces se toma como tipo de dato únicamente el modificador de tamaño, omitiendo la palabra clave **int**.

Los restantes tipos de dato se definen para codificar valores reales. Hay que tener en cuenta que el conjunto de los reales es no numerable (entre dos reales siempre hay un real y, por tanto, hay infinitos reales). Los tipos de dato que los codifican ofrecen una codificación finita sí numerable. Esos tipos de dato codifican subconjuntos del conjunto de los reales; subconjuntos que, en ningún caso, pueden tomarse como un intervalo del conjunto de los reales.

A esta codificación de los reales se le llama de coma flotante. Así codifica el lenguaje C (y muchos lenguajes) los valores no enteros. Tomando como notación para escribir esos números la llamada notación científica (signo, mantisa, base, exponente: por ejemplo, el numero de Avogadro, $+6.023 \times 10^{23}$: signo positivo, mantisa 6,023, base decimal y exponente 23), almacena en memoria, de forma normalizada (**norma IEEE754**) el signo del número, su mantisa y su exponente. No es necesario codificar cuál es la base, porque se entiende que, en todos los casos, se trabaja con la base binaria.

Los tipos de dato primitivos en coma flotante que ofrece el lenguaje C son tres: **float**, que reserva 4 bytes para su codificación y que toma

valores en el rango señalado en la tabla 7.1.; **double**, que reserva 8 bytes; y **long double**, que reserva un número de bytes entre 8 y 16 (dependiendo del compilador y de otros factores). En estos tres tipos de dato el dominio abarca tantos valores positivos como negativos.

Existe además un tipo de dato que no reserva espacio en memoria: su tamaño es nulo. Es el tipo de dato **void**. No se pueden declarar variables de ese tipo. Más adelante se verá la necesidad y utilidad de tener definido un tipo de dato de estas características. Por ejemplo es muy conveniente para el uso de funciones.

En C el carácter que indica el fin de la parte entera y el comienzo de la parte decimal se escribe mediante el carácter punto. La sintaxis no acepta interpretaciones de semejanza, y para el compilador el carácter coma es un operador que nada tiene que ver con el punto decimal. Una equivocación en ese carácter causará habitualmente un error en tiempo de compilación.

Como ha podido comprobar, el lenguaje C dedica nueve palabras para la identificación de los tipos de dato primitivos: **void**, **char**, **int**, **float**, **double**, **short**, **long**, **signed** e **unsigned**. Ya se ha visto, por tanto más de un 25 % del léxico total del lenguaje C. Existen más palabras clave para la declaración y creación de variables, pero no introducen nuevos tipos de dato primitivos, sino que modifican el modo en que esas variables reservan sus espacios de memoria. Se verán más adelante.

Formación de literales

Cuando en una expresión deseamos introducir un valor literal (por ejemplo x = y + 2: aquí 2 es una valor literal) este valor puede interpretarse inicialmente de muchas formas: ¿es entero o es de coma flotante? ¿Viene expresado en base decimal, en base octal o en base hexadecimal? ¿Lo consideramos entero de 2 ó de 4 bytes; o de coma flotante de 4, ó de 8, ó de 10 bytes?

Para especificar sin equívoco un valor literal conviene conocer las siguientes reglas:

VALORES ENTEROS.

- Si queremos expresar el valor entero en base octal, deberemos comenzar su escritura con el dígito 0.
- Si queremos expresar el valor entero en base hexadecimal, deberemos comenzar su escritura con los caracteres `0x` (ó `0X`, poniendo la letra `'X'` que sigue al cero en mayúscula)
- Si queremos expresar el valor entero en base decimal, lo expresamos como es usual, pero no iniciando nunca ese valor con el dígito 0.
- Si queremos que el valor expresado se interprete como **unsigned**, entonces podemos terminar el literal con el sufijo `'u'`, ó `'U'`.
- Si queremos que el valor expresado se interprete como **long**, entonces podemos terminar el literal con el sufijo `'l'` ó `'L'`.
- Si queremos que el valor expresado se interprete como **unsigned long**, entonces podemos terminar el literal con los dos sufijos: `'u'` ó `'U'`, y `'l'` ó `'L'`.

En Código 7.1. se recogen algunos ejemplos de declaración de variables.

Código 7.1. Algunos ejemplos de declaración de variables en C.

```
short a = 071;    // Expresado en base octal.
short b = 0XABCD; // Expresado en base hexadecimal.
long c = 37l;     // Expresado en base decimal, indicando
                  // que el literal es de tipo long.
long d = 081;     // Error: va expresado en base octal.
                  // No puede tener el dígito 8.
```

Observaciones: Cuando se expresa el literal en base decimal, y deseamos que ese valor sea negativo, simplemente utilizamos el operador signo (`'-'`). Al expresar el literal en base hexadecimal o en base octal NO puede usarse ese operador de signo, porque en la misma expresión del literal se indica el signo: bit más significativo igual a 1: valor negativo; bit más significativo igual a 0: valor positivo. En Código 7.2. se muestran ejemplos de asignación de valores en hexadecimal.

Código 7.2. Algunos ejemplos de declaración de variables en C.

```
short a = 0x90E0;              // < 0: 16 bits:
                               // el + significativo es 1.
long b = 0x90E0;               // > 0: b vale 0x000090E0.
unsigned short c = 0x90E0;     // > 0: es variable unsigned.
long d = 0xABCD0000u;          // < 0: comienza con un 1.
```

En el último ejemplo de Código 7.2., al asignar a la variable d un valor que comienza con el dígito 1, y siendo d una variable con signo, la única interpretación posible es que su valor sea negativo: el literal indicado será **`unsigned`** realmente, pero el valor recogido no pertenece al dominio de los valores del tipo de dato **`signed long`**.

VALORES REALES DE COMA FLOTANTE.

Cuando deseamos expresar un valor literal de coma flotante utilizamos el carácter punto (`'.'`) para indicar la separación entre la parte entera y su parte fraccionaria. Y disponemos entonces de dos posibles sufijos para los tipos de dato de coma flotante: Si deseamos que el valor literal sea de tipo **`float`**, lo indicamos con el sufijo `'f'` ó `'F'`; si deseamos que el valor sea expresión de un **`long double`**, lo indicamos con el sufijo `'l'` ó `'L'`. Si no indicamos nada, se entiende que el valor es **`double`**.

Tipos de datos primitivos en C: sus operadores.

Ya se dijo que un tipo de dato explicita un conjunto de valores, llamado dominio, sobre el que son aplicables una serie de operadores. No queda, por tanto, del todo definido un tipo de dato presentando sólo su dominio. Falta indicar cuáles son los operadores que están definidos para cada tipo de dato.

Los operadores pueden aplicarse a una sola variable, a dos variables, e incluso a varias variables. Llamamos **operación unaria** a la que se aplica a una sola variable. Una operación unaria es, por ejemplo, el signo del valor.

No tratamos ahora las operaciones que se pueden aplicar sobre una variable creada para almacenar caracteres. Más adelante hay un capítulo entero dedicado a este tipo de dato **char**.

Las variables enteras, y las **char** cuando se emplean como variables enteras de pequeño rango, además del operador unario del signo, tienen definidos el operador asignación, los operadores aritméticos, los relacionales y lógicos y los operadores a nivel de bit.

Los operadores de las variables con coma flotante son el operador asignación, todos los aritméticos (excepto el operador módulo o cálculo del resto de una división), y los operadores relacionales y lógicos. Las variables **float**, **double** y **long double** no aceptan el uso de operadores a nivel de bit.

Operador asignación.

El operador asignación permite al programador modificar los valores de las variables y alterar, por tanto, el estado de la memoria del ordenador.

El carácter que representa al operador asignación es el carácter '='. La forma general de este operador es

```
nombre_variable = expresión;
```

Donde expresión puede ser un literal, otra variable, o una combinación de variables, literales y operadores y funciones. Podemos definirlo como una secuencia de operandos y operadores que unidos según ciertas reglas producen un resultado.

Este carácter '=', en C, no significa igualdad en el sentido matemático al que estamos acostumbrados, sino asignación. No debería llevar a equívocos expresiones como la siguiente:

```
a = a + 1;
```

Ante esta instrucción, el procesador toma el valor de la variable a, lo copia en un registro de la ALU donde se incrementa en una unidad, y almacena (asigna) el valor resultante en la variable a, modificando por ello el valor anterior de esa posición de memoria. La expresión arriba recogida no es una igualdad de las matemáticas, sino una orden para incrementar en uno el valor almacenado en la posición de memoria reservada por la variable a.

El operador asignación tiene dos extremos: el izquierdo (que toma el nombre Lvalue en muchos manuales) y el derecho (Rvalue). La apariencia del operador es, entonces:

```
LValue = RValue;
```

Donde Lvalue sólo puede ser el nombre de una variable, y nunca una expresión, ni un literal. Expresiones como a + b = c; ó 3 = variable; son erróneas, pues ni se puede cambiar el valor del literal 3 que, además, no está en memoria porque es un valor literal; ni se puede almacenar un valor en la expresión a + b, porque los valores se almacenan en variables, y a + b no es variable alguna, sino una expresión que recoge la suma de dos valores codificados en dos variables.

Un error de este estilo interrumpe la compilación del programa. El compilador dará un mensaje de error en el que hará referencia a que el Lvalue de la asignación no es correcto.

Cuando se trabaja con variables enteras, al asignar a una variable un valor mediante un literal (por ejemplo, `v = 3;`) se entiende que ese dato viene expresado en base 10.

Pero en C es posible asignar valores en la base hexadecimal. Si se quiere dar a una variable un valor en hexadecimal, entonces ese valor va precedido de un cero y una letra equis. Por ejemplo, si se escribe `v = 0x20`, se está asignando a la variable v el valor 20 en hexadecimal, es decir, el 32 en decimal, es decir, el valor 100000 en binario.

Operadores aritméticos.

Los operadores aritméticos son:

1. **Suma**. El identificador de este operador es el carácter `'+'`. Este operador es aplicable sobre cualquier par de variables o expresiones de tipo de dato primitivo de C. Si el operador `'+'` se emplea como operador unario, entonces es el operador de signo positivo.
2. **Resta**. El identificador de este operador es el carácter `'–'`. Este operador es aplicable sobre cualquier par de variables o expresiones de tipo de dato primitivo de C. Si el operador `'–'` se emplea como operador unario, entonces es el operador de signo negativo.
3. **Producto**. El identificador de este operador es el carácter `'*'`. Este operador es aplicable sobre cualquier par de variables o expresiones de tipo de dato primitivo de C: es un operador binario. Existe el operador `'*'` como operador unario, pero lo veremos más adelante, en el capítulo dedicado a los punteros.
4. **Cociente o División**. El identificador de este operador es el carácter `'/'`. Este operador es aplicable sobre cualquier par de variables o expresiones de tipo de dato primitivo de C.

   Cuando el cociente se realiza con variables enteras el resultado será también un entero. En los estándares C90 y anteriores, el valor que

devuelve este operador es el del mayor entero menor que el cociente. Por ejemplo, 5 dividido entre 2 es igual a 2. Y 3 dividido entre 4 es igual a 0. Es importante tener esto en cuenta cuando se trabaja con enteros.

Supongamos la expresión

```
superficie = (1 / 2) * base * altura;
```

para el cálculo de la superficie de un triángulo, y supongamos que todas las variables que intervienen han sido declaradas enteras. Así expresada la sentencia o instrucción de cálculo, el resultado será siempre el valor 0 para la variable `superficie`, sea cual sea el valor actual de las variable `base` o `altura`: y es que al realzar la ALU la operación 1 dividido entre 2, ofrece como resultado el valor 0.

Cuando el resultado del cociente es negativo, también se aplica, en los estándares C90 y previos, la regla del mayor entero menor que el resultado. Así, el cociente `(-1) / 2` ofrece como resultado el valor `(-1)`. Como verá en el siguiente epígrafe, esta forma de definir el cociente ha cambiado en los estándares posteriores.

Cuando el cociente se realiza entre variables de coma flotante, entonces el resultado es también de coma flotante.

Siempre se debe evitar el cociente en el que el denominador sea igual a cero, porque en ese caso se dará un error de ejecución y el programa quedará abortado.

5. **Módulo**. El identificador de este operador es el carácter `'%'`. Este operador calcula el resto del cociente entero. Por su misma definición, no tiene sentido su aplicación entre variables no enteras: su uso con variables de coma flotante provoca error de compilación. Como en el cociente, tampoco su divisor puede ser cero.
6. **Incremento y decremento**. Estos dos operadores no existen en otros lenguajes. El identificador de estos operadores son los

caracteres "++" para el incremento, y "--" para el decremento. Este operador es válido para todos los tipos de dato primitivos de C.

La expresión `a++;` es equivalente a la expresión `a = a + 1;`. Y la expresión `a--` es equivalente a la expresión `a = a - 1;`.

Estos operadores condensan, en uno sola expresión, un operador asignación, un operador suma (o resta) y un valor literal: el valor 1. Y como se puede apreciar son operadores unarios: se aplican a una sola variable.

Dónde se ubique el operador con respecto a la variable (operador delante de la variable, o prefijo; o detrás de la variable, o sufijo) tiene su importancia, porque varía su comportamiento dentro del total de la expresión.

Por ejemplo, el siguiente código

```
unsigned short int a, b = 2, c = 5;
a = b + c++;
```

modifica dos variables: por el operador asignación, la variable `a` tomará el valor resultante de sumar los contenidos de `b` y `c`; y por la operación incremento, que lleva consigo asociado otro operador asignación, se incrementa en uno el valor de la variable `c`.

Pero queda una cuestión abierta: ¿Qué operación se hace primero: incrementar `c` y luego calcular `b + c` para asignar su resultado a la variable `a` (a valdría en ese caso 8); o hacer primero la suma (en ese caso `a` quedaría con el valor 7) y sólo después incrementar la variable `c`? En ambos casos, queda claro que la variable `b` no modifica su valor y que la variable `c` pasa de valer 5 a valer 6.

Eso lo indicará la posición del operador. Si el operador incremento (o decremento) precede a la variable, entonces se ejecuta antes de evaluar el resto de la expresión; si se coloca después de la variable, entonces primero se evalúa la expresión donde está implicada la variable y sólo después se incrementa o decrementa esa variable.

En el ejemplo antes sugerido, el operador está ubicado a la derecha de la variable c. Por lo tanto, primero se efectúa la suma y la asignación sobre a, que pasa a valer 7; y luego se incrementa la variable c, que pasa a valer 6. La variable b no modifica su valor.

Por completar el ejemplo, si la expresión hubiera sido

```
a = b + ++c;
```

entonces, al final tendríamos que c vale 6 y que a vale 8, puesto que no se realizaría la suma y la asignación sobre a hasta después de haber incrementado el valor de la variable c.

Los operadores incremento y decremento, y el juego de la precedencia, son muy cómodos y se emplean mucho en los códigos escritos en lenguaje C.

Aunque hasta el tema siguiente no se va a ver el modo en que se pueden recibir datos desde el teclado (función `scanf`) y el modo de mostrar datos por pantalla (función `printf`), vamos a recoger a lo largo de este capítulo algunas cuestiones muy sencillas para resolver. Por ahora lo importante no es entender el programa entero, sino la parte que hace referencia a la declaración y uso de las variables.

Una consideración sobre el cociente de enteros.

El estándar C90 define el resultado de la división de dos enteros, en el supuesto de que ese cociente no sea exacto, como "el mayor entero menor que el cociente algebraico". Y esto, sea cual sea el signo de cada uno de los dos enteros del cociente: sea pues el resultado menor o mayor que cero.

El estándar C99 cambia esa definición; en este nuevo estándar, el resultado de la división de dos enteros, en el supuesto de que ese cociente no sea exacto, es el cociente algebraico al que se le ha

eliminado la parte fraccional: lo que se conoce habitualmente como "truncado hacia el cero". Y eso, de nuevo, independientemente de cuál sea el signo de la operación.

Cuando el resultado es positivo, entonces ambas definiciones coinciden. Pero cuando llegamos a un cociente no exacto de dos enteros de signos opuestos, los resultados de los cocientes C90 y C99 difieren en la unidad.

Por ejemplo: `short a = -3 , b = 2, c = a / b;`

¿Cuánto vale c? Si este programa se compila de acuerdo con el estándar C90, entonces el valor de la variable c es –2: el mayor entero menor que –1,5, que es el resultado algebraico del cociente, es –2. Si, en cambio, se compila y ejecuta de acuerdo con el estándar C99, entonces el valor de la variable c es –1: el truncado del resultado algebraico –1,5 es –1. ¿Por qué C99 introduce esta modificación de concepto?

Para explicarlo, considere las líneas de Código 7.3. ¿Cuál es el valor de c, ó de e, según el estándar C90?: el mayor entero menor que el resultado es –2: ése es el valor de esas variables. Y ¿cuál es el valor de h? Ahora primero se realiza el cociente, donde tenemos que el mayor entero menor que el resultado es el +1; luego, al cambiarle el signo, tendremos como resultado el valor –1. ¿No es curioso que (–3 / 2) sea distinto a –(3 / 2)? Eso es lo que resuelve la definición de cociente del estándar C99.

Código 7.3. Cociente de enteros: Distinto comportamiento C90 y C99.

```
signed short a = 3 , b = -2, c;
signed short d = -3 , e = 2 ; e;
signed short f = 3 , g = 2 ; h;
c = a / b;
e = d / e;
h = -(f / g);
```

La verdad es que cada una de las dos definiciones tiene una justificación inteligente. El estándar C99 no viene a corregir un error absurdo introducido en los estándares previos. Más adelante comprenderá por qué el C90 había propuesto su definición, y verá que con el cambio de definición se pierden propiedades computacionales que eran útiles con la definición del C90.

Cuando se programa hay que tener en mente estas pequeñas consideraciones sobre las definiciones de los operadores.

Operadores relacionales y lógicos.

Los operadores relacionales y los operadores lógicos crean expresiones que se evalúan como verdaderas o falsas.

En muchos lenguajes existe un tipo de dato primitivo para estos valores booleanos de verdadero o falso. En C ese tipo de dato no existe.

El lenguaje C toma como falsa cualquier expresión que se evalúe como 0. Y toma como verdadera cualquier otra evaluación de la expresión. Y cuando en C se evalúa una expresión con operadores relacionales y/o lógicos, la expresión queda evaluada a 0 si el resultado es falso; y a 1 si el resultado es verdadero.

Los operadores relacionales son seis: **igual que** ("=="), **distintos** ("!=") , **mayor que** (">"), **mayor o igual que** (">="), **menor que** ("<") y **menor o igual que** ("<=").

Todos ellos se pueden aplicar a cualquier tipo de dato primitivo de C.

Una expresión con operadores relacionales sería, por ejemplo, `a != 0`, que será verdadero si a toma cualquier valor diferente al 0, y será falso si a toma el valor 0. Otras expresiones relacionales serían, por ejemplo, `a > b + 2;` ó `x + y == z + t;`

Con frecuencia interesará evaluar una expresión en la que se obtenga verdadero o falso no sólo en función de una relación, sino de varias. Por

ejemplo, se podría necesitar saber (obtener verdadero o falso) si el valor de una variable concreta está entre dos límites superior e inferior. Para eso necesitamos concatenar dos relacionales. Y eso se logra mediante los operadores lógicos.

Un error frecuente (y de graves consecuencias en la ejecución del programa) al programar en C ó C++ es escribir el operador asignación ('='), cuando lo que se pretendía escribir era el operador relacional "igual que" ("=="). En los lenguajes C ó C++ la expresión `variable = valor;` será siempre verdadera si `valor` es distinto de cero. Si colocamos una asignación donde deseábamos poner el operador relacional "igual que", tendremos dos consecuencias graves: se cambiará el valor de la variable colocada a la izquierda del operador asignación (cosa que no queríamos) y, si el valor de la variable de la derecha es distinto de cero, la expresión se evaluará como verdadera al margen de cuáles fueran los valores iniciales de las variables.

Los operadores lógicos son: **AND**, cuyo identificador está formado por el carácter repetido "&&"; **OR**, con el identificador "||"; y el operador **negación**, cuyo identificador es el carácter de admiración final ('!').

Estos operadores binarios actúan sobre dos expresiones que serán verdaderas (o distintas de cero), o falsas (o iguales a cero), y devuelven como valor 1 ó 0 dependiendo de que la evaluación haya resultado verdadera o falsa.

La Tabla 7.2. recoge el resultado de los tres operadores en función del valor de cada una de las dos expresiones que evalúan.

Por ejemplo, supongamos el siguiente código en C:

```
int a = 1 , b = 3 , x = 30 , y = 10;
int resultado;
resultado = a * x == b * y;
```

El valor de la variable `resultado` quedará igual a 1.

a	b	a && b	a || b	!a
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Tabla 7.2. Resultados de los operadores lógicos.

Y si queremos saber si la variable x guarda un valor entero positivo menor que cien, escribiremos la expresión

```
(x > 0 && x < 100)
```

Con estos dos grupos de operadores son muchas las expresiones de evaluación que se pueden generar. Quizá en este momento no adquiera mucho sentido ser capaz de expresar algo así; pero más adelante se verá cómo la posibilidad de verificar sobre la veracidad y falsedad de muchas expresiones permite crear estructuras de control condicionales, o de repetición.

Una expresión con operadores relacionales y lógicos admite varias formas equivalentes Por ejemplo, la antes escrita sobre el intervalo de situación del valor de la variable x es equivalente a escribir `!(x < 0 || x >= 100)`

- ***Evalúe las expresiones recogidas en Código 7.4.***

  La última expresión recogida en Código 7.4. trae su trampa: Por su estructura se ve que se ha pretendido crear una expresión lógica formada por dos sencillas enlazadas por el operador OR. Pero al establecer que uno de los extremos de la condición es `a = 10` (asignación, y no operador relacional "igual que") se tiene que en esta expresión recogida la variable a pasa a valer `10` y la expresión es verdadera puesto que el valor `10` es verdadero (todo valor distinto de cero es verdadero).

Código 7.4. Ejemplos de expresiones lógicas.

```
short a = 0, b = 1, c = 5;

a;                      // FALSO
b;                      // VERDADERO
a < b;                  // VERDADERO
5 * (a + b) == c;       // VERDADERO

float pi = 3.141596;
long x = 0, y = 100, z =1234;

3 * pi < y && (x + y) * 10 <= z / 2;           // FALSO
3 * pi < y || (x + y) * 10 <= z / 2;           // VERDADERO
3 * pi < y && !((x + y) * 10 <= z / 2);        // VERDADERO

long a = 5, b = 25, c = 125, d = 625;

5 * a == b;             // VERDADERO
5 * b == c;             // VERDADERO
a + b + c + d < 1000;   // VERDADERO
a > b || a = 10;        // VERDADERO: operador asignación!
```

Operadores a nivel de bit.

El lenguaje C tiene la capacidad de trabajar a muy bajo nivel, modificando un bit de una variable, o logrando códigos que manipulan la codificación interna de la información. Eso se logra gracias a los operadores de nivel de bit.

Todos los operadores a nivel de bit están definidos únicamente sobre variables de tipo entero. No se puede aplicar sobre una variable **`float`**, ni sobre una **`double`**, ni sobre una **`long double`**. Si se utilizan con variables de esos tipos, se produce un error de compilación.

Los operadores a nivel de bit son seis:

1. Operador **AND a nivel de bit**. Su identificador es un solo carácter `'&'`. Se aplica sobre variables del mismo tipo, con la misma longitud

de bits. Bit a bit compara los dos de cada misma posición y asigna al resultado un 1 en su bit de esa posición si los dos bits de las dos variables sobre las que se opera valen 1; en otro caso asigna a esa posición del bit el valor 0.

2. Operador **OR a nivel de bit**. Su identificador es un solo carácter '|'. Se aplica sobre variables del mismo tipo, con la misma longitud de bits. Bit a bit compara los dos de cada misma posición y asigna al resultado un 1 en su bit de esa posición si alguno de los dos bits de las dos variables sobre las que se opera valen 1; si ambos bits valen cero, asigna a esa posición del bit el valor 0.

   Es frecuente en C y C++ el error de pretender escribir el operador lógico "and" ("&&"), o el "or" ("||") y escribir finalmente un operador a nivel de bit ('&' ó '|'). Desde luego el significado de la sentencia no será el deseado. Será un error de difícil detección.

3. Operador **OR EXCLUSIVO, ó XOR a nivel de bit**. Su identificador es un carácter '^'. Se aplica sobre variables del mismo tipo, con la misma longitud de bits. Bit a bit compara los dos de cada misma posición y asigna al resultado un 1 en ese bit en esa posición si los dos bits de las dos variables tienen valores distintos: el uno es 1 y el otro 0, o viceversa; si los dos bits son iguales (dos unos o dos ceros), asigna a esa posición del bit el valor 0.

Por ejemplo, y antes de seguir con los otros tres operadores a nivel de bit, supongamos que tenemos el código recogido en Código 7.5.

Código 7.5. Expresiones con operadores a nivel de bit.

```
unsigned short int a = 0xABCD, b = 0x6789;
unsigned short int a_and_b = a & b;
unsigned short int a_or_b = a | b;
unsigned short int a_xor_b = a ^ b;
```

La variable a vale, en hexadecimal ABCD, y en decimal 43981. La variable b 6789, que en base diez es 26505. Para comprender el comportamiento de estos tres operadores, se muestra ahora en la Tabla 7.3. los valores de a y de b en base dos, donde se puede ver bit a bit de ambas variables, y veremos también el bit a bit de las tres variables calculadas.

variable	binario	hex.	dec.
a	1010 1011 1100 1101	ABCD	43981
b	0110 0111 1000 1001	6789	26505
a_and_b	0010 0011 1000 1001	2389	9097
a_or_b	1110 1111 1100 1101	EFCD	61389
a_xor_b	1100 1100 0100 0100	CC44	52292

Tabla 7.3. Valores de las variables de Código 7.5.

En la variable a_and_b se tiene un 1 en aquellas posiciones de bit donde había 1 en a y en b; un 0 en otro caso. En la variable a_or_b se tiene un 1 en aquellas posiciones de bit donde había al menos un 1 entre a y b; un 0 en otro caso. En la variable a_xor_b se tiene un 1 en aquellas posiciones de bit donde había un 1 en a y un 0 en b, o un 0 en a y un 1 en b; y un cero cuando ambos bits coincidían de valor en esa posición.

La tabla de valores de estos tres operadores queda recogida en la Tabla 7.4.

4. Operador **complemento a uno**. Este operador unario devuelve el complemento a uno del valor de la variable a la que se aplica (cfr. Capítulo 2: recuerde que en la base binaria, el complemento a uno de un número se obtiene sin más que cambiar los ceros por unos, y los unos por ceros.). Su identificador es el carácter '~'. Si se tiene

que a la variable x de tipo **short** se le asigna el valor hexadecimal ABCD, entonces la variable y, a la que se asigna el valor ~x valdrá 5432. O si x vale 578D, entonces y valdrá A872. Puede verificar estos resultados calculando los complementos a uno de ambos números.

		and	or	xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabla 7.4. Valores que adoptan los tres operadores a nivel de bit

5. Operador **desplazamiento a izquierda**. Su identificador es la cadena "<<". Es un operador binario, que realiza un desplazamiento de todos los bits de la variable o valor literal sobre la que se aplica un número dado de posiciones hacia la izquierda. Los bits más a la izquierda (los más significativos) se pierden; a la derecha se van introduciendo tantos bits puestos a cero como indique el desplazamiento.

   Por ejemplo, si tenemos el siguiente código:

```
short int var1 = 0x7654;
short int var2 = var1 << 3;
```

   La variable var2 tendrá el valor de la variable var1 a la que se le aplica un desplazamiento a izquierda de 3 bits.

   Si la variable var1 tiene el valor, en base binaria

```
0111  0110  0101  0100  (estado de la variable var1)
 (7)   (6)   (5)   (4)
```

entonces la variable var2, a la que se asigna el valor de la variable var1 al que se han añadido tres ceros a su derecha y se le han eliminado los tres dígitos más a la izquierda queda:

```
1011  0010  1010  0000  (estado de la variable var2)
 (B)   (2)   (A)   (0)
```

Es decir, var2 valdrá, en hexadecimal, B2A0.

Una observación sobre esta operación. Introducir un cero a la derecha de un número es lo mismo que multiplicarlo por la base.

En el siguiente código

```
unsigned short int var1 = 12;
unsigned short int d = 1;
unsigned short int var2 = var1 << d;
```

var2 será el doble que var1, es decir, 24. Y si d hubiera sido igual a 2, entonces var2 sería cuatro veces var1, es decir, 48. Y si d hubiera sido igual a 3, entonces var2 sería ocho veces var1, es decir, 96.

Si llega un momento en que el desplazamiento obliga a perder algún dígito 1 a la izquierda, entonces ya habremos perdido esa progresión, porque la memoria no será suficiente para albergar todos sus dígitos y la cifra será truncada.

Si las variables var1 y var2 están declaradas como **signed**, y si la variable var1 tiene asignado un valor negativo (por ejemplo, –7), también se cumple que el desplazamiento equivalga a multiplicar por dos. Es buen ejercicio de codificación de enteros con signo verificar los datos que a continuación se presentan:

var1 = –7;: estado de memoria FFF9

var2 = var1 << 1;: estado de memoria para la variable var2 será FFF2, que es la codificación del entero –14.

6. Operador **desplazamiento a derecha**. Su identificador es la cadena ">>". Es un operador binario, que realiza un desplazamiento de

todos los bits de la variable o valor literal sobre la que se aplica un número dado de posiciones hacia la derecha. Los bits más a la derecha (los menos significativos) se pierden; a la izquierda se van introduciendo tantos bits como indique el desplazamiento. En esta ocasión, el valor de los bits introducidos por la izquierda dependerá del signo del entero sobre el que se aplica el operador desplazamiento. Si el entero es positivo, se introducen tantos ceros por la izquierda como indique el desplazamiento. Si el entero es negativo, se introducen tantos unos por la izquierda como indique el desplazamiento. Evidentemente, si el entero sobre el que se aplica el desplazamiento a derecha está declarado como **unsigned**, únicamente serán ceros lo que se introduzca por su izquierda, puesto que en ningún caso puede codificar un valor negativo.

Si desplazar a la izquierda era equivalente a multiplicar por la base, ahora, desplazar a la derecha es equivalente a dividir por la base. Aquí hay que recordar las dos definiciones de cociente propuestas por los estándares C90 y C99. En ambas, si el entero sobre el que se realiza el desplazamiento no es negativo, el comportamiento es, efectivamente, el de dividir por 2. Pero cuando el entero sobre el que se realiza el desplazamiento es negativo (comienza por 1), entonces el resultado obtenido coincide con la definición de cociente del estándar C90, pero no con la del estándar C99.

Por ejemplo, si tenemos el siguiente código

```
signed short int var1 = -231;
signed short int var2 = var1 >> 1;
```

Entonces, el estado que codifica el valor de `var1` es, expresado en hexadecimal, FF19. Y el valor que codifica entonces `var2`, si lo hemos desplazado un bit a la derecha, será FF8C, que es la codificación del entero negativo –116, que coincide con el resultado de dividir –231 entre 2 en C90, pero no en C99, cuyo valor final de ese cociente será –115. La definición de cociente propuesta por el

estándar C90 permitía realizar cocientes con denominador potencia de 2 de una manera muy ágil desde un punto de vista computacional.

```
1111  1111  0001  1001
 (F)   (F)   (1)   (9)   (estado de la variable var1)
1111  1111  1000  1100
 (F)   (F)   (8)   (C)   (estado de la variable var2)
```

Los operadores a nivel de bit tienen una gran potencialidad. De todas formas no son operaciones a las que se está normalmente habituado, y eso hace que no resulte muchas veces evidente su uso. Los operadores a nivel de bit operan a mucha mayor velocidad que, por ejemplo, el operador producto o cociente. En la medida en que se sabe, quien trabaja haciendo uso de esos operadores puede lograr programas notablemente más veloces en ejecución.

Operadores compuestos.

Ya se ha visto que una expresión de asignación en C trae, a su izquierda (Lvalue), el nombre de una variable y, a su derecha (Rvalue) una expresión a evaluar, o un literal, o el nombre de otra variable. Y ocurre frecuentemente que la variable situada a la izquierda forma parte de la expresión de la derecha. En estos casos, y si la expresión es sencilla, todos los operadores aritméticos y los operadores a nivel de bit binarios (exceptuando, por tanto, los operadores de signo, incremento, decremento y complemento) pueden presentar otra forma, en la que se asocia el operador con el operador asignación. Son los llamados operadores de asignación compuestos:

```
+=      x += y;     es lo mismo que decir x = x + y;
-=      x -= y;     es lo mismo que decir x = x - y;
*=      x *= y;     es lo mismo que decir x = x * y;
/=      x /= y;     es lo mismo que decir x = x / y;
%=      x %= y;     es lo mismo que decir x = x % y;
>>=     x >>= y;    es lo mismo que decir x = x >> y;
<<=     x <<= y;    es lo mismo que decir x = x << y;
&=      x &= y;     es lo mismo que decir x = x & y;
```

```
|=        x |= y;       es lo mismo que decir x = x | y;
^=        x ^= y;       es lo mismo que decir x = x ^ y;
```

Tenga cuidado al usar esos operadores. Si lo que usted quiere expresar es que `a = a – b;` entonces podrá decir `a –= b;`. Pero si lo que quiere expresar es `a = b – a;` entonces no debe, en ese caso, utilizar el operador compuesto. Tenga en cuenta que las operaciones producto y suma son conmutativas, pero no lo son, en cambio, ni el cociente ni el resto, ni la resta.

Puede parecer que estos operadores no facilitan la comprensión del código escrito. Quizá una expresión de la forma `F*=n––;` no tenga una presentación muy clara. Pero de hecho estos operadores compuestos se usan frecuentemente y, quien se habitúa a ellos, agradece que se hayan definido para el lenguaje C.

Por cierto, que la expresión del párrafo anterior es equivalente a escribir estas dos líneas de código: `F = F * n;` y `n = n – 1;`.

Intercambio de valores de dos variables.

Una operación bastante habitual en un programa es el **intercambio de valores** entre dos variables. Supongamos el siguiente ejemplo:

<variable1, **int**, R_1, 10> y <variable2, **int**, R_2, 20>

Si queremos que `variable1` almacene el valor de `variable2` y que `variable2` almacene el de `variable1`, es necesario acudir a una variable `auxiliar`. El proceso se recoge en Código 7.6.

Porque no podemos copiar el valor de `variable2` en `variable1` sin perder con esta asignación el valor de `variable1` que queríamos guardar en `variable2`.

Con el operador or exclusivo se puede hacer intercambio de valores sin acudir, para ello, a una variable `auxiliar`. El procedimiento es el recogido en la segunda ventana de Código 7.6.

Código 7.6. Intercambio de valor entre dos variables.

```
auxiliar  = variable1;
variable1 = variable2;
variable2 = auxiliar;
```

```
variable1 ^= variable2;
variable2 ^= variable1;
variable1 ^= variable2;
```

Veamos un ejemplo para comprender cómo realiza el intercambio:

```
short int variable1 = 3579;
short int variable2 = 2468;
```

en base binaria la evolución del valor de las dos variables es:

```
               variable1  0 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1
               variable2  0 0 0 0 1 0 0 1 1 0 1 0 0 1 0 0
  variable1 ^= variable2  0 0 0 0 0 1 0 0 0 1 0 1 1 1 1 1
  variable2 ^= variable1  0 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1
  variable1 ^= variable2  0 0 0 0 1 0 0 1 1 0 1 0 0 1 0 0
```

Al final del proceso, el valor de `variable1` es el que inicialmente tenía `variable2`, y al revés. Basta comparar valores. Para verificar que las operaciones están correctas (que lo están) hay que tener en cuenta que el proceso va cambiando los valores de `variable1` y de `variable2`, y esos cambios hay que tenerlos en cuenta en las siguientes operaciones or exclusivo a nivel de bit.

Operador **`sizeof`**.

Ya sabemos el número de bytes que ocupan en memoria todas las variables de tipo de dato primitivo en C: 1 byte las variables tipo **`char`**; 2 las de tipo **`short`**; 4 las de tipo **`long`** y **`float`**; 8 las de tipo **`double`**, y entre 8 y 15 las variables **`long double`**.

Pero ya se ha dicho que además de estos tipos primitivos, C permite la definición de otros tipos diferentes, combinación de esos primitivos. Y los tamaños de esos tipos definidos pueden ser tan diversos como diversas pueden ser las definiciones de esos nuevos tipos. No es extraño trabajar con tipos de 13 bytes, ó 1045, ó cualquier otro tamaño.

C ofrece un operador que devuelve la cantidad de bytes que ocupa una variable o un tipo de dato concreto. El valor devuelto es tomado como un entero, y puede estar presente en cualquier expresión de C. Es el operador **sizeof**. Su sintaxis es:

sizeof(nombre_variable); ó **sizeof**(nombre_tipo_de_dato);

ya que se puede utilizar tanto con una variable concreta como indicándole al operador el nombre del tipo de dato sobre el que queramos conocer su tamaño. No es válido utilizar este operador indicando entre paréntesis el tipo de dato **void** (más adelante conocerá este tipo de dato): esa instrucción daría error en tiempo de compilación.

Código 7.7. Para ver el tamaño de los tipos de dato primitivos.

```
#include <stdio.h>
main()      {
      printf("int           => %d\n",sizeof(int));
      printf("char          => %d\n",sizeof(char));
      printf("short         => %d\n",sizeof(short));
      printf("long          => %d\n",sizeof(long));
      printf("float         => %d\n",sizeof(float));
      printf("double        => %d\n",sizeof(double));
}
//La función printf se presenta en el próximo capítulo.
```

Con este operador aseguramos la portabilidad, al no depender la aplicación del tamaño del tipo de datos de la máquina que se vaya a

usar. Aunque ahora mismo no se ha visto en este texto qué utilidad puede tener en un programa conocer, como dato de cálculo, el número de bytes que ocupa una variable, la verdad es que con frecuencia ese dato es muy necesario. Podemos ver el tamaño de los diferentes tipos de datos primitivos de C. Basta teclear en nuestro editor las líneas recogidas en Código 7.7.

Expresiones en las que intervienen variables de diferente tipo.

Hay lenguajes de programación que no permiten realizar operaciones con valores de tipos de dato distintos. Se dice que son lenguajes de *tipado* fuerte, que fuerzan la comprobación de la coherencia de tipos en todas las expresiones, y lo verifican en tiempo de compilación.

El lenguaje C NO es de esos lenguajes, y permite la compilación de un programa con expresiones que mezclan los tipos de datos.

Y aunque en C se pueden crear expresiones en las que intervengan variables y literales de diferente tipo de dato, el procesador trabaja de forma que todas las operaciones que se realizan en la ALU sean con valores del mismo dominio.

Para lograr eso, cuando se mezclan en una expresión diferentes tipos de dato, el compilador convierte todas las variables a un único tipo compatible; y sólo después de haber hecho la conversión se realiza la operación.

Esta conversión se realiza de forma que no se pierda información: en una expresión con elementos de diferentes dominios, todos los valores se recodifican de acuerdo con el tipo de dato de mayor rango.

La ordenación de rango de los tipos de dato primitivos de C es, de menor a mayor, la siguiente:

char – short – long – float – double – long double

Así, por ejemplo, si se presenta el siguiente código:

```
char ch = 7;
short sh = 2;
long ln = 100, ln2;
double x = 12.4, y;
y = (ch * ln) / sh – x;
```

La expresión para el cálculo del valor que se almacenará finalmente en la variable y va cambiando de tipo de dato a medida que se realizan las operaciones y se introducen nuevas variables de nuevos tipos de dato: en el producto de la variable **char** con la variable **long**, se fuerza el cambio de la variable de tipo **char**, que se recodificará y así quedará para su uso en la ALU, a tipo **long**. Esa suma será por tanto un valor **long**. Luego se realizará el cociente con la variable **short**, que deberá convertirse en **long** para poder dividir al resultado **long** antes obtenido. Y, finalmente, el resultado del cociente se debe convertir en un valor **double**, para poder restarle el valor de la variable x.

El resultado final será pues un valor del tipo de dato **double**. Y así será almacenado en la posición de memoria de la variable y de tipo **double**.

Si la última instrucción hubiese sido `ln2 = (ch * ln) / sh – x;` todo hubiera sido como se ha explicado, pero a la hora de almacenar la información en la memoria reservada para la variable **long** ln2, el resultado final, que venía expresado en formato **double**, deberá recodificarse para ser guardado como **long**. Y es que, en el trasiego de la memoria a los registros de la ALU, bien se puede hacer un cambio de tipo y por tanto un cambio de forma de codificación y, especialmente, de número de bytes empleados para esa codificación. Pero lo que no se puede hacer es que en una posición de memoria como la del ejemplo, que dedica 32 bits a almacenar información, se quiera almacenar un valor de 64 bits, que es lo que ocupan las variables **double**.

Ante el operador asignación, si la expresión evaluada, situada en la parte derecha de la asignación, es de un tipo de dato diferente al tipo de dato de la variable indicada a la izquierda de la asignación, entonces el

valor del lado derecho de la asignación se convierte al tipo de dato del lado izquierdo. En este caso el forzado de tipo de dato puede consistir en llevar un valor a un tipo de dato de menor rango. Y ese cambio corre el riesgo de perder —truncar— la información.

Operador para forzar cambio de tipo.

En el epígrafe anterior se ha visto el cambio o conversión de tipo de dato que se realiza de forma implícita en el procesador cuando encuentra expresiones que contienen diferentes tipos de dato. También existe una forma en que programador puede forzar un cambio de tipo de forma explícita. Este cambio se llama cambio por promoción, o ***casting***. C dispone de un operador para forzar esos cambios.

La sintaxis de este operador unario es la siguiente:

```
(tipo) nombre_variable;
```

El operador de promoción de tipo, o *casting*, precede a la variable. Se escribe entre paréntesis el nombre del tipo de dato hacia donde se desea forzar el valor de la variable sobre la que se aplica el operador.

La operación de conversión debe utilizarse con cautelas, de forma que los cambios de tipo sean posibles y compatibles. No se puede realizar cualquier cambio; especialmente cuando se trabaja con tipos de dato creados (no primitivos). También hay que estar vigilante a los cambios de tipo que fuerzan a una disminución en el rango: por ejemplo, forzar a que una variable **`float`** pase a ser de tipo **`long`**. El rango de valores de una variable **`float`** es mucho mayor, y si el valor de la variable es mayor que el valor máximo del dominio de los enteros de 4 bytes, entonces el resultado del operador forzar tipo será imprevisible. Y tendremos entonces una operación que no ofrece problema alguno en tiempo de compilación, pero que bien puede llevar a resultados equivocados en tiempo de ejecución.

Desde luego, el operador forzar tipo no altera en nada la información en memoria. Las variables serán siempre del tipo de dato con el que fueron creadas. Y este operador no cambia el valor de esas variables. Lo que sí hace el operador es cambiar el tipo en la expresión concreta donde se aplica. Por ejemplo, suponga el siguiente código:

```
short a = 1, b = 2;
double c = (double)a / b;
```

Aquí el operador forzar tipo está bien empleado: al calcular el valor para la variable c se ha querido que se realice el cociente con decimales. Para ello se ha forzado el tipo de la variable a y, así, se ha logrado que también promocione la variable b y que el resultado final sea un **double** (0.5). Pero en ningún momento el valor de la variable a ha dejado de estar codificado en formato entero de 2 bytes: porque la variable a ocupa 2 bytes, y no podrá ocupar 8 por más que se le apliquen operadores. Aquí lo que ha cambiado es el modo en que, en esta expresión y sólo en esta expresión, el valor de la variable a se ha codificado en la ALU

No se pueden realizar conversiones del tipo **void** a cualquier otro tipo, pero sí de cualquier otro tipo al tipo **void**. Eso se entenderá mejor más adelante.

Propiedades de los operadores.

Al evaluar una expresión formada por diferentes variables y literales, y por diversos operadores, hay que lograr expresar realmente lo que se desea operar. Por ejemplo, la expresión a + b * c… ¿Se evalúa como el producto de la suma de a y b, con c; o se evalúa como la suma del producto de b con c, y a?

Para definir unas reglas que permitan una interpretación única e inequívoca de cualquier expresión, se han definido tres propiedades en los operadores:

1. Su **posición**. Establece dónde se coloca el operador con respecto a sus operandos. Un operador se llamará **infijo** si viene a colocarse entre sus operandos; y se llamará **prefijo** o **postfijo** si el operador precede al operando o si le sigue. Operador infijo es, por ejemplo, el operador suma; operador prefijo es, por ejemplo, el operador casting; operador postfijo es, por ejemplo, el operador incremento.
2. Su **precedencia**. Establece el orden en que se ejecutan los distintos operadores implicados en una expresión. Existe un orden de precedencia perfectamente definido, de forma que en ningún caso una expresión puede tener diferentes interpretaciones. Y el compilador de C siempre entenderá las expresiones de acuerdo con su orden de precedencia establecido.
3. Su **asociatividad**. Esta propiedad resuelve la ambigüedad en la elección de operadores que tengan definida la misma precedencia.

En la práctica habitual de un programador, se acude a dos reglas para lograr escribir expresiones que resulten correctamente evaluadas:

1. Hacer uso de **paréntesis**. Los paréntesis son un operador más; de hecho son los primeros en el orden de ejecución. De acuerdo con esta regla, la expresión antes recogida podría escribirse `(a + b) * c;` ó `a + (b * c);` en función de cuál de las dos se desea. Ahora, con los paréntesis, estas expresiones no llevan a equívoco alguno. De todas formas, para el compilador tampoco antes, cuando no utilizamos los paréntesis, había ninguna ambigüedad.
2. Conocer y aplicar las **reglas de precedencia** y de asociación por izquierda y derecha. Este orden podrá ser siempre alterado mediante el uso de paréntesis. Según esta regla, la expresión `a + b * c` se interpreta como `a + (b * c)`.

   Se considera buena práctica de programación conocer esas reglas de precedencia y no hacer uso abusivo de los paréntesis. De todas formas, cuando se duda sobre cómo se evaluará una expresión, lo

habitual es echar mano de los paréntesis. A veces una expresión adquiere mayor claridad si se recurre al uso de los paréntesis.

Las reglas de precedencia se recogen en la Tabla 7.5. Cuanto más alto en la tabla esté el operador, más alta es su precedencia, y antes se evalúa ese operador que cualquier otro que esté más abajo en la tabla. Y para aquellos operadores que estén en la misma fila, es decir, que tengan el mismo grado de precedencia, el orden de evaluación, en el caso en que ambos operadores intervengan en una expresión, viene definido por la asociatividad: de derecha a izquierda o de izquierda a derecha. Tenga en cuenta que todavía no hemos presentado todos los operadores que existen en C. En la Tabla 7.5. se recogen todos los operadores: algunos, aún no los ha conocido.

Existen 16 categorías de precedencia. Algunas de esas categorías tan solo tienen un operador., Todos los operadores colocados en la misma categoría tienen la misma precedencia.

1	()	[]	->	.							I – D
2	!	~	++	--	+	-	*	&			D – I
3	.*	->*									I – D
4	*	/	%								I – D
5	+	-									I – D
6	<<	>>									I – D
7	>	>=	<	<=							I – D
8	==	!=									I – D
9	&										I – D
10	^										I – D
11	|										I – D
12	&&										I – D
13	||										D – I
14	?:										I – D
15	=	+=	-=	*=	/=	%=	&=	|=	<<=	>>=	D – I
16	,										I – D

Tabla 7.5. Precedencia y Asociatividad de los operadores.

Cuando un operador viene duplicado en la tabla, la primera ocurrencia es como operador unario, la segunda como operador binario.

Cada categoría tiene su regla de asociatividad: de derecha a izquierda (anotada como D – I), o de izquierda a derecha (anotada como I – D).

Por ejemplo, la expresión a * x + b * y – c / z se evalúa en el siguiente orden: primero los productos y el cociente, y ya luego la suma y la resta. Todos estos operadores están en categorías con asociatividad de izquierda a derecha, por lo tanto, primero se efectúa el producto más a la izquierda y luego el segundo, más al centro de la expresión. Después se efectúa el cociente, que está más a la derecha; luego la suma y finalmente la resta.

Todos los operadores de la Tabla 7.5. que faltan por presentar en el manual se emplean para vectores y cadenas y para operatoria de punteros. Más adelante se conocerán todos ellos.

Valores fuera de rango en una variable.

Ya hemos dicho repetidamente que una variable es un espacio de memoria, de un tamaño concreto en donde la información se codifica de una manera determinada por el tipo de dato que se vaya a almacenar en esa variable. Espacio de memoria… limitado. Es importante conocer los límites de nuestras variables (cfr. Tabla 7.1.).

Cuando en un programa se pretende asignar a una variable un valor que no pertenece al dominio, el resultado es habitualmente extraño. Se suele decir que es imprevisible, pero la verdad es que la electrónica del procesador actúa de la forma para la que está diseñada, y no son valores aleatorios los que se alcanzan entonces, aunque sí, muchas veces, valores no deseados, o valores erróneos.

Se muestra ahora el comportamiento de las variables ante un desbordamiento (que así se le llama) de la memoria. Si la variable es

entera, ante un desbordamiento de memoria el procesador trabaja de la misma forma que lo hace, por ejemplo, el cuenta kilómetros de un vehículo. Si en un cuenta kilómetros de cinco dígitos, está marcado el valor 99.998 kilómetros, al recorrer cinco kilómetros más, el valor que aparece en pantalla será 00.003. Se suele decir que se le ha dado la vuelta al marcador. Y algo similar ocurre con las variables enteras. En la Tabla 7.6. se muestran los valores de diferentes operaciones con desbordamiento.

signed short	32767 + 1 da el valor –32768
unsigned short	65535 + 1 da el valor 0
signed long	2147483647 +1 da el valor -2147483648
unsigned long	4294967295 + 1 da el valor 0

Tabla 7.6. Desbordamiento en las variables de tipo entero.

Si el desbordamiento se realiza por asignación directa, es decir, asignando a una variable un literal que sobrepasa el rango de su dominio, o asignándole el valor de una variable de rango superior, entonces se almacena el valor truncado. Por ejemplo, si a una variable **unsigned short** se le asigna un valor que requiere 25 dígitos binarios, únicamente se quedan almacenados los 16 menos significativos. A eso hay que añadirle que, si la variable es **signed short**, al tomar los 16 dígitos menos significativos, interpretará el más significativo de ellos como el bit de signo, y según sea ese bit, interpretará toda la información codificada como entero negativo en complemento a la base, o como entero positivo.

Hay situaciones y problemas donde jugar con las reglas de desbordamiento de enteros ofrece soluciones muy rápidas y buenas. Pero, evidentemente, en esos casos hay que saber lo que se hace.

Si el desbordamiento es por asignación, la variable entera desbordada almacenará un valor que nada tendrá que ver con el original. Si el desbordamiento tiene lugar por realizar operaciones en un tipo de dato de coma flotante y en las que el valor final es demasiado grande para ese tipo de dato, entonces el resultado es completamente imprevisible, y posiblemente se produzca una interrupción en la ejecución del programa. Ese desbordamiento se considera, sin más, error de programación. No resulta sencillo pensar en una situación en la que ese desbordamiento pudiera ser deseado.

Constantes (variables **const**). Directiva #define.

Cuando se desea crear una variable a la que se asigna un valor inicial que no debe modificarse, se la precede, en su declaración, de la palabra clave de C **const**.

```
const tipo var_1 = val1[, var_2 = val2, ..., var_N = valN];
```

Se declara con la palabra reservada **const**. Pueden definirse constantes de cualquiera de los tipos de datos simples.

```
const float DOS_PI = 6.28;
```

Como no se puede modificar su valor, la variable declarada como constante no podrá estar en la parte izquierda de una asignación (excepto en el momento de su inicialización). Si se intenta modificar el valor de la variable declarada como constante mediante el operador asignación el compilador dará un error y no se creará el programa.

Otro modo de definir constantes es mediante la directiva de preprocesador o de compilación #define. Ya se ha visto en el capítulo cuatro otra directiva (directiva #include) que se emplea para indicar al compilador los archivos de cabecera.

Un ejemplo sencillo de uso de esta nueva directiva es el siguiente:

```
#define DOS_PI 6.28
```

La directiva `#define` no es una sentencia. Las directivas de compilación son órdenes que se dirigen al compilador y que ejecuta el compilador antes de compilar. La directiva `#define` sustituye, en todas las líneas de código posteriores a su creación, cada aparición de la primera cadena de caracteres por la segunda cadena: en el ejemplo antes presentado, la cadena `DOS_PI` por el valor `6.28`.

Ya se verá cómo esta directiva pude resultar muy útil en ocasiones.

Los enteros muy largos y otras consideraciones adicionales sobre tipos de dato en C90 y C99.

El nuevo estándar C99 admite, además de todos los tipos presentados hasta el momento, algunos otros nuevos tipos. Entre otros, define el tipo de dato booleano y el tipo complejo. No vamos a presentar éstos aquí. Ahora nos centramos únicamente en la ampliación que ofrece C99 para los enteros y las definiciones de sus límites de rango de valores, muy útil a la hora de programar.

C99 ofrece cinco tipos de dato enteros diferentes: **`char`**, **`short int`**, **`int`**, **`long int`** y **`long long int`**. El tamaño de una variable **`int`** dependerá de la arquitectura del entorno de ejecución. Su rango de valores, como se verá en breve, queda definido como cualquier valor entre `INT_MIN` e `INT_MAX`. Se ha introducido un nuevo tipo de dato entero, de mayor longitud que el que hasta este momento se tenía: el tipo **`long long`**, de 64 bits (8 bytes). Para indicar que un literal expresa un valor de tipo **`long long`** se coloca el sufijo `ll`, ó `LL`, ó `llu`, ó `LLU`.

Existe un archivo de cabecera, ya disponible en el estándar C90: `<limits.h>`, que recoge todos los límites definidos para los rangos de valores o dominios de los tipos de datos enteros y estándares en C. Los tamaños recogidos en `<limits.h>` son los siguientes:

- Número de bits para el menor objeto de memoria
  `#define CHAR_BIT 8`

- Valor mínimo para un objeto de tipo **signed char**:

```
#define SCHAR_MIN          (-128)
```

- Valor máximo para un objeto de tipo **signed char**:

```
#define SCHAR_MAX          127
```

- Valor máximo para un objeto de tipo **unsigned char**

```
#define UCHAR_MAX          255
```

- Valor mínimo para un objeto de tipo **char**

```
#define CHAR_MIN           SCHAR_MIN
```

- Valor máximo para un objeto de tipo **char**

```
#define CHAR_MAX           SCHAR_MAX
```

- Máximo valor de un objeto tipo **long int**

```
#define LONG_MAX           2147483647
```

- Mínimo valor de un objeto tipo **long int**

```
#define LONG_MIN           (-LONG_MAX-1)
```

- Máximo valor de un objeto tipo **unsigned long int**

```
#define ULONG_MAX          0xffffffff
```

- Máximo valor de un objeto tipo **short int**

```
#define SHRT_MAX           32767
```

- Mínimo valor de un objeto tipo **short int**

```
#define SHRT_MIN           (-SHRT_MAX-1)
```

- Máximo valor de un objeto tipo **unsigned short int**

```
#define USHRT_MAX          0xffff
```

- Máximo valor de un objeto tipo **long long int**

```
#define LLONG_MAX          9223372036854775807LL
```

- Mínimo valor de un objeto tipo **long long int**

```
#define LLONG_MIN          (-LLONG_MAX - 1)
```

- Máximo valor de un objeto tipo **unsigned long long int**

```
#define ULLONG_MAX         (2ULL * LLONG_MAX + 1)
```

- Máximo valor de un objeto tipo **int** (dependiendo de la arquitectura)

```
#define INT_MAX            SHRT_MAX
#define INT_MAX            LONG_MAX
```

- Mínimo valor de un objeto tipo **int** (dependiendo de la arquitectura)

```
#define INT_MIN          SHRT_MIN
#define INT_MIN          LONG_MIN
```

- Máximo valor de un objeto tipo **unsigned int** (dependiendo de la arquitectura)

```
#define UINT_MIN         USHRT_MIN
#define UINT_MIN         ULONG_MIN
```

Por otro lado, C99 ofrece un archivo de cabecera, <stdint.h>, que recoge una colección de definiciones, útiles a la hora de programar con enteros. Declara un conjunto de tipos de dato enteros en especificando de forma explícita su tamaño. También define una colección de macros que marcan los límites de estos nuevos tipos de dato enteros.

Los tipos de dato enteros en este archivo, vienen definidos de acuerdo a las siguientes categorías:

- Tipos enteros que tienen una anchura (en bytes) concreta y determinada.

  El formato genérico para expresar cualquiera de estos tipos de dato es el siguiente: intN_t, ó uintN_t, donde N indicará el número de bits y donde la u indica que el tipo de dato es sin signo.

```
int8_t        // 1 byte con signo
uint8_t       // 1 byte sin signo
int16_t       // 2 byte con signo
uint16_t      // 2 byte sin signo
int32_t       // 4 byte con signo
uint32_t      // 4 byte sin signo
int64_t       // 8 byte con signo
uint64_t      // 8 byte sin signo
```

- Tipos enteros que tienen, al menos, una anchura (en bytes) determinada. Estos tipos de datos garantizan un tamaño mínimo de entero, independientemente del sistema en que se compile el programa. Si, por ejemplo, trabajáramos en un sistema que proporcionase sólo tipos uint32_t y uint64_t, entonces el tipo uint_least16_t será equivalente a uint32_t.

El formato genérico para expresar cualquiera de estos tipos de dato es el siguiente; `int_leastN_t`, ó `uint_leastN_t`, donde N indicará el número de bits y donde la u indica que el tipo de datos es sin signo.

```
int_least8_t         // 1 byte con signo
uint_least8_t        // 1 byte sin signo
int_least16_t        // 2 byte con signo
uint_least16_t       // 2 byte sin signo
int_least32_t        // 4 byte con signo
uint_least32_t       // 4 byte sin signo
int_least64_t        // 8 byte con signo
uint_least64_t       // 8 byte sin signo
```

- Los tipos enteros más rápidos que tienen, al menos, una anchura (en bytes) determinada. Con este tipo de dato el compilador trata de establecer, dentro de las condiciones del tipo (anchura y signo) cuál es el óptimo en relación a su velocidad dentro del sistema en que se compila el programa.

  El formato genérico para expresar cualquiera de estos tipos de dato es el siguiente; `int_fastN_t`, ó `uint_fastN_t`, donde N indicará el número de bits y donde la u indica que el tipo de datos es sin signo.

```
int_fast8_t          // 1 byte con signo
uint_fast8_t         // 1 byte sin signo
int_fast16_t         // 2 byte con signo
uint_fast16_t        // 2 byte sin signo
int_fast32_t         // 4 byte con signo
uint_fast32_t        // 4 byte sin signo
int_fast64_t         // 8 byte con signo
uint_fast64_t        // 8 byte sin signo
```

- Los tipos enteros de mayor tamaño.

```
intmax_t       // es el entero con signo de mayor tamaño.
uintmax_t      // es el entero sin signo de mayor tamaño.
```

El archivo `stdint.h` recoge también la definición de límites respecto a estos nuevos tipos de dato (semejante a la recogida en `limits.h`:

- Los límites para los valores del tipo de dato entero de tamaño exacto:

```
#define INT8_MIN     (-128)
#define INT16_MIN    (-32768)
#define INT32_MIN    (-2147483647 - 1)
#define INT64_MIN    (-9223372036854775807LL - 1)

#define INT8_MAX     127
#define INT16_MAX    32767
#define INT32_MAX    2147483647
#define INT64_MAX    9223372036854775807LL

#define UINT8_MAX    0xff /* 255U */
#define UINT16_MAX   0xffff /* 65535U */
#define UINT32_MAX   0xffffffff  /* 4294967295U */
#define UINT64_MAX   0xffffffffffffffffULL
                     /* 18446744073709551615ULL */
```

- Los límites para los valores del tipo de dato de tamaño "al menos" dependerán evidentemente del sistema para el que se haya definido el compilador. Si suponemos un sistema que soporta todos los tamaños: 8, 16, 32 y 64, esos valores serán:

```
#define INT_LEAST8_MIN    INT8_MIN
#define INT_LEAST16_MIN   INT16_MIN
#define INT_LEAST32_MIN   INT32_MIN
#define INT_LEAST64_MIN   INT64_MIN

#define INT_LEAST8_MAX    INT8_MAX
#define INT_LEAST16_MAX   INT16_MAX
#define INT_LEAST32_MAX   INT32_MAX
#define INT_LEAST64_MAX   INT64_MAX

#define UINT_LEAST8_MAX   UINT8_MAX
#define UINT_LEAST16_MAX  UINT16_MAX
#define UINT_LEAST32_MAX  UINT32_MAX
#define UINT_LEAST64_MAX  UINT64_MAX
```

- Los límites para los valores del tipo de dato de tamaño "los más rápidos" también dependerán evidentemente del sistema para el que se haya definido el compilador. Si suponemos un sistema que soporta todos los tamaños: 8, 16, 32 y 64, esos valores serán:

```
#define INT_FAST8_MIN     INT8_MIN
#define INT_FAST16_MIN    INT16_MIN
#define INT_FAST32_MIN    INT32_MIN
#define INT_FAST64_MIN    INT64_MIN

#define INT_FAST8_MAX     INT8_MAX
```

```
#define INT_FAST16_MAX     INT16_MAX
#define INT_FAST32_MAX     INT32_MAX
#define INT_FAST64_MAX     INT64_MAX

#define UINT_FAST8_MAX     UINT8_MAX
#define UINT_FAST16_MAX    UINT16_MAX
#define UINT_FAST32_MAX    UINT32_MAX
#define UINT_FAST64_MAX    UINT64_MAX
```

- Los límites para los valores del tipo de dato de tamaño mayor:

```
#define INTMAX_MIN         INT64_MIN
#define INTMAX_MAX         INT64_MAX
#define UINTMAX_MAX        UINT64_MAX
```

Ayudas *On line*.

Muchos editores y compiladores de C cuentan con ayudas en línea abundante. Todo lo referido en este capítulo puede encontrarse en ellas. Es buena práctica de programación saber manejarse por esas ayudas, que llegan a ser muy voluminosas y que gozan de buenos índices para lograr encontrar el auxilio necesario en cada momento.

Recapitulación.

Después de estudiar este capítulo, ya sabemos crear y operar con nuestras variables. También conocemos muchos de los operadores definidos en C. Podemos realizar ya muchos programas sencillos.

Si conocemos el rango o dominio de cada tipo de dato, sabemos también de qué tipo conviene que sea cada variable que necesitemos. Y estaremos vigilantes en las operaciones que se realizan con esas variables, para no sobrepasar ese dominio e incurrir en un overflow.

También hemos visto las reglas para combinar, en una expresión, variables y valores de diferente tipo de dato. Es importante conocer bien todas las reglas que gobiernan estas combinaciones porque con frecuencia, si se trabaja sin tiento, se llegan a resultados erróneos.

Ejemplos y ejercicios propuestos.

7.1. ***Escribir un programa que realice las operaciones de suma, resta, producto, cociente y módulo de dos enteros introducidos por teclado.***

Una posible solución a este programa viene recogida en Código 7.8.

Código 7.8. Posible solución al ejercicio 7.1.

```
#include <stdio.h>
int main(void)
{
      signed long a, b;
      signed long sum, res, pro, coc, mod;

      printf("1er. operando ... ");      scanf(" %ld",&a);
      printf("2do. operando ... ");      scanf(" %ld",&b);

// Cálculos
      sum = a + b;
      res = a - b;
      pro = a * b;
      coc = a / b;
      mod = a % b;

// Mostrar resultados por pantalla.
      printf("La suma es igual a %ld\n",       sum);
      printf("La resta es igual a %ld\n",      res);
      printf("El producto es igual a %ld\n",   pro);
      printf("El cociente es igual a %ld\n",   coc);
      printf("El resto es igual a %ld\n",      mod);

      return 0;
}
```

Observación: cuando se realiza una operación de cociente o de resto es muy recomendable antes verificar que, efectivamente, el divisor no es igual a cero. Aún no sabemos hacer esta operación de verificación, (espere al Capítulo 9). Al ejecutarlo será importante que el usuario no introduzca un valor para la variable b igual a cero.

7.2. Repetir el mismo programa para números de coma flotante.

Una posible solución a este programa viene recogida en Código 7.9.

Código 7.9. Posible solución al ejercicio 7.2.

```
#include <stdio.h>
int main(void)
{
      float a, b;
      float sum, res, pro, coc;

      printf("1er. operando ... ");      scanf(" %f",&a);
      printf("2do. operando ... ");      scanf(" %f",&b);

// Cálculos
      sum = a + b;
      res = a - b;
      pro = a * b;
      coc = a / b;
// mod = a % b; : esta operación no está permitida

// Mostrar resultados por pantalla.
      printf("La suma es igual a %f\n", sum);
      printf("La resta es igual a %f\n", res);
      printf("El producto es igual a %f\n", pro);
      printf("El cociente es igual a %f\n", coc);
      return 0;
}
```

En este caso se ha tenido que omitir la operación módulo, que no está definida para valores del dominio de los números de coma flotante. Al igual que en el ejemplo anterior, se debería verificar (aún no se han presentado las herramientas que lo permiten), antes de realizar el cociente, que el divisor era diferente de cero.

7.3. Escriba un programa que solicite un entero y muestre por pantalla su valor al cuadrado y su valor al cubo.

Una posible solución a este programa viene recogida en Código 7.10. Es importante crear una presentación cómoda para el usuario. No tendría sentido comenzar el programa por la función `scanf`, porque en ese caso el programa comenzaría esperando un dato, sin aviso previo que le indicase al usuario qué es lo que debe hacer. En el siguiente capítulo se presentan las dos funciones de entrada y salida por consola.

Código 7.10. Posible solución al ejercicio 7.3.

```
#include <stdio.h>
int main(void)
{
      short x;
      long cuadrado, cubo;

      printf("Introduzca un valor: ");   scanf(" %hi",&x);

      cuadrado = (long)x * x;
      cubo = cuadrado * x;

      printf("El cuadrado de %hd es %li\n",x, cuadrado);
      printf("El cubo de %hd es %li\n",x, cubo);
      return 0;
}
```

La variable x es **short**. Al calcular el valor de la variable cuadrado forzamos el tipo de dato para que el valor calculado sea **long** y no se pierda información en la operación. En el cálculo del valor de la variable cubo no es preciso hacer esa conversión, porque ya la variable cuadrado es de tipo **long**. En esta última operación no queda garantizado que no se llegue a un desbordamiento: cualquier valor de x mayor de 1290 tiene un cubo no codificable con 32 bits. Se puede probar qué ocurre con valores mayores que éste indicado: verá que el programa se ejecuta normalmente pero ofrece resultados erróneos.

7.4. ***Escriba un programa que solicite los valores de la base y de la altura de un triángulo y que muestre por pantalla el valor de la superficie.***

Una posible solución a este programa viene recogida en Código 7.11.

Código 7.11. Posible solución al ejercicio 7.4.

```
#include <stdio.h>
int main(void)
{
      double b, h, S;

      printf("Base ..... ");        scanf(" %lf",&b);
      printf("Altura ... ");        scanf(" %lf",&h);

      S = b * h / 2;

      printf("La superficie del triangulo de base");
      printf(" %.2lf y altura %.2lf es %.2lf", b , h , S);
      return 0;
}
```

Las variables se han tomado **double**. Así no se pierde información en la operación cociente. Puede probar a declarar las variables como de tipo **short**, modificando también algunos parámetros de las funciones de entrada y salida por consola (cfr. Código 7.12.). Si al ejecutar el programa de Código 7.12. el usuario introduce, por ejemplo, los valores 5 para la base y 3 para la altura, el valor de la superficie que mostrará el programa será ahora de 7, y no de 7,5.

Código 7.12. Posible solución al ejercicio 7.4. con variables enteras.

```
#include <stdio.h>
int main(void)
{
      short b, h, S;

      printf("Base ..... ");        scanf(" %hd",&b);
      printf("Altura ... ");        scanf(" %hd",&h);

      S = b * h / 2;

      printf("La superficie del triangulo de ");
      printf("base %hd y altura %hd ", b, h);
      printf("es %hd", S);
      return 0;
}
```

7.5. ***Escriba un programa que solicite el valor del radio y muestre la longitud de su circunferencia y la superficie del círculo inscrito en ella.***

Una posible solución a este programa viene recogida en Código 7.13. En este ejemplo hemos mezclado tipos de dato. El radio lo tomamos como entero. Luego, en el cálculo de la longitud `l`, como la expresión tiene el

valor de la constante `pi`, que es **`double`**, se produce una conversión implícita de tipo de dato, y el resultado final es **`double`**.

Para el cálculo de la superficie, en lugar de emplear la constante `pi` se ha tomado un identificador PI definido con la directiva `#define`.

Código 7.13. Posible solución al ejercicio 7.5.

```
#include <stdio.h>
#define PI 3.14159
int main(void)
{
      signed short int r;
      double l, S;
      const double pi = 3.14159;

      printf("Valor del radio ... ");    scanf(" %hd", &r);

      l = 2 * pi * r;
      printf("Longitud circunferencia: %lf. \n", l);

      S = PI * r * r;
      printf("Superficie circunferencia: %lf. \n", S);

      return 0;
}
```

¿Cómo calcularía el volumen de la circunferencia de radio `r`? (Recuerde que $V = (4 / 3) \times \pi \times r^3$)

7.6. ***Escriba un programa que solicite el valor de la temperatura en grados Fahrenheit y muestre por pantalla el equivalente en grados Celsius. La ecuación que define esta trasformación es:***

Celsius = (5 / 9) · (Fahrenheit – 32).

Código 7.14. Posible (y errónea!) solución al ejercicio 7.6.

```
#include <stdio.h>
int main(void)
{
      double fahr, cels;

      printf("Temperatura en grados Fahrenheit ... ");
      scanf(" %lf",&fahr);

      cels = (5 / 9) * (fahr - 32);

      printf("La temperatura en grados Celsius ");
      printf("resulta ... %lf.",cels);

      return 0;
}
```

Una posible solución a este programa viene recogida en Código 7.15. En principio parece que todo está bien. Pero si ensayamos el programa con distintas entradas de los grados expresados en Fahrenheit… ¡siempre obtenemos como temperatura en Celsius 0 grados! ¿Por qué?

Pues porque (5 / 9) es una operación cociente entre dos enteros, cuyo resultado es un entero: en este caso, al ser positivo, tanto para C90 como para C99 será el valor truncado; es decir, 0.

¿Cómo se debería escribir la operación?

```
cels = (5 / 9.0) * (fahr - 32);
cels = (5.0 / 9) * (fahr - 32);
cels = ((double)5 / 9) * (fahr - 32);
cels = (5 / (double)9) * (fahr - 32);
cels = 5 * (fahr - 32) / 9;
```

Hay muchas formas: lo importante es saber en todo momento qué operación realizará la sentencia que escribimos.

7.7. ***Indicar el valor que toman las variables en las asignaciones recogidas en Código 7.15. Justifique sus respuestas***

Código 7.15. Código enunciado para la pregunta 7.7.

```
int a = 10, b = 4, c, d;
float x = 10.0 , y = 4.0, z, t, v;

c = a/b;                          // c valdrá 2
d = x/y;                          // d valdrá 2
z = a/b;                          // z valdrá 2.0
z = x / y;                        // z vale ahora 2.5
t = (1 / 2) * x;                  // t vale 0.0
v = (1.0 / 2) * x;                // v vale 5.0
```

c1 = 7 c2 = 7 d1 = 7.0 d2 = 7.5

7.8. ***Vea el programa propuesto en Código 7.16. y justifique la salida que ofrece por pantalla.***

Código 7.16. Código enunciado para la pregunta 7.8.

```
#include <stdio.h>
#include <stdint.h>
void main(void)
{
      int8_t a = 127;
      a++;
      printf("%hd", a);
}
```

La salida que se obtiene con este código es…–128. Intente justificar por qué ocurre. No se preocupe si aún no conoce el funcionamiento de la función `printf`. Verdaderamente la variable a ahora vale –128. ¿Por qué?

7.9. *Escribir un programa que indique cuántos bits y bytes ocupa una variable* `long`*.*

Una posible solución a este programa solicitado podría ser el propuesto en Código 7.17. La solución es inmediata gracias al operador **`sizeof`**.

Código 7.17. Posible solución al ejercicio 7.9.

```
#include <stdio.h>
int main(void)
{
	short bits, bytes;

	bytes = sizeof(long);
	bits = 8 * bytes;
	printf("BITS = %hd - BYTES = %hd.", bits, bytes);
	return 0;
}
```

7.10. *Rotaciones de bits dentro de un entero.*

Una operación que tiene uso en algunas aplicaciones de tipo criptográficos es la de rotación de un entero. Rotar un número x posiciones hacia la derecha consiste en desplazar todos sus bits hacia la

derecha esas x posiciones, pero sin perder los x bits menos significativos, que pasarán a situarse en la parte más significativa del número. Por ejemplo, el número a = 1101 0101 0011 1110 rotado 4 bits a la derecha queda 1110 1101 0101 0011, donde los cuatro bits menos significativos (1110) se han venido a posicionar en la izquierda del número.

La rotación hacia la izquierda es análogamente igual, donde ahora los x bits más significativos del número pasan a la derecha del número. El número a rotado 5 bits a la izquierda quedaría: 1010 0111 1101 1010.

¿Qué órdenes deberíamos dar para lograr la rotación de un entero, 5 bits a la izquierda?*: Puede verlas en Código 7.18.*

Código 7.18. Rotación de 5 bits a izquierda.

```
unsigned short int a, b, despl;

a = 0xABCD;
despl = 5;
b = ((a << despl) | (a >> (8 * sizeof(short) - despl)));
```

Veamos cómo funciona este código:

```
                          a:  1010 1011 1100 1101
                      despl:  5
                 a << despl:  0111 1001 1010 0000     ←
    8 * sizeof(short) – despl:  8 * 2 – 5 = 11
                  a >> 11:  0000 0000 0001 0101     ←
                          b:  0111 1001 1011 0101
```

Y para lograr la rotación de bits a la derecha: Para ello, se puede ejecutar el código propuesto en Código 7.19.

Código 7.19. Rotación de 5 bits a derecha.

```
unsigned short int a, b, despl;

a = 0xABCD;
despl = 5;
b = ((a >> despl) | (a << (8 * sizeof(short) - despl)));
```

7.11. ***Al elevar al cuadrado el número áureo se obtiene el mismo valor que al sumarle 1. Haga un programa que calcule y muestre por pantalla el número áureo.***

Una posible solución a este programa viene recogida en Código 7.20.

Código 7.20. Posible solución al ejercicio 7.11.

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

int main(void)
{
      double au;

      printf("Número AUREO: tal que x + 1 = x * x.\n");

// Cálculo del número Aureo
      au = (1 + sqrt(5)) / 2;

      printf("El número AUREO es %10.8lf\n", au);
      printf("aureo + 1 ........ %10.8lf\n", au + 1);
      printf("aureo * aureo .... %10.8lf\n", au * au);

      return 0;
}
```

La función `sqrt` está definida en la biblioteca `math.h`. Calcula el valor de la raíz cuadrada de un número. Espera como parámetro una variable de tipo **`double`**, y devuelve el valor en este formato o tipo de dato.

El ejercicio es muy sencillo. La única complicación (si se le puede llamar complicación a esta trivialidad) es saber cómo se calcula el número áureo a partir de la definición aportada. Muchas veces el problema de la programación no está en el lenguaje, sino en saber expresar una solución viable de nuestro problema.

7.12. **Escriba un programa que solicite un valor entero no negativo menor que 10000 y muestre en pantalla y separados el dígito de los millares, de las centenas, de las decenas y de las unidades.**

Un posible código del problema podría ser el propuesto en Código 7.21.

Código 7.21. Posible solución al ejercicio 7.12.

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

	printf("Valor de a ... ");	scanf("%lu", &a);
	printf("\nMillares: %hd", a / 1000);
	a = a % 1000;
	printf("\nCentenas: %hd", a / 100);
	a = a % 100;
	printf("\nDecenas: %hd", a / 10);
	a = a % 10;
	printf("\nUnidades: %hd", a);

	return 0;
}
```

7.13. **Lea el código propuesto en Código 7.22. y muestre la salida que ofrecerá por pantalla.**

Código 7.22. Código enunciado para la pregunta 7.13.

```
#include <stdio.h>
#include <stdint.h>
int main(void)
{
      int16_t a = 0xFFF4, b = 12, c = a == -b ? 1: 0;
      printf("%hd \t %hd \t %hd", a, b, c);
      return 0;
}
```

SOLUCIÓN: **-12** **12** **1**. ¡Explíquelo!

7.14. **Muestre la salida que, por pantalla, ofrecerá el código propuesto en Código 7.23.**

Código 7.23. Código enunciado para la pregunta 7.14.

```
#include <stdio.h>
#include <stdint.h>
int main(void)          {
      int16_t a = 0xFFFF, b = 0xFFFF;
      printf("%s", a == b ? "IGUALES" : "DISTINTOS");
      return 0;
}
```

La solución es : **DISTINTOS**. ¿Por qué?

La variable a es de tipo entero corto con signo; su rango o dominio de valores es el intervalo [-32.768 ... +32.767]. Al asignarle el literal hexadecimal 0xFFFF se le da un valor negativo (su bit más significativo es un 1), igual a –1.

La variable b es de tipo entero corto sin signo; su rango o dominio de valores es el intervalo [0 ... +65.535]. Al asignarle el literal hexadecimal 0xFFFF se le da un valor positivo (este tipo de dato no admite valores menores que cero), igual a +65535.

Desde luego ambos valores son diferentes, y así los considera el programa que aquí se recoge.

Qué ocurriría si el enunciado hubiera sido otro con ésta de ahora declaración de variables:

```
short a = 0xFFFF;
long b = 0xFFFF;
```

De nuevo el resultado hubiera sido "DISTINTOS", porque ahora el valor de la variable b es positivo puesto que la variable b tiene 32 bits y el literal sólo asigna valor a los 16 menos significativos: el valor de b comienza con el bit cero.

CAPÍTULO 8

FUNCIONES DE ENTRADA Y SALIDA POR CONSOLA.

Hasta el momento, hemos presentado las sentencias de creación y declaración de variables. También hemos visto multitud de operaciones que se pueden realizar con las diferentes variables y literales. Pero aún no sabemos cómo mostrar un resultado por pantalla. Y tampoco hemos aprendido todavía a introducir información, para un programa en ejecución, desde el teclado.

El objetivo de este capítulo es iniciar en la comunicación entre el programa y el usuario.

Lograr que el valor de una variable almacenada de un programa sea mostrado por pantalla sería una tarea compleja si no fuese porque ya ANSI C ofrece funciones que realizan esta tarea. Y lo mismo ocurre cuando el programador quiere que sea el usuario quien teclee una entrada durante la ejecución del programa. Quizá a usted le parezca

trivial el hecho de que, al pulsar una tecla del teclado aparezca un carácter concreto y correcto en la pantalla. Quizá eso es lo que usted esperaba que hiciera, y lleva toda su vida acostumbrado a que así se cumpla. Pero que usted esté acostumbrado a verlo no quiere decir que sea trivial. De hecho, usted posiblemente no sepa cómo ocurre.

Estas funciones, de entrada y salida estándar de datos por consola, están declaradas en un archivo de cabecera llamado stdio.h. Siempre que deseemos usar estas funciones deberemos añadir, al principio del código de nuestro programa, la directriz `#include <stdio.h>`.

Salida de datos. La función `printf`.

El prototipo de la función es el siguiente:

```
int printf(const char *cadena_control[, argumento, ...]);
```

Qué es un **prototipo** de una función es cuestión que habrá que explicar en otro capítulo. Sucintamente, diremos que el prototipo indica el modo de empleo de la función: su interfaz: qué tipo de dato devuelve y qué valores espera recibir cuando se hace uso de ella. El prototipo nos sirve para ver cómo se emplea esta función.

La función `printf` devuelve un valor entero. Se dice que es de tipo **`int`**. La función `printf` devuelve un entero que indica el número de bytes que ha impreso en pantalla. Si, por la causa que sea, la función no se ejecuta correctamente, en lugar de ese valor entero lo que devuelve es un valor que significa error (por ejemplo un valor negativo). No descendemos a más detalles.

La función, como toda función, lleva después del nombre un par de paréntesis. Entre ellos va redactado el texto que deseamos que quede impreso en la pantalla. La `cadena_control` indica el texto que debe ser impreso, con unas especificaciones que indican el formato de esa impresión; es una cadena de caracteres recogidos entre comillas, que

indica el texto que se ha de mostrar por pantalla. A lo largo de este capítulo aprenderemos a crear esas cadenas de control que especifican la salida y el formato que ha de mostrar la función `printf`.

Para comenzar a practicar, empezaremos por escribir en el editor de C el código propuesto en Código 8.1. Es muy recomendable que a la hora de estudiar cualquier lenguaje de programación, y ahora en concreto el lenguaje C, se trabaje delante de un ordenador que tenga un editor y un compilador de código.

Código 8.1.

```
#include <stdio.h>
int main(void)
{
      printf("Texto a mostrar en pantalla");
      return 0;
}
```

Que ofrecerá la siguiente salida por pantalla

```
Texto a mostrar en pantalla
```

Y así, cualquier texto que se escriba entre las comillas aparecerá en pantalla.

Si introducimos ahora otra instrucción con la función `printf` a continuación y debajo de la otra, por ejemplo

```
printf("Otro texto");
```

Entonces lo que tendremos en pantalla será

```
Texto a mostrar en pantallaOtro texto
```

Y es que la función `printf` continua escribiendo donde se quedó la vez anterior.

Muchas veces nos va a interesar introducir, en nuestra cadena de caracteres que queremos imprimir por pantalla, algún carácter de, por ejemplo, salto de línea. Pero si tecleamos la tecla intro en el editor de C lo que hace el cursor en el editor es cambiar de línea y eso que no queda reflejado luego en el texto que muestra el programa en tiempo de ejecución.

Para poder escribir este carácter de salto de línea, y otros que llamamos caracteres de control, se escribe, en el lugar de la cadena donde queremos que se imprima ese carácter especial, una barra invertida ('\') seguida de una letra. Cuál letra es la que se debe poner dependerá de qué carácter especial se desea introducir. Esos caracteres de control son caracteres no imprimibles, o caracteres que tienen ya un significado especial en la función `printf`.

Por ejemplo, el código anterior quedaría mejor de la forma propuesta en Código 8.2.

Código 8.2.

```
#include <stdio.h>
int main(void)
{
      printf("Texto a mostrar en pantalla\n");
      printf("Otro texto.");
      return 0;
}
```

que ofrecerá la siguiente salida por consola:

```
Texto a mostrar en pantalla
Otro texto
```

Ya que al final de la cadena del primer `printf` hemos introducido un carácter de control "\n" llamado carácter salto de línea.

\a	Carácter sonido. Emite un pitido breve.
\v	Tabulador vertical.
\0	Carácter nulo.
\n	Nueva línea.
\t	Tabulador horizontal.
\b	Retroceder un carácter.
\r	Retorno de carro.
\f	Salto de página.
\'	Imprime la comilla simple.
\"	Imprime la comilla doble.
****	Imprime la barra invertida '\'.
\xdd	dd es el código ASCII, en hexadecimal, del carácter que se desea imprimir.

Tabla 8.1. Caracteres de control en la cadena de texto de la función `printf`.

Las demás letras con significado para un carácter de control en esta función vienen recogidas en la Tabla 8.1.

Muchas pruebas se pueden hacer ya en el editor de C, para compilar y ver la ejecución que resulta. Gracias a la última opción de la Tabla 8.1. es posible imprimir todos los caracteres ASCII y los tres inmediatamente anteriores sirven para imprimir caracteres que tienen un significado preciso dentro de la cadena de la función `printf`. Gracias a ellos podemos imprimir, por ejemplo, un carácter de comillas dobles evitando que la función `printf` interprete ese carácter como final de la cadena que se debe imprimir.

El siguiente paso, una vez visto cómo imprimir texto prefijado, es imprimir en consola el valor de una variable de nuestro programa.

Cuando en un texto a imprimir se desea intercalar el valor de una variable, en la posición donde debería ir ese valor se coloca el carácter '%' seguido de algunos caracteres. Según los caracteres que se introduzcan, se imprimirá un valor de un tipo de dato concreto, con un formato de presentación determinado. Ese carácter '%' y esos

caracteres que le sigan son los **especificadores de formato**. Al final de la cadena, después de las comillas de cierre, se coloca una coma y el nombre de la variable que se desea imprimir.

Por ejemplo, el código recogido en Código 8.3. ofrece, por pantalla, la siguiente salida:

```
Ahora c vale 15
y b vale ahora 11
```

Código 8.3.

```
#include <stdio.h>
int main(void)
{
      short int a = 5 , b = 10 , c;
      c = a + b++;
      printf("Ahora c vale %hd\n" , c);
      printf("y b vale ahora %hd" , b);

      return 0;
}
```

Una cadena de texto de la función `printf` puede tener tantos especificadores de formato como se desee. Tantos como valores de variables queramos imprimir por pantalla. Al final de la cadena, y después de una coma, se incluyen tantas variables, separadas también por comas, como especificadores de formato se hayan incluido en la cadena de texto. Cada grupo de caracteres encabezado por `%` en el primer argumento de la función (la cadena de control) está asociado con el correspondiente segundo, tercero, etc. argumento de esa función. Debe existir una correspondencia biunívoca entre el número de especificadores de formato y el número de variables que se recogen después de la cadena de control; de lo contrario se obtendrán resultados imprevisibles y sin sentido.

%d	Entero con signo, en base decimal.
%i	Entero con signo, en base decimal.
%o	Entero (con o sin signo) codificado en base octal.
%u	Entero sin signo, en base decimal.
%x	Entero (con o sin signo) codificado en base hexadecimal, usando letras minúsculas. Codificación interna de los enteros.
%X	Entero (con o sin signo) codificado en base hexadecimal, usando letras mayúsculas. Codificación interna de los enteros.
%f	Número real con signo.
%e	Número real con signo en formato científico, con el exponente 'e' en minúscula.
%E	Número real con signo en formato científico, con el exponente 'e' en mayúscula.
%g	Número real con signo, a elegir entre formato e ó f según cuál sea el tamaño más corto.
%G	Número real con signo, a elegir entre formato E ó f según cuál sea el tamaño más corto.
%c	Un carácter. El carácter cuyo ASCII corresponda con el valor a imprimir.
%s	Cadena de caracteres.
%p	Dirección de memoria.
%n	No lo explicamos aquí ahora.
%%	Si el carácter % no va seguido de nada, entonces se imprime el carácter sin más.

Tabla 8.2. Especificadores de tipo de dato en la función `printf`

El especificador de formato instruye a la función sobre la forma en que deben mostrarse cada uno de los valores de las variables Dicho especificador tienen la forma:

```
%[flags][ancho campo][.precisión][F/N/h/l/L] type
```

Veamos los diferentes componentes del especificador de formato:

- **type**: Es el único argumento necesario. Consiste en una letra que indica el **tipo de dato** a que corresponde al valor que se desea imprimir en esa posición. En la Tabla 8.2. se recogen todos los valores que definen tipos de dato. Esta tabla está accesible en las ayudas de editores y

compiladores de C. El significado de estas letras es bastante intuitivo: si queremos imprimir un valor tipo **char**, pondremos %c. Si queremos imprimir un valor de tipo **int** pondremos %i. Si ese entero es sin signo (**unsigned int**), entonces pondremos %u. La función printf permite expresar los enteros en diferentes bases. Si no especificamos nada, y ponemos %i, entonces imprime en base 10; si queremos que el valor numérico vaya expresado en base octal pondremos %o; si queremos que en hexadecimal, %x ó %X: en minúscula o mayúscula dependiendo de cómo queramos que aparezcan (en minúscula o mayúscula) los dígitos hexadecimales A, B, C, D, E, y F. Si queremos indicarle expresamente que los valores aparezcan en base decimal, entonces pondremos %d: poner %d y poner %i es equivalente. Por último, si queremos imprimir un valor de tipo coma flotante (**float**, **double**, o **long double**), pondremos %f; también puede poner %G ó %E si desea ver esos valores en notación científica (signo, exponente, mantisa) Los demás valores presentados en la Tabla 8.2. no se los puedo explicar todavía: no ha visto aún qué son los punteros ni las cadenas de caracteres. Habrá que esperar.

- **[F / N / h / l / L]**: Estas cinco letras son **modificadores de tipo** y preceden a las letras que indican el tipo de dato que se debe mostrar por pantalla.

  La letra **h** es el modificador **short** para valores enteros. Se puede poner con las letras de tipo de dato i, u, d, o, x, X. No tendría sentido poner, por ejemplo, %hc, ó %hf, porque no existe el tipo de dato short char, ó short float.

  La letra **l** tiene dos significados: es el modificador **long** para valores enteros. Y precediendo a la letra f indica que allí debe ir un valor de tipo **double**. Por lo tanto, podemos poner %li, ó %lu, ó %ld, para variables **signed long**; ó %lu para variables **unsigned long**; ó %lo, ó %lx, ó %lX para variables tanto **signed long** como **unsigned long**: al mostrar el valor en su codificación binaria (el octal o el hexadecimal no son en realidad más que formas cómodas de expresar binario) queremos ver

sus ceros y unos: es tarea entonces nuestra saber interpretar si esos valores son, en cada caso, positivos o negativos. Si ponemos `%lf`, entonces es que vamos a imprimir el valor de una variable **`double`**.

La letra **`L`** precediendo a la letra `f` indica que allí debe ir un valor de tipo **`long double`**. No hay otro significado posible. Cualquier otra combinación con la letra modificadora `L` será un error.

Con el estándar C99 se introdujeron los tipos de datos **`long long`**. Para indicar a la función `printf` que el valor a insertar es de ese tipo, se ponen las letras `ll` (dos veces la ele). Si el valor a mostrar es **`signed long long int`** entonces podremos `%lli` (ó `%llX`, ó `%llx`, ó `%llo`, ó `%lld`). Si el valor a mostrar es **`unsigned long long int`** entonces podremos `%llu` (ó `%llX`, ó `%llx`, ó `%llo`).

El estándar C99 permite trabajar con variables enteras de 1 byte: tipo **`char`**. En ese caso las letras que indican el tipo de dato del valor a insertar, son "`hh`". Si, por ejemplo, el valor a mostrar es **`signed char`** entonces podremos `%hhi` (ó `%hhX`, ó `%hhx`, ó `%hho`, ó `%hhd`).

- **`[ancho campo][.precisión]`**: Estos dos indicadores opcionales deben ir antes de los indicadores del tipo de dato. Con el **ancho de campo**, el especificador de formato indica a la función `printf` la **longitud mínima** que debe ocupar la impresión del valor que allí se debe mostrar. Para mostrar información por pantalla, la función `printf` emplea un tipo de letra de paso fijo. Esto quiere decir que cada carácter impreso ocasiona el mismo desplazamiento del cursor hacia la derecha. Al decir que el ancho de campo indica la longitud mínima se quiere decir que este parámetro señala cuántos avances de cursor deben realizarse, como mínimo, al imprimir el valor.

  Por ejemplo, las instrucciones recogidas en Código 8.4. tienen la siguiente salida:

```
La variable a vale ...    123.
La variable b vale ...   4567.
La variable c vale ... 135790.
```

Código 8.4.

```
long int a = 123, b = 4567, c = 135790;
printf("La variable a vale ... %6li.\n",a);
printf("La variable b vale ... %6li.\n",b);
printf("La variable c vale ... %6li.\n",c);
```

Donde vemos que los tres valores impresos en líneas diferentes quedan alineados en sus unidades, decenas, centenas, etc. gracias a que todos esos valores se han impreso con un ancho de campo igual a 6: su impresión ha ocasionado tantos desplazamientos de cursos como indica el ancho de campo.

Si la cadena o número es mayor que el ancho de campo indicado ignorará el formato y se emplean tantos pasos de cursor como sean necesarios para imprimir correctamente el valor.

Es posible **rellenar con ceros los huecos del avance** de cursor. Para ellos se coloca un 0 antes del número que indica el ancho de campo. Por ejemplo, la instrucción

```
printf("La variable a vale ... %06li.\n",a);
```

ofrece como salida la siguiente línea en pantalla:

```
La variable a vale ... 000123.
```

El **parámetro de precisión** se emplea para valores con coma flotante. Indica el número de decimales que se deben mostrar. Indica cuántos dígitos no enteros se deben imprimir: las posiciones decimales. A ese valor le precede un punto. Si el número de decimales del dato almacenado en la variable es menor que la precisión señalada, entonces la función `printf` completa con ceros ese valor. Si el número de decimales del dato es mayor que el que se indica en el parámetro de precisión, entonces la función `printf` trunca el número.

Código 8.5.

```
double raiz_2 = sqrt(2);
printf("A. Raiz de dos vale %lf\n",raiz_2);
printf("B. Raiz de dos vale %12.1lf\n",raiz_2);
printf("C. Raiz de dos vale %12.3lf\n",raiz_2);
printf("D. Raiz de dos vale %12.5lf\n",raiz_2);
printf("E. Raiz de dos vale %12.7lf\n",raiz_2);
printf("F. Raiz de dos vale %12.9lf\n",raiz_2);
printf("G. Raiz de dos vale %12.11lf\n",raiz_2);
printf("H. Raiz de dos vale %5.7lf\n",raiz_2);
printf("I. Raiz de dos vale %012.4lf\n",raiz_2);
```

Por ejemplo, las líneas de Código 8.5., muestran por pantalla:

```
A. Raiz de dos vale 1.414214
B. Raiz de dos vale          1.4
C. Raiz de dos vale        1.414
D. Raiz de dos vale      1.41421
E. Raiz de dos vale    1.4142136
F. Raiz de dos vale  1.414213562
G. Raiz de dos vale 1.41421356237
H. Raiz de dos vale 1.4142136
I. Raiz de dos vale 0000001.4142
```

La función `sqrt` está declarada en el archivo de cabecera `math.h`. Esta función devuelve un valor **`double`** igual a la raíz cuadrada del valor (también **`double`**) recibido entre paréntesis como parámetro de entrada.

Por defecto, se toman 6 decimales, sin formato. Se ve en el ejemplo el truncamiento de decimales. En el caso G, la función `printf` no hace caso del ancho de campo pues se exige que muestre un valor con 11 caracteres decimales, más uno para la parte entera y otro para el punto decimal (en total 13 caracteres), y se le limita el ancho total a 12. La función `printf`, en línea H, no puede hacer caso a la indicación del ancho de campo: no tiene sentido pretender que el número ocupe un total de 5 espacios y a la vez exigir que sólo la parte decimal ocupe 7 espacios.

Código 8.6.

```
#include <stdio.h>
int main(void)
{
      short a = 2345;
      printf("%4hd\n", a);
      printf("%4hd\n", -a);
      return 0;
}
```

Cuya salida por pantalla es:

```
2345
-2345
```

```
#include <stdio.h>
int main(void)
{
      short a = 2345;
      printf("%+4hd\n", a);
      printf("%+4hd\n", -a);
      return 0;
}
```

Cuya salida por pantalla es:

```
+2345
-2345
```

```
#include <stdio.h>
int main(void)
{
      short a = 2345;
      printf("% 4hd\n", a);
      printf("% 4hd\n", -a);
      return 0;
}
```

Cuya salida por pantalla es:

```
 2345
-2345
```

- **[flags]**: Son caracteres que introducen unas últimas modificaciones en el modo en que se presenta el valor. Algunos de sus valores y significados son:
  - carácter '–': el valor queda justificado hacia la izquierda.
  - carácter '+': el valor se escribe con signo, sea éste positivo o negativo. En ausencia de esta bandera, la función `printf` imprime el signo únicamente si es negativo.
  - carácter en blanco: Si el valor numérico es positivo, deja un espacio en blanco. Si es negativo imprime el signo. Esto permite una mejor

alineación de los valores, haciendo coincidir unidades con unidades, decenas con decenas, centenas con centenas, etc. En Código 8.6. se muestran varias funciones `main` con sus correspondientes salidas por pantalla.

- Existen otras muchas funciones que muestran información por pantalla. Muchas de ellas están definidas en el archivo de cabecera **stdio.h**. Con la ayuda a mano, es sencillo aprender a utilizar muchas de ellas.

Entrada de datos. La función `scanf`.

La función `scanf` de nuevo la encontramos declarada en el archivo de cabecera `stdio.h`. Permite la entrada de datos desde el teclado. La ejecución del programa queda suspendida en espera de que el usuario introduzca un valor y pulse la tecla de validación (intro).

La ayuda de cualquier editor y compilador de C es suficiente para lograr hacer un buen uso de ella. Presentamos aquí unas nociones básicas, suficientes para su uso más habitual. Para la entrada de datos, al igual que ocurría con la salida, hay otras funciones válidas que también pueden conocerse a través de las ayudas de los distintos editores y compiladores de C.

El prototipo de la función es:

```
int scanf(const char *cadena_control[,direcciones,...]);
```

La función `scanf` puede leer del teclado tantas entradas como se le indiquen. De todas formas, se recomienda usar una función `scanf` para cada entrada distinta que se requiera.

El valor que devuelve la función es el del número de entradas diferentes que ha recibido. Si la función ha sufrido algún error, entonces devuelve un valor que significa error (por ejemplo, un valor negativo).

En la cadena de control se indica el tipo de dato del valor que se espera

recibir por teclado. No hay que escribir texto alguno en la cadena de control de la función `scanf`: únicamente el especificador de formato.

El formato de este especificador es similar al presentado en la función `printf`: un carácter `%` seguido de una o dos letras que indican el tipo de dato que se espera. Luego, a continuación de la cadena de control, y después de una coma, se debe indicar **dónde** se debe almacenar ese valor: la posición de una variable que debe ser del mismo tipo que el indicado en el especificador. El comportamiento de la función `scanf` es imprevisible cuando no coinciden el tipo señalado en el especificador y el tipo de la variable.

Las letras que indican el tipo de dato a recibir se recogen en la Tabla 8.3. Los modificadores de tipo de dato son los mismos que para la función `printf`.

La cadena de control tiene otras especificaciones, pero no las vamos a ver aquí. Se pueden obtener en la ayuda del compilador.

`%d`	Entero con signo, en base decimal.
`%i`	Entero con signo, en base decimal.
`%o`	Entero codificado en base octal.
`%u`	Entero sin signo, en base decimal.
`%x`	Entero codificado en base hexadecimal, usando letras minúsculas. Codificación interna de los enteros.
`%X`	Entero codificado en base hexadecimal, usando letras mayúsculas. Codificación interna de los enteros.
`%f`	Número real con signo.
`%e`	Número real con signo en formato científico, con el exponente 'e' en minúscula.
`%c`	Un carácter. El carácter cuyo ASCII corresponda con el valor a imprimir.
`%s`	Cadena de caracteres.
`%p`	Dirección de memoria.
`%n`	No lo explicamos aquí ahora.

Tabla 8.3.: Especificadores de tipo de dato en la función `scanf`.

Además de la cadena de control, la función `scanf` requiere de otro parámetro: el lugar dónde se debe almacenar el valor introducido. **La función** `scanf` **espera, como segundo parámetro, el lugar donde se aloja la variable, no el nombre**. Espera la dirección de la variable. Así está indicado en su prototipo. Para poder saber la dirección de una variable, C dispone de un operador unario: `&`. El operador dirección, prefijo a una variable, devuelve la dirección de memoria de esta variable. El olvido de este operador en la función `scanf` es frecuente en programadores noveles. Y de consecuencias desastrosas: siempre ocurre que el dato introducido no se almacena en la variable que deseábamos, alguna vez producirá alteraciones de otros valores y las más de las veces llevará a la inestabilidad del sistema y se deberá finalizar la ejecución del programa.

Dificultades habituales con las entradas de caracteres.

Cuando la función `scanf` pretende tomar del teclado un valor de tipo **`char`**, con frecuencia aparecen problemas. Lo mismo ocurrirá si utilizamos otras funciones estándares de `stdio.h` para la toma de valores de variables de tipo **`char`** o para cadenas de texto (cfr. Capítulo 12 para este último concepto de tipo de dato)

Por ejemplo, copie el código propuesto en Código 8.7., en un nuevo proyecto en su IDE y cree la siguiente función `main`:

Lo que ocurrirá al ejecutar este código será que la ventana de ejecución quedará a la espera de que usted introduzca el valor de un carácter para la variable `a`; y luego no permita introducir el siguiente carácter para la variable `b`, sino que, directamente, se salte a la entrada de la variable `c`; finalmente tampoco permitirá la entrada del valor para la variable `d`.

No se crea lo que le digo: hágalo: escriba el código y ejecute.

Código 8.7.

```
#include <stdio.h>

int main(void)
{
      char a, b, c, d;

      printf("Caracter a ... ");    scanf("%c", &a);
      printf("Caracter b ... ");    scanf("%c", &b);
      printf("Caracter c ... ");    scanf("%c", &c);
      printf("Caracter d ... ");    scanf("%c", &d);

      printf("\n\nValores introducidos ... \n");
      printf("Caracter a ... %c\n", a);
      printf("Caracter b ... %c\n", b);
      printf("Caracter c ... %c\n", c);
      printf("Caracter d ... %c\n", d);

      return 0;
}
```

Al ejecutar Código 8.7. verá por pantalla algo parecido a lo siguiente:

```
Caracter a ... g
Caracter b ... Caracter c ... q
Caracter d ...

Valores introducidos ...
Caracter a ... g
Caracter b ...

Caracter c ... q
Caracter d ...
```

Este fallo en la ejecución es debido al modo en que se gestionan las entradas por teclado. Desde luego, al ejecutar nuestro programa, la función scanf no toma la entrada directamente del teclado: la función se alimenta de la información que está disponible en el buffer intermedio, que se comporta como un archivo de entrada.

Explicar este comportamiento erróneo exigiría algo de teoría que explicara la forma en que el ordenador y los programas tratan realmente los datos de entrada (en realidad caracteres introducidos al pulsar cada una de las teclas del teclado). Habría que explicar que el ordenador trata las entradas de teclado como si de un archivo se tratase: un archivo que es estándar y que se llama **`stdin`**. Cuando el usuario pulsa una tecla alimenta con un nuevo dato ese archivo de entrada `stdin`. El ordenador dispone de este espacio de memoria o buffer para poder gestionar esas entradas de teclado. Así, el ritmo de entrada de datos del usuario no deberá ir necesariamente acompasado con el ritmo de lectura del ordenador o de sus programas. Cuando el usuario pulsa teclas, ingresa valores de carácter en el buffer de `stdin`. Cuando algún programa acude a la entrada para obtener información, toma la información del buffer y la interpreta de acuerdo a lo que el programa esperaba: cadena de caracteres, valor numérico entero o decimal, etc.

Puede resolver este problema de varias maneras. Sugerimos algunas:

(1) Inserte en la cadena de control de la función `scanf`, entre la primera de las comillas y el carácter `%`, un espacio en blanco. Así se le indica a la función `scanf` que ignore como entrada de carácter los caracteres en blanco, los tabuladores o los caracteres de nueva línea.

Código 8.8.

```
printf("Caracter a ... ");   scanf(" %c", &a);
printf("Caracter b ... ");   scanf(" %c", &b);
printf("Caracter c ... ");   scanf(" %c", &c);
printf("Caracter d ... ");   scanf(" %c", &d);
```

```
printf("Caracter a ... "); flush(stdin);  scanf(" %c", &a);
printf("Caracter b ... "); flush(stdin);  scanf(" %c", &b);
printf("Caracter c ... "); flush(stdin);  scanf(" %c", &c);
printf("Caracter d ... "); flush(stdin);  scanf(" %c", &d);
```

(2) En general, las dificultades en las entradas por teclado quedan resueltas con la función `fflush`. Esta función descarga el contenido del archivo que reciba como parámetro. Si, previo a la ejecución de la función `scanf` insertamos la sentencia `fflush(stdin);` no tendremos problemas en la lectura de datos tipo carácter por teclado.

El código antes propuesto funcionará correctamente si las líneas de entrada quedan como recoge Código 8.8. Si ejecuta ahora el código, comprobará que no se produce ningún error.

Recapitulación.

Hemos presentado el uso de las funciones `printf` y `scanf`, ambas declaradas en el archivo de cabecera `stdio.h`. Cuando queramos hacer uno de una de las dos funciones, o de ambas, deberemos indicarle al programa con la directiva de preprocesador `#include <stdio.h>`.

El uso de ambas funciones se aprende en su uso habitual. Los ejercicios del capítulo anterior pueden ayudar, ahora que ya las hemos presentado, a practicar con ellas.

Ejercicios.

8.1. ***Escribir un programa que muestre al código ASCII de un carácter introducido por teclado.***

Cfr. Código 8.9. Primero mostramos el carácter introducido con el especificador de tipo `%c`. Y luego mostramos el mismo valor de la variable `ch` con el especificador `%hhd`, es decir, como entero corto de 1 byte, y entonces nos muestra el valor numérico de ese carácter.

Código 8.9. Posible solución a la pregunta 8.1.

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

int main(void)
{
	int8_t ch;

	printf("Introduzca un carácter por teclado ... ");
	fflush(stdin);    // NO ES NECESARIA
	scanf(“ %c",&ch);

	printf("El carácter introducido ha sido %c\n",ch);
	printf("Su código ASCII es el %hhd", ch);

	return 0;
}
```

8.2. ***Lea el programa recogido en Código 8.10., e intente explicar la salida que ofrece por pantalla.***

Código 8.10. Enunciado pregunta 8.2.

```
#include <stdio.h>
int main(void)
{
	signed long int sli;
	signed short int ssi;

	printf("Introduzca un valor negativo para sli ... ");
	scanf(“ %ld",&sli);
	printf("Introduzca un valor negativo para ssi ... ");
	scanf(“ %hd",&ssi);

	printf("El valor sli es %ld\n",sli);
	printf("El valor ssi es %ld\n\n",ssi);
```

Código 8.10. (Cont.)

```
      printf("El valor sli como \"%%lX\" es %lX\n",sli);
      printf("El valor ssi como \"%%hX\" es %hX\n\n",ssi);

      printf("El valor sli como \"%%lu\" es %lu\n",sli);
      printf("El valor ssi como \"%%hu\" es %hu\n\n",ssi);

      printf("El valor sli como \"%%hi\" es %hi\n",sli);
      printf("El valor ssi como \"%%li\" es %li\n\n",ssi);

      printf("El valor sli como \"%%hu\" es %hu\n",sli);
      printf("El valor ssi como \"%%lu\" es %lu\n\n",ssi);

      return 0;
}
```

La salida que ha obtenido su ejecución es la siguiente (valores ingresados: `sli: -23564715`, y `ssi: -8942`).

```
El valor sli es -23564715
El valor ssi es -8942

El valor sli como "%lX" es FE986E55
El valor ssi como "%hX" es DD12

El valor sli como "%lu" es 4271402581
El valor ssi como "%hu" es 56594

El valor sli como "%hi" es 28245
El valor ssi como "%li" es -8942

El valor sli como "%hu" es 28245
El valor ssi como "%lu" es 4294958354
```

Las dos primeras líneas no requieren explicación alguna: recogen las entradas que se han introducido por teclado cuando se han ejecutado las instrucciones de la función `scanf`: el primero (`sli`) es **`long int`**, y se muestra con el especificador de formato `%ld` ó `%li`; el segundo (`ssi`) es **`short int`**, y se muestra con el especificador de formato `%hd` ó `%hi`.

Las siguientes líneas de salida son:

```
El valor sli como "%lX" es FE986E55
El valor ssi como "%hX" es DD12
```

Que muestran los números tal y como los tiene codificados el ordenador: al ser enteros con signo, y ser negativos, codifica el bit más significativo (el bit 31 en el caso de `sli`, el bit 15 en el caso de `ssi`) a uno porque es el bit del signo; y codifica el resto de los bits (desde el bit 30 al bit 0 en el caso de `sli`, desde el bit 14 hasta el bit 0 en el caso de `ssi`) como el complemento a la base del valor absoluto del número codificado.

El número $(8942)_{10}$= $(22EE)_{16}$ se codifica, como número negativo (dígito 15 a 1 y resto el valor en su complemento a la base), de la forma $\boldsymbol{DD12}$. Y el número $(12564715)_{10}$=$(167\ 91AB)_{16}$ se codifica, como número negativo, de la forma **FE98 6E55**.

Las dos siguientes líneas son:

```
El valor sli como "%lu" es 4271402581
El valor ssi como "%hu" es 56594
```

Muestra el contenido de la variable `sli` que considera ahora como entero largo sin signo. Y por tanto toma esos 32 bits, que ya no los considera como un bit de signo y 31 de complemento a la base del número negativo, sino 32 bits de valor positivo codificado en binario: $(FE98\ 6E55)_{16}$=$(4.271.402.581)_{10}$.

Y muestra el contenido de la variable `ssi` que considera ahora como entero corto sin signo. Y por tanto toma esos 16 bits, que ya no los considera como un bit de signo y 15 de complemento a la base del número negativo, sino 16 bits de valor positivo codificado en binario: $(DD12)_{16}$=$(56.594)_{10}$.

Las dos siguientes líneas son:

```
El valor sli como "%hi" es 28245
El valor ssi como "%li" es -8942
```

Al mostrar el valor de `sli` como variable corta, ha truncado los 16 bits más significativos: en lugar de mostrar expresado en base 10, del código $(\texttt{FE98 6E55})_{16}$, muestra únicamente el valor en base 10 del código $(\texttt{6E55})_{16}$: ahora lo interpreta como un entero positivo (comienza con el dígito hexadecimal 6: 0110) cuyo valor expresado en base decimal es 28.245.

Respecto al valor de la variable `ssi`, al ser esta corta y promocionarla a entero largo, simplemente añade a su izquierda dieciséis ceros si el entero codificado es positivo; y dieciséis unos si el entero codificado es negativo: así logra recodificar el valor a un tipo de dato de mayor rango, sin variar su significado: imprime el valor $(\texttt{FFFF DD12})_{16}$, que es, de nuevo, el valor introducido por teclado: $(\texttt{-8942})_{10}$.

Las dos últimas líneas son:

```
El valor sli como "%hu" es 28245
El valor ssi como "%lu" es 4294958354
```

La primera de ellas considera la variable `sli` como una variable de 16 bits. Toma, de los 32 bits de la variable, los 16 bits menos significativos y los interpreta como entero corto sin signo: $(\texttt{6E55})_{16} = (\texttt{28.245})_{10}$. El comportamiento es igual que en el caso anterior: entero truncado, interpretado como positivo.

En la segunda línea muestra el valor numérico que, expresado en base hexadecimal, es igual a $(\texttt{FFFF DD12})_{16}$. Como se explicó antes, ha añadido 16 unos a la izquierda. Pero ahora, al tener que interpretarlos como entero sin signo, lo que hace es mostrar ese valor en su base 10: no toma en consideración ninguna referencia al signo, ni interpreta el valor codificado como complemento a la base: simplemente muestra, en base 10, el valor numérico, es decir, $(\texttt{4.294.958.354})_{10}$.

8.3. Escriba el programa propuesto en Código 8.11. y compruebe cómo es la salida que ofrece por pantalla.

Código 8.11. Enunciado pregunta 8.3.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
      double a = M_PI;
      printf(" 1. El valor de Pi es ... %20.1lf\n"  , a);
      printf(" 2. El valor de Pi es ... %20.2lf\n"  , a);
      printf(" 3. El valor de Pi es ... %20.3lf\n"  , a);
      printf(" 4. El valor de Pi es ... %20.4lf\n"  , a);
      printf(" 5. El valor de Pi es ... %20.5lf\n"  , a);
      printf(" 6. El valor de Pi es ... %20.6lf\n"  , a);
      printf(" 7. El valor de Pi es ... %20.7lf\n"  , a);
      printf(" 8. El valor de Pi es ... %20.8lf\n"  , a);
      printf(" 9. El valor de Pi es ... %20.9lf\n"  , a);
      printf("10. El valor de Pi es ... %20.10lf\n" , a);
      printf("11. El valor de Pi es ... %20.11lf\n" , a);
      printf("12. El valor de Pi es ... %20.12lf\n" , a);
      printf("13. El valor de Pi es ... %20.13lf\n" , a);
      printf("14. El valor de Pi es ... %20.14lf\n" , a);
      printf("15. El valor de Pi es ... %20.15lf\n" , a);
      return 0;
}
```

La salida que ofrece por pantalla es la siguiente:

```
 1. El valor de Pi es ... ···············3.1
 2. El valor de Pi es ... ··············3.14
 3. El valor de Pi es ... ·············3.142
 4. El valor de Pi es ... ············3.1416
 5. El valor de Pi es ... ···········3.14159
 6. El valor de Pi es ... ··········3.141593
 7. El valor de Pi es ... ·········3.1415927
 8. El valor de Pi es ... ·········3.14159265
 9. El valor de Pi es ... ·······3.141592654
10. El valor de Pi es ... ·····3.1415926536
11. El valor de Pi es ... ·····3.14159265359
12. El valor de Pi es ... ····3.141592653590
13. El valor de Pi es ... ····3.1415926535898
14. El valor de Pi es ... ···3.14159265358979
15. El valor de Pi es ... ···3.141592653589793
```

Donde hemos cambiado los espacios en blanco por puntos en la parte de la impresión de los números. Y donde el archivo de biblioteca `math.h` contiene el valor del número pi, en la constante o identificador `M_PI`. Efectivamente, emplea 20 espacios de carácter de pantalla para mostrar cada uno de los números. Y cambia la posición de la coma decimal, pues cada línea exigimos a la función `printf` que muestre un decimal más que en la línea anterior.

CAPÍTULO 9

ESTRUCTURAS DE CONTROL I: ESTRUCTURAS DE SELECCIÓN O SENTENCIAS CONDICIONADAS.

El lenguaje C pertenece a la familia de lenguajes del paradigma de la programación estructurada. En este capítulo y en el siguiente quedan recogidas las reglas de la programación estructurada y, en concreto, las reglas sintácticas que se exige en el uso del lenguaje C para el diseño de esas estructuras.

El objetivo de este capítulo es mostrar cómo crear estructuras condicionales. Una estructura condicional permite al programa decidir, en función de una condición expresada a través de variables y literales y funciones y operadores, si ejecutar una determinada sentencia o grupo de sentencias, o ejecutar otras sentencias alternativas, o ninguna. Veremos también una estructura especial que permite seleccionar un camino de ejecución de entre varios establecidos. Y también se presenta

un nuevo operador, que permite seleccionar, entre dos expresiones posibles, de acuerdo con una condición previa que se evalúa como verdadera o como falsa.

Introducción a las estructuras de control.

Ya hemos visto cuáles son las reglas básicas de la programación estructurada. Conviene recordarlas, al inicio de este capítulo:

1. Todo programa consiste en una serie de acciones o sentencias que se ejecutan en **secuencia**, una detrás de otra.
2. Cualquier acción puede ser sustituida por dos o más acciones en secuencia. Esta **regla** se conoce como la **de apilamiento**.
3. Cualquier acción puede ser sustituida por cualquier estructura de control; y sólo se consideran tres estructuras de control: **la secuencia, la selección y la repetición**. Esta regla se conoce como **regla de anidamiento**. Todas las estructuras de control de la programación estructurada tienen un solo punto de entrada y un solo punto de salida.
4. Las reglas de apilamiento y de anidamiento pueden aplicarse tantas veces como se desee y en cualquier orden.

Ya hemos visto cómo se crea una sentencia: con un punto y coma precedido de una expresión que puede ser una asignación, la llamada a una función, una declaración de una variable, etc. O, si la sentencia es compuesta, agrupando entonces varias sentencias simples en un bloque encerrado por llaves.

Los programas discurren, de instrucción a instrucción, una detrás de otra, en una ordenación secuencial y nunca se ejecutan dos al mismo tiempo. Ya se vio en el Capítulo 5.

Pero un lenguaje de programación no sólo ha de poder ejecutar las instrucciones en orden secuencial: es necesaria la capacidad para

modificar ese orden de ejecución. Para ello están las estructuras de control. Al acabar este capítulo y el siguiente, una vez conocidas todas las estructuras de control, las posibilidades de resolver diferentes problemas mediante el lenguaje de programación C se habrán multiplicado enormemente.

A lo largo del capítulo iremos viendo algunos ejemplos. Es conveniente pararse en cada uno: comprender el código que se propone en el manual, o lograr resolver aquellos que se dejan propuestos. En algunos casos ofreceremos el código en C; en otros dejaremos apuntado el modo de resolver el problema ofreciendo el pseudocódigo del algoritmo o el flujograma.

Transferencia de control condicionada.

La programación estructurada tiene tres estructuras básicas de control: la secuencia, la selección y la repetición. La secuencia se logra sin más que colocando las instrucciones en el orden correcto. La iteración será objeto del siguiente Capítulo. Ahora nos centramos en las estructuras de decisión.

Las dos formas que rompen el orden secuencial de ejecución de sentencias son:

1. **Instrucción condicionada**: Se evalúa una condición y si se cumple se transfiere el control a una nueva dirección indicada por la instrucción. Es el caso de las dos estructuras básicas de decisión y de iteración.
2. **Instrucción incondicionada**. Se realiza la transferencia a una nueva dirección sin evaluar ninguna condición (por ejemplo, llamada a una función: lo veremos en los Capítulos 14, 15 y 17).

En ambos casos la transferencia del control se puede realizar con o sin retorno: en el caso de que exista retorno, después de ejecutar el bloque

de instrucciones de la nueva dirección se retorna a la dirección que sigue o sucede a la que ha realizado el cambio de flujo. En este capítulo vamos a ver las estructuras que trasfieren el control a una nueva dirección, de acuerdo a una condición evaluada.

Bifurcación Abierta.
La estructura condicional `if`.

Las estructuras de control condicionales que se van a ver son la bifurcación abierta y la bifurcación cerrada. Un esquema del flujo de ambas estructuras ha quedado recogido en la Figura 9.1.

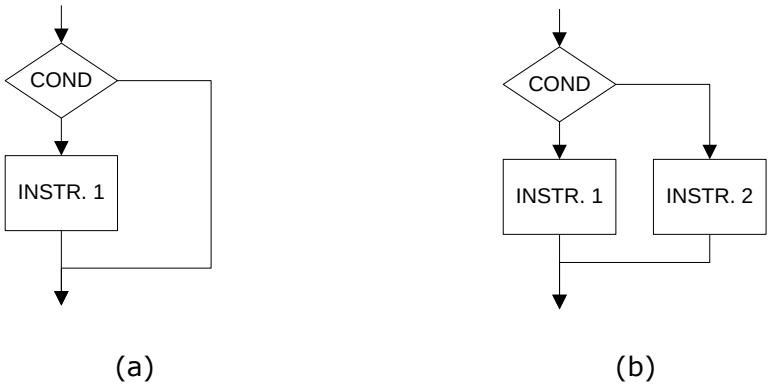


Figura 9.1.: Estructura de selección: (a) Abierta. (b) Cerrada.

La sentencia que está precedida por la estructura de control condicionada se ejecutará si la condición de la estructura de control es verdadera; en caso contrario no se ejecuta la instrucción condicionada y continúa el programa con la siguiente instrucción. En la Figura 9.1.(a) se puede ver un esquema del comportamiento de la bifurcación abierta.

La sintaxis de la estructura de control condicionada abierta es la siguiente:

```
if(condición) sentencia;
```

Si la condición es verdadera (distinto de cero en el lenguaje C), se ejecuta la sentencia. Si la condición es falsa (igual a cero en el lenguaje C), no se ejecuta la sentencia.

Si en lugar de una sentencia, se desean condicionar varias de ellas, entonces se crea una sentencia compuesta mediante llaves.

Podemos ver un primer ejemplo del uso de esta estructura de control en Código 9.1. Tenemos ahí un programa que solicita del usuario dos valores enteros y muestra, luego, su cociente. La división se efectuará únicamente en el caso en que se verifique la condición de que `d != 0`.

Código 9.1.

```
#include <stdio.h>
int main(void)
{
      short D, d;

      printf("Programa para dividir dos enteros...\n");
      printf("Introduzca el dividendo ... ");
      scanf(“ %hd",&D);
      printf("Introduzca el divisor ... ");
      scanf(“ %hd",&d);

      if(d != 0) printf("%hu / %hu = %hu", D, d, D / d);

      return 0;
}
```

Bifurcación Cerrada.
La estructura condicional **`if - else`**.

En una bifurcación cerrada, la sentencia que está precedida por una estructura de control condicionada se ejecutará si la condición de la

estructura de control es verdadera; en caso contrario se ejecuta una instrucción alternativa. Después de la ejecución de una de las dos sentencias (nunca las dos), el programa continúa la normal ejecución de las restantes sentencias que vengan a continuación.

La sintaxis de la estructura de control condicionada cerrada es la siguiente:

```
if(condición) sentencia1;
else sentencia2;
```

Si la condición es verdadera (distinto de cero en el lenguaje C), se ejecuta la sentencia llamada `sentencia1`. Si la condición es falsa (igual a cero en el lenguaje C), se ejecuta la sentencia llamada `sentencia2`.

Si en lugar de una sentencia, se desean condicionar varias de ellas, entonces se crea una sentencia compuesta mediante llaves.

Código 9.2.

```
#include <stdio.h>
int main(void)
{
	short D, d;

	printf("Programa para dividir dos enteros...\n");
	printf("Dividendo ... ");	scanf(" %hd",&D);
	printf("Divisor ..... ");	scanf(" %hd",&d);

	if(d != 0)
	{
		printf("%hu / %hu = %hu\n", D, d, D / d);
	}
	else
	{
		printf("ERROR: división por cero!\n");
	}
	return 0;
}
```

En Código 9.2. se puede ver un programa semejante al propuesto en código 9.1., donde ahora se ha definido una acción a realizar en el caso de que la condición (que `d` sea distinto de cero) resulte evaluada como falsa. Se efectuará la división únicamente en el caso en que se verifique la condición de que `d != 0`. Si el divisor introducido es igual a cero, entonces imprime en pantalla un mensaje de advertencia.

Anidamiento de estructuras condicionales.

Decimos que se produce anidamiento de estructuras de control cuando una estructura aparece dentro de otra del mismo o de distinto tipo.

Una anidamiento de estructuras condicionadas tiene, por ejemplo, la forma esquemática apuntada en Código 9.4. Se puede ver el flujograma correspondiente a ese código en la Figura 9.2. Tanto en la parte **`if`** como en la parte **`else`**, los anidamientos pueden llegar a cualquier nivel. De esa forma podemos elegir entre numerosas sentencias estableciendo las condiciones necesarias.

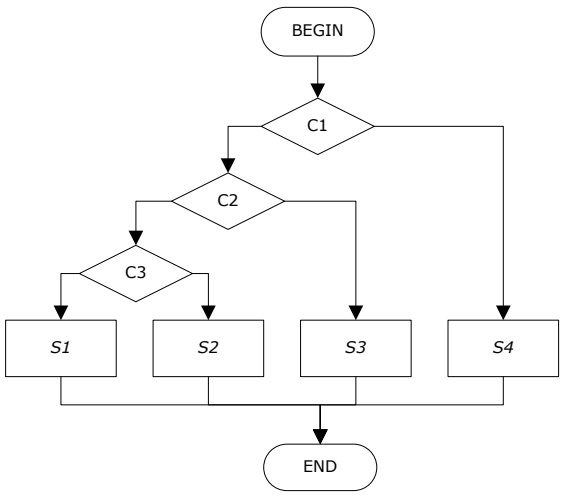


Figura 9.2.: Ejemplo de estructuras condicionales anidadas.

Código 9.3. Esquema de bloque de estructuras condicionales anidadas

```
if(expresión_1)              /* primer if */
{
      if(expresión_2)        /* segundo if */
      {
            if(expresión_3)  /* tercer if */
            {
                  sentencia_1;
            }
            else             /* alternativa al tercer if */
            {
                  sentencia_2;
            }
      }
      else                   /* alternativa al 2º if */
      {
            sentencia_3;
      }
}
else                         /* alternativa al primer if */
{
      sentencia_4;
}
```

Cada **else** se asocia al **if** más próximo en el bloque en el que se encuentre y que no tenga asociado un **else**. No está permitido (no tendría sentido) utilizar un **else** sin un **if** previo. Y la estructura **else** debe ir inmediatamente después de la sentencia condicionada con su **if**.

Un ejemplo de estructura anidada sería, siguiendo con los ejemplos anteriores, el caso de que, si el divisor introducido ha sido el cero, el programa preguntase si se desea introducir un divisor distinto. Puede ver el programa con la nueva corrección en Código 9.4.

La función `getchar` está definida en la biblioteca `stdio.h`. Esta función espera a que el usuario pulse una tecla del teclado y, una vez pulsada, devuelve el código ASCII de la tecla pulsada.

Código 9.4.

```
#include <stdio.h>

int main(void)
{
	short D, d;
	char opcion;

	printf("Programa para dividir dos enteros...\n");
	printf("Introduzca dividendo ... ");
	scanf(" %hd",&D);
	printf("Introduzca el divisor .. ");
	scanf(" %hd",&d);

	if(d != 0)
	{
		printf("%hu / %hu = %hu", D, d, D / d);
	}

	else
	{
		printf("No se puede dividir por cero.\n");
		printf("¿Introducir otro denominador (s/n)?");
		opcion = getchar();
		if(opcion == 's')
		{
			printf("\nNuevo denominador ... ");
			scanf(" %hd",&d);
			if(d != 0)
			{
				printf("%hu / %hu = %hu",
					D, d, D/d);
			}
			else
			{
				printf("De nuevo introduce 0!");
			}
		}
	}
	return 0;
}
```

En el caso de que el usuario del programa introduzca el carácter `'s'` en la ejecución de la función `getchar`, el programa le permitirá introducir otro denominador. Si, de nuevo, el usuario introduce un cero, entonces el programa no hará la división y terminará su ejecución. En este ejemplo hemos llegado hasta un tercer nivel de anidación.

Escala **`if – else`**.

Cuando se debe elegir entre una lista de opciones, y únicamente una de ellas ha de ser válida, se llega a producir una concatenación de condiciones de la siguiente forma indicada en Código 9.5.

Código 9.5. Esquema de bloque de estructuras condicionales anidadas

```
if(condición1)
{
      setencia1;
}
else
{
      if(condición2)
      {
            sentencia2;
      }
      else
      {
            if(condición3)
            {
                  sentencia3;
            }
            else
            {
                  sentencia4;
            }
      }
}
```

Código 9.6. Concatenación de estructuras condicionales: bifurcaciones cerrada y abierta.

```
if(condición1) sentencia1;
else if (condición2) sentencia2;
else if(condición3) sentencia3;
else sentencia4;
```

```
if(condición1) sentencia1;
else if (condición2) sentencia2;
else if(condición3) sentencia3;
```

El flujograma recogido en la Figura 9.2. representaría esta situación sin más que intercambiar los caminos de verificación de las condiciones C1, C2 y C3 recogidas en él (es decir, intercambiando los rótulos de "Sí" y de "No").

Este tipo de anidamiento se resuelve en C con la estructura **else if**, que permite una concatenación de las condicionales. Un código como el antes escrito quedaría tal y como recoge Código 9.6.

Como se ve, una estructura así anidada se escribe con mayor facilidad y expresa también más claramente las distintas alternativas. No es necesario que, en un anidamiento de sentencias condicionales, encontremos un **else** final: el último **if** puede ser una bifurcación abierta, como muestra el segundo ejemplo de Código 9.6.

Un ejemplo de concatenación podría ser el siguiente programa, que solicita al usuario la nota de un examen y muestra por pantalla la calificación académica obtenida. Puede verlo en Código 9.7.

Únicamente se evaluará un **else if** cuando no haya sido cierta ninguna de las condiciones anteriores. Si todas las condiciones resultan ser falsas, entonces se ejecutará (si existe) el último **else**.

Código 9.7. Ejemplo de concatenación de estructuras condicionales.

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

      printf("Introduzca la nota del examen ... ");
      scanf(“ %f",&nota);

      if(nota < 0 || nota > 10)
                              printf("Nota incorrecta.");
      else if(nota < 5)       printf("Suspenso.");
      else if(nota < 7)       printf("Aprobado.");
      else if(nota < 9)       printf("Notable.");
      else if(nota < 10)      printf("Sobresaliente.");
      else                    printf("Matrícula de honor.");

      return 0;
}
```

La estructura condicional y el operador condicional.

Existe un operador que selecciona entre dos opciones, y que realiza, de forma muy sencilla y bajo ciertas limitaciones la misma operación de selección que la estructura de bifurcación cerrada. Es el **operador interrogante, dos puntos** (`?:`).

La sintaxis del operador es la siguiente:

`expresión_1 ? expresión_2 : expresión3;`

Se evalúa `expresión_1`; si resulta ser verdadera (distinta de cero), entonces se ejecutará la sentencia recogida en `expresión_2`; y si es falsa (igual a cero), entonces se ejecutará la sentencia recogida en `expresión_3`. Tanto `expresión_2` como `expresión_3` pueden ser funciones, o expresiones muy complejas, pero siempre deben ser sentencias simples.

Por ejemplo, el código:

```
if(x >= 0)  printf(“Positivo\n”);
else        printf(“Negativo\n”);
```

es equivalente a:

```
printf(“%s\n”, x >= 0 ? “Positivo”: “Negativo”);
```

El uso de este operador también permite anidaciones. Por ejemplo, al implementar el programa que, dada una nota numérica de entrada, muestra por pantalla la calificación en texto (Aprobado, sobresaliente, etc.), podría quedar, con el operador interrogante dos puntos de la forma que se propone en Código 9.8.

Código 9.8. Operador interrogante, dos puntos.

```
#include <stdio.h>

int main(void)
{
      short nota;
      printf("Nota obtenida ... ");
      scanf(“ %hd", &nota);

      printf("%s", nota < 5 ? "SUSPENSO"  :
                  nota < 7 ? "APROBADO"  :
                  nota < 9 ? "NOTABLE"   :
                             "SOBRESALIENTE");
      return 0;
}
```

Es conveniente no renunciar a conocer algún aspecto de la sintaxis de un lenguaje de programación. Es cierto que el operador "*interrogante dos puntos*" se puede fácilmente sustituir por la estructura de control condicional **if – else**. Pero el operador puede, en muchos casos, simplificar el código o hacerlo más elegante. Es mucho más sencillo

crear una expresión con este operador que una estructura de control. Por ejemplo, si deseamos asignar a la variable c el menor de los valores de las variables a y b, podremos, desde luego, usar el siguiente código:

```
if(a < b)          c = a;
else               c = b;
```

Pero es mucho más práctico expresar así la operación de asignación:

```
c = a < b ? a : b;
```

Estructura de selección múltiple: **switch**.

La estructura **switch** permite transferir el control de ejecución del programa a un punto de entrada etiquetado en un bloque de sentencias. La decisión sobre a qué instrucción del bloque se trasfiere la ejecución se realiza mediante una expresión entera.

Código 9.9. Sintaxis esquemática de la estructura **switch**.

```
switch(expresión_del_switch)
{
        case expresionConstante1:
                [sentencias;]
                [break;]

        case expresionConstante2:
                [sentencias;]
                [break;]

        […]
        case expresionConstanteN:
                [sentencias;]
                [break;]

        [default
                sentencias;]
}
```

Esta estructura tiene su equivalencia entre las estructuras derivadas de la programación estructurada y que ya vimos en el Capítulo 5.

Puede ver, en Código 9.9., la forma general de la estructura **switch**. Una estructura **switch** comienza con la palabra clave **switch** seguida de una expresión (`expresión_del_switch`) recogida entre paréntesis. Si la expresión del **switch** no es entera entonces el código dará error en tiempo de compilación.

El cuerpo de la estructura **switch** se conoce como bloque **switch** y permite tener sentencias prefijadas con las etiquetas `case`. Una etiqueta **case** es una constante entera (variables de tipo **char** ó **short** ó **long**, con o sin signo). Si el valor de la expresión de **switch** coincide con el valor de una etiqueta **case**, el control se transfiere a la primera sentencia que sigue a la etiqueta. No puede haber dos **case** con el mismo valor de constante. Si no se encuentra ninguna etiqueta **case** que coincida, el control se transfiere a la primera sentencia que sigue a la etiqueta **default**. Si no existe esa etiqueta **default**, y no existe una etiqueta coincidente, entonces no se ejecuta ninguna sentencia del **switch** y se continúa, si la hay, con la siguiente sentencia posterior a la estructura.

En la Figura 9.3. se muestra un flujograma sencillo que se implementa fácilmente gracias a la estructura **switch**. En Código 9.10. se recoge el código correspondiente a ese flujograma.

Código 9.10. Código que implementa el Flujograma de la Figura 9.3.

```
switch(a)
{
      case 1:     printf("UNO\t");
      case 2:     printf("DOS\t");
      case 3:     printf("TRES\t");
      default:    printf("NINGUNO\n");
}
```

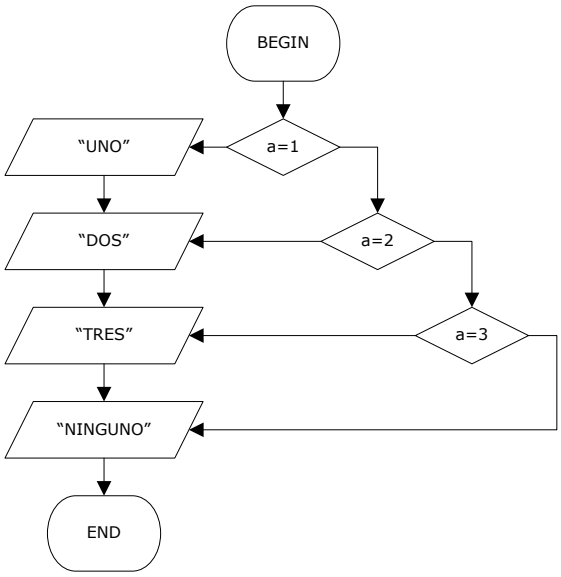


Figura 9.3.: Flujograma del programa ejemplo con **switch** sin sentencias **break**.

Si el valor de a es, por ejemplo, 2, entonces comienza a ejecutar el código del bloque a partir de la línea que da entrada el **case** 2:. Producirá, por pantalla, la salida DOS TRES NINGUNO.

Una vez que el control se ha trasferido a la sentencia que sigue a una etiqueta concreta, ya se ejecutan todas las demás sentencias del bloque **switch**, de acuerdo con la semántica de dichas sentencias. El que aparezca una nueva etiqueta **case** no obliga a que se dejen de ejecutar las sentencias del bloque.

Si se desea detener la ejecución de sentencias en el bloque **switch**, debemos transferir explícitamente el control al exterior del bloque. Y eso se realiza utilizando la sentencia **break**. Dentro de un bloque **switch**, la sentencia break transfiere el control a la primera sentencia posterior al **switch**. Ese es el motivo por el que en la sintaxis de la estructura **switch** se escriba (en forma opcional) las sentencias **break** en las instrucciones inmediatamente anteriores a cada una de las etiquetas.

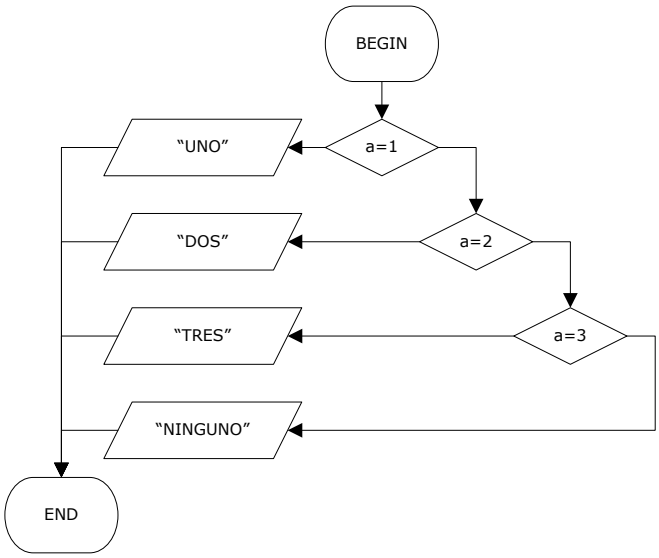


Figura 9.4.: Flujograma del programa ejemplo con **switch** y con sentencias **break**.

Si colocamos la sentencia **break** al final de las sentencias de cada **case**, (cfr. Código 9.11.) el algoritmo cambia y toma la forma indicada en la Figura 9.4. Si la variable a tiene el valor 2, entonces la salida por pantalla será únicamente: `DOS.` La ejecución de las instrucciones que siguen más allá de la siguiente etiqueta **case** puede ser útil en algunas circunstancias. Pero lo habitual será que aparezca una sentencia **break** al final del código de cada etiqueta **case**.

Código 9.11. Código que implementa el Flujograma de la Figura 9.4.

```
switch(a)
{
      case 1:     printf("UNO");          break;
      case 2:     printf("DOS");          break;
      case 3:     printf("TRES");         break;
      default:    printf("NINGUNO");
}
```

Una sola sentencia puede venir marcada por más de una etiqueta **case**. El ejemplo recogido en Código 9.12. resuelve, de nuevo, el programa que indica, a partir de la nota numérica, la calificación textual. En este ejemplo, los valores 0, 1, 2, 3, y 4, etiquetan la misma línea de código.

No se puede poner una etiqueta **case** fuera de un bloque **switch**. Y tampoco tiene sentido colocar instrucciones dentro del bloque **switch** antes de aparecer el primer **case**: eso supondría un código que jamás podría llegar a ejecutarse. Por eso, la primera sentencia de un bloque **switch** debe estar ya etiquetada.

Código 9.12.

```
#include <stdio.h>
int main(void)                          {
      short int nota;
      printf("Nota del examen ... "); scanf(" %hd",&nota);
      switch(nota)                      {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:     printf("SUSPENSO");
                        break;
            case 5:
            case 6:     printf("APROBADO");
                        break;
            case 7:
            case 8:     printf("NOTABLE");
                        break;
            case 9:     printf("SOBRESALIENTE");
                        break;
            case 10:    printf("MATRÍCULA DE HONOR");
                        break;
            default:    printf("Nota introducida errónea");
      }
      return 0;
}
```

Se pueden anidar distintas estructuras **switch**.

El ejemplo de las notas, que ya se mostró al ejemplificar una anidación de sentencias **if–else–if** puede servir para comentar una característica importante de la estructura **switch**. Esta estructura no admite, en sus distintas entradas **case**, ni expresiones lógicas o relacionales, ni expresiones aritméticas, sino literales. La única relación aceptada es, pues, la de igualdad. Y además, el término de la igualdad es siempre entre una variable o una expresión entera (la del **switch**) y valores literales: no se puede indicar el nombre de una variable. El programa de las notas, si la variable `nota` hubiese sido de tipo **float**, como de hecho quedo definida cuando se resolvió el problema con los condicionales **if–else–**`inf`, no tiene solución posible mediante la estructura **switch**.

Y una última observación: las sentencias de un **case** no forman un bloque y no tiene porqué ir entre llaves. La estructura **switch** completa, con todos sus **case**'s, sí es un bloque.

Recapitulación.

Hemos presentado las estructuras de control existentes en el lenguaje C que permiten condicionar la ejecución de una o varias sentencias, o elegir entre una o varias posibles: las estructuras condicionales de bifurcación abierta o cerrada, condicionales anidadas, operador interrogante, dos puntos, y estructura **switch**.

Ejercicios.

En todos estos ejercicios quizá convenga, antes de implementar el código, intentar diseñar el algoritmo en pseudocódigo o mediante un diagrama de flujo. Si en algún caso el problema planteado supone especial dificultad quedará recogido ese flujograma en estas páginas.

9.1. ***Proponer un programa que solicite los parámetros a y b de una ecuación de primer grado (a·x + b = 0) y la resuelva.***

ENUNCIADO: Resolución de una ecuación de primer grado.

1. Inicializar Variables: entrada de valores por el usuario: a, b.
2. **IF** a ≠ 0
   THEN
   R ← –b / a
   Mostrar R
   ELSE
   Mostrar: "No hay ecuación"
   END IF

El código en C podría ser el recogido en Código 9.13. Las variables a y b podrían haber sido declaradas como de tipo **double**, u otro tipo.

Código 9.13. Posible solución al problema 9.1.

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

      printf("Coeficiente a ... ");      scanf(" %hd", &a);
      printf("Coeficiente b ... ");      scanf(" %hd", &b);

      if(a)       printf("x = %lf\n" , (double)(-b) / a);
      else        printf("Ecuación erronea\n");

      return 0;
}
```

9.2. ***Solicitar del usuario tres números y mostrarlos por pantalla ordenados de menor a mayor.***

Una posible solución se recoge en código 9.14., donde, como ya se vio en el tema anterior, el código que se ejecuta en cada estructura **if** intercambia los valores de las dos variables.

Código 9.14. Posible solución al problema 9.2.

```
#include <stdio.h>
int main(void)
{
      unsigned short int a0,a1,a2;
      printf("Introduzca tres enteros ... \n\n");
      printf("Primer entero ... ");      scanf(“ %hu",&a0);
      printf("Segundo entero ... ");     scanf(“ %hu",&a1);
      printf("Tercer entero ... ");      scanf(“ %hu",&a2);

      if(a0 > a1) {     a0 ^= a1;   a1 ^= a0;   a0 ^= a1;   }
      if(a0 > a2) {     a0 ^= a2;   a2 ^= a0;   a0 ^= a2;   }
      if(a1 > a2) {     a1 ^= a2;   a2 ^= a1;   a1 ^= a2;   }

      printf("\nOrdenados... \n");
      printf("%hu <= %hu <= %hu.", a0, a1, a2);

      return 0;
}
```

9.3. *Solicitar del usuario cuatro números y mostrarlos por pantalla ordenados de menor a mayor.*

Una posible solución se recoge en código 9.15.

Quizá al principio podía parecer que ordenar cuatro enteros era lo mismo que ordenar tres, pero un poquito más. Desde luego hay métodos y algoritmos de ordenación muy eficientes, como veremos más adelante en el Capítulo 19. Pero con este algoritmo de ahora (también lo

verá más adelante: este algoritmo que aquí hemos implementado se llama algoritmo de la Burbuja), cada vez que se incrementa en uno la cantidad de enteros a ordenar, se incrementa en mucho la cantidad de comparaciones a realizar. En realidad, esta cantidad de comparaciones es igual a $n \cdot (n-1)/2$. Y así, por ejemplo, si necesitamos ordenar 1000 enteros, habrá que realizar… ¡¡¡499.500 comparaciones!!!

Código 9.15. Posible solución al problema 9.3.

```
#include <stdio.h>
int main(void)
{
      unsigned short int a0,a1,a2,a3;
      printf("Introduzca cuatro enteros ... \n\n");
      printf("Primer entero ... ");      scanf(“ %hu",&a0);
      printf("Segundo entero ... ");     scanf(“ %hu",&a1);
      printf("Tercer entero ... ");      scanf(“ %hu",&a2);
      printf("Cuarto entero ... ");      scanf(“ %hu",&a3);

      if(a0 > a1) {     a0 ^= a1;   a1 ^= a0;   a0 ^= a1;   }
      if(a0 > a2) {     a0 ^= a2;   a2 ^= a0;   a0 ^= a2;   }
      if(a0 > a3) {     a0 ^= a3;   a3 ^= a0;   a0 ^= a3;   }
      if(a1 > a2) {     a1 ^= a2;   a2 ^= a1;   a1 ^= a2;   }
      if(a1 > a3) {     a1 ^= a3;   a3 ^= a1;   a1 ^= a3;   }
      if(a2 > a3) {     a2 ^= a3;   a3 ^= a2;   a2 ^= a3;   }

      printf("\nOrdenados... \n");
      printf("%hu <= %hu <= %hu <= %hu.", a0, a1, a2, a3);

      return 0;
}
```

9.4. **Escriba un programa que solicite al usuario las coordenadas x e y de un punto del plano e indique por pantalla si ese punto está dentro de la circunferencia centrada en (x_0,y_0) y de radio r o si está sobre ella, o si está fuera de la misma.**

Como sabe, los puntos (x, y) de la circunferencia de centro (x_0, y_0) y radio r verifican que $(x - x_0)^2 + (y - y_0)^2 = r^2$. Código 9.16. recoge una posible solución.

Código 9.16. Posible solución al problema 9.4.

```
#include <stdio.h>
int main(void)
{
      double x0 , y0 , r , x , y;

      printf("Coord. centro: x0 = ");     scanf(" %lf", &x0);
      printf("Coord. centro: y0 = ");     scanf(" %lf", &y0);
      printf("radio = ");                 scanf(" %lf", &r);

      printf("Valor de x ... ");          scanf(" %lf", &x);
      printf("Valor de y ... ");          scanf(" %lf", &y);

      double p = (x - x0) * (x - x0) + (y - y0) *(y - y0);

      if(p == r * r)          printf("punto de la circunf.");
      else if(p < r * r)      printf("punto del circulo. ");
      else                    printf("punto fuera circulo");
      return 0;
}
```

9.5. ***Escriba un programa que resuelva una ecuación de segundo grado. Tendrá como entrada los coeficientes a, b y c de la ecuación y ofrecerá como resultado las dos soluciones reales o imaginarias, para el caso en que el discriminante sea negativo.***

En la Figura 5.17. tiene el flujograma que resuelve la ecuación de segundo grado. En Código 5.8. tiene una posible implementación del programa aquí solicitado.

Procure obtener un código sin mirar la solución. Recuerde que el programa deberá comprobar: (1) Que el coeficiente a sea o no igual a cero. Si es cero, deberá entonces verificar (2) que no lo sea también b: en ese caso, no habría ecuación; si b no es cero, tenemos una sencilla ecuación de primer grado y mostramos su única solución. Si a es distinto de cero, debemos calcular el discriminante; (3) si éste es negativo, entonces tendremos dos soluciones imaginarias; de lo contrario, tendremos dos soluciones reales.

9.6. Escriba un programa que solicite al usuario que introduzca por teclado un día, mes y año, y muestre entonces por pantalla el día de la semana que le corresponde.

El anterior enunciado presentaba un problema también de calendario. En aquel enunciado quedaba recogido el algoritmo para determinar si una fecha era correcta o no. Ahora el enunciado es muy breve. No recoge ninguna pista sobre cómo resolver el problema.

Dejando de lado cualquier consideración sobre la programación… ¿Cómo saber que el 15 de febrero de 1975 fue sábado? Porque si no se sabe cómo conocer los días de la semana de cualquier fecha, entonces nuestro problema no es de programación, sino de algoritmo. Antes de intentar implementar un programa que resuelva este problema, será necesario preguntarse si somos capaces de resolverlo sin programa.

Buscando en Internet es fácil encontrar información parecida a la siguiente: Para saber a qué día de la semana corresponde una determinada fecha, basta aplicar la siguiente expresión:

$$d = [(26 \times M - 2) / 10 + D + A + A / 4 + C / 4 - 2 \times C] \bmod 7$$

Donde d es el día de la semana (d = 0 es el domingo; d = 1 es el lunes,…, d = 6 es el sábado); D es el día del mes de la fecha; M es el

mes de la fecha; A es el año de la fecha; y C es la centuria (es decir, los dos primero dígitos del año) de la fecha.

A esos valores hay que introducirle unas pequeñas modificaciones: se considera que el año comienza en marzo, y que los meses de enero y febrero son los meses 11 y 12 del año anterior.

Hagamos un ejemplo a mano: ¿Qué día de la semana fue el 15 de febrero de 1975?:

- D = 15.
- M = 12: hemos quedado que en nuestra ecuación el mes de febrero es el décimo segundo mes del año anterior.
- A = 74: hemos quedado que el mes de febrero corresponde al último mes del año anterior.
- C = 19.

Con todos estos valores, el día de la semana vale 6, es decir, sábado.

Sólo queda hacer una última advertencia a tener en cuenta a la hora de calcular nuestros valores de A y de C: Si queremos saber el día de la semana del 1 de febrero de 2000, tendremos que M = 12, A = 99 y C = 19: es decir, primero convendrá hacer las rectificaciones al año y sólo después calcular los valores de A y de C. Ése día fue martes (d = 2)

Código 9.17. recoge una implementación del programa. Convendrá aclarar por qué hemos sumado 70 en la expresión de cálculo del valor de d. Suponga la fecha 2 de abril de 2001. Tendremos que D toma el valor 2, M el valor 2, A el valor 1 y C el valor 20. Entonces, el valor de d queda –27 % 7, que es igual a –6, que es un valor fuera del rango esperado. Por suerte, la operación módulo establece una relación de equivalencia entre el conjunto de los enteros y el conjunto de valores comprendidos entre 0 y el valor del módulo menos 1. Le sumamos al valor calculado un múltiplo de 7 suficientemente grande para que sea cual sea el valor de las variables, al final obtenga un resultado positivo. Así, ahora, el valor obtenido será (70 – 27) % 7 = 1, es decir, LUNES.

Código 9.17. Posible solución al problema 9.6.

```
#include <stdio.h>
int main(void)                                  {
      unsigned short D , mm , aaaa;
      unsigned short M , A , C , d;

      printf("Introduzca la fecha ... \n");
      printf("Día ... ");     scanf(" %hu", &D);
      printf("Mes ... ");     scanf(" %hu", &mm);
      printf("Año ... ");     scanf(" %hu", &aaaa);
// Valores de las variables:
// El valor de D ya ha quedado introducido por el usuario.
// Valor de M:
      if(mm < 3)                                {
            M = mm + 10;
            A = (aaaa - 1) % 100;
            C = (aaaa - 1) / 100;
      }
      else                                      {
            M = mm - 2;
            A = aaaa % 100;
            C = aaaa / 100;
      }
      printf("El día %2hu / %2hu / %4hu fue ",D, mm, aaaa);
      d = (70+(26*M-2)/10 + D + A + A/4 + C/4 – C * 2) % 7;
      switch(d)                                 {
      case 0: printf("DOMINGO");      break;
      case 1: printf("LUNES");        break;
      case 2: printf("MARTES");       break;
      case 3: printf("MIÉRCOLES");    break;
      case 4: printf("JUEVES");       break;
      case 5: printf("VIERNES");      break;
      case 6: printf("SÁBADO");
      }
      return 0;
}
```

CAPÍTULO 10

ESTRUCTURAS DE CONTROL II: ESTRUCTURAS DE REPETICIÓN O SENTENCIAS ITERADAS.

En el capítulo anterior hemos vito las estructuras de control que permiten condicionar sentencias. En este capítulo quedan recogidas las reglas sintácticas que se exige en el uso del lenguaje C para el diseño de estructuras de iteración. También veremos las sentencias de salto que nos permiten abandonar el bloque de sentencias iteradas por una estructura de control de repetición.

Introducción.

Una estructura de repetición o de iteración es aquella que nos permite repetir un conjunto de sentencias mientras que se cumpla una determinada condición.

Las estructuras de iteración o de control de repetición, en C, se implementan con las estructuras **do-while**, **while** y **for**. Todas ellas permiten la anidación de unas dentro de otras a cualquier nivel. Puede verse un esquema de su comportamiento en la Figura 10.1.

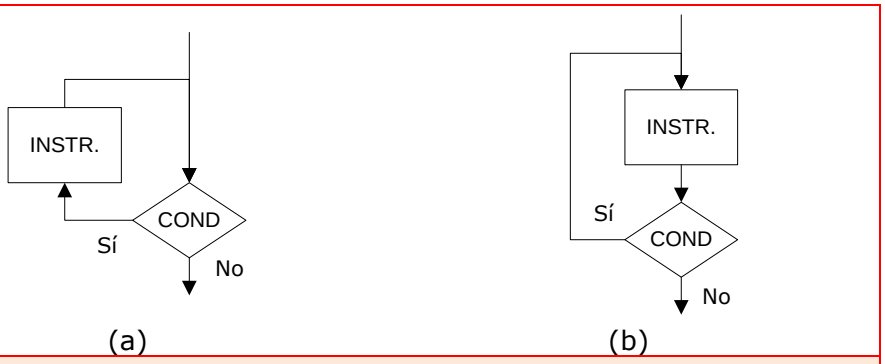


Figura 10.1. Estructuras de repetición: (a) Estructura WHILE. (b) Estructura DO – WHILE.

Estructura **while**.

La estructura **while** se emplea en aquellos casos en que no se conoce por adelantado el número de veces que se ha de repetir la ejecución de una determinada sentencia o bloque: **ninguna, una o varias**.

La sintaxis de la estructura **while** es la que sigue:

```
while(condición) sentencia;
```

donde condición es cualquier expresión válida en C. Esta expresión se evalúa cada vez, antes de la ejecución de la sentencia iterada (simple o compuesta). Puede ocurrir, por tanto, que la sentencia controlada por esta estructura no se ejecute ni una sola vez. En general, la sentencia se volverá a ejecutar una y otra vez mientras condición siga siendo una expresión verdadera. Cuando la condición resulta ser falsa, entonces el contador de programa se sitúa en la inmediata siguiente instrucción posterior a la sentencia gobernada por la estructura.

Código 10.1. Tabla de multiplicar.

```
#include <stdio.h>
int main(void)
{
	short int n,i;

	printf("Tabla de multiplicar del ... ");
	scanf(" %hd",&n);

	i = 0;
	while(i <= 10)
	{
		printf("%3hu * %3hu = %3hu\n",i,n,i * n);
		i++;
	}
	return 0;
}
```

Veamos un ejemplo sencillo. Hagamos un programa que solicite un entero y muestre entonces por pantalla la tabla de multiplicar de ese número. El programa es muy sencillo gracias a las sentencias de repetición. Puede verlo implementado en Código 10.1.

Después de solicitar el entero, inicializa a 0 la variable `i` y entonces, mientras que esa variable contador sea menor o igual que 10, va mostrando el producto del entero introducido por el usuario con la variable contador `i`. La variable `i` cambia de valor dentro del bucle de la estructura, de forma que llega un momento en que la condición deja de cumplirse; ¿cuándo?: cuando la variable `i` tiene un valor mayor que 10.

Conviene asegurar que en algún momento va a dejar de cumplirse la condición; de lo contrario la ejecución del programa podría quedarse atrapada en un **bucle infinito**. De alguna manera, dentro de la sentencia gobernada por la estructura de iteración, hay que modificar alguno de los parámetros que intervienen en la condición. Más adelante, en este capítulo, veremos otras formas de salir de la iteración.

Este último ejemplo ha sido sencillo. Veamos otro ejemplo que requiere un poco más de imaginación. Supongamos que queremos hacer un programa que solicite al usuario la entrada de un entero y que entonces muestre por pantalla el factorial de ese número. Ya se sabe la definición de factorial: $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$.

Antes de comentar el código de esta sencilla aplicación (vea Código 10.2.), conviene volver a una idea comentada capítulos atrás. Efectivamente, habrá que saber decir en el lenguaje C cómo se realiza esta operación. Pero previamente debemos ser capaces de expresar el procedimiento en castellano. En el Capítulo 5 se encuentra documentado éste y otros algoritmos de iteración.

Código 10.2. Cálculo del factorial de un número.

```
#include <stdio.h>
int main(void)
{
	unsigned short n;
	unsigned long Fact;

	printf("Introduzca el entero ... ");
	scanf(“ %hu",&n);

	printf("El factorial de %hu es ... ", n);

	Fact = 1;
	while(n != 0)
	{
		Fact = Fact * n;
		n = n - 1;
	}

	printf("%lu.", Fact);
	return 0;
}
```

n	5	4	3	2	1	0
Fact	5	20	60	120	120	120

Tabla 10.1. Valores que van tomando las variables en el programa del cálculo del factorial si la entrada ha sido 5.

El valor de la variable `Fact` se inicializa a uno antes de comenzar a usarla. Efectivamente es muy importante no emplear esa variable sin darle el valor inicial que a nosotros nos interesa. La variable `n` se inicializa con la función `scanf`.

Mientras que `n` no sea cero, se irá multiplicando `Fact` (inicialmente a uno) con `n`. En cada iteración el valor de `n` se irá decrementando en uno.

La tabla de los valores que van tomando ambas variables se muestra en la Tabla 10.1. (se supone, en esa tabla, que la entrada por teclado ha sido el número 5). Cuando la variable `n` alcanza el valor cero termina la iteración. En ese momento se espera que la variable `Fact` tenga el valor que corresponde al factorial del valor introducido por el usuario.

La iteración se ha producido tantas veces como el cardinal del número introducido. Por cierto, que si el usuario hubiera introducido el valor cero, entonces el bucle no se hubiera ejecutado ni una sola vez, y el valor de `Fact` hubiera sido uno, que es efectivamente el valor por definición ($0! = 1$).

Código 10.3. Distintos modos de escribir el mismo código.

(a)

```
while(n != 0)
{
Fact *= n;
n = n - 1;
}
```

(b)

```
while(n)
{
Fact *= n;
n = n - 1;
}
```

(c)

```
while(n)
{
Fact *= n;
n--;
}
```

(d)

```
while(n)
{
Fact *= n--;
}
```

La estructura de control mostrada admite formas de presentación más compacta y, de hecho, lo habitual será que así se presente. Puede ver, en Código 10.3., el mismo código repetido en sucesivas evoluciones, cada vez más compactadas. En (a) hemos aplicado, en la primera sentencia iterada, el operador compuesto para el producto (`Fact *= n;`), En (b) hemos simplificado la condición de permanencia en la iteración: la condición que n sea distinto de cero es la misma que la de que n sea verdadera. En (c) hemos hecho uso del operador decremento en la segunda sentencia iterada. En (d) hemos hecho uso de la precedencia de operadores y del hecho de que el operador decremento a la derecha se ejecuta siempre después de la asignación. Así las cosas, la estructura queda `while(n) Fact *= n--;`

Hacemos un comentario más sobre la estructura **`while`**. Esta estructura permite iterar una sentencia sin cuerpo. Por ejemplo, supongamos que queremos hacer un programa que solicite continuamente del usuario que pulse una tecla, y que esa solicitud no cese hasta que éste introduzca el carácter, por ejemplo, `'a'`. La estructura quedará tan simple como lo que sigue:

```
while((ch = getchar()) != 'a');
```

Esta línea de programa espera una entrada por teclado. Cuando ésta se produzca comprobará que hemos tecleado el carácter `'a'` minúscula; de no ser así, volverá a esperar otro carácter.

Un último ejemplo clásico de uso de la estructura **`while`**. El cálculo del **máximo común divisor** de dos enteros que introduce el usuario por teclado. Tiene explicado el algoritmo que resuelve este problema en el Ejercicio 5.4. y el flujograma que lo describe en la Figura 5.14. Ahora falta poner esto en lenguaje C. Lo puede ver en Código 10.4.

Hemos tenido que emplear una variable auxiliar, que hemos llamado aux, para poder hacer el intercambio de variables: que a pase a valer el valor de b y b el del resto de dividir a por b.

Código 10.4. Algoritmo de Euclides.

```
#include <stdio.h>
int main(void)
{
	short int a, b, aux;
	short int mcd;

	printf("Valor de a ... ");	scanf(" %hd",&a);
	printf("Valor de b ... ");	scanf(" %hd",&b);

	printf("El mcd de %hd y %hd es ... ", a, b);
	while(b)
	{
		aux = a % b;
		a = b;
		b = aux;
	}
	printf("%hu", a);

	return 0;
}
```

Así como queda escrito el código, se irán haciendo los intercambios de valores en las variables a y b hasta llegar a un valor de b igual a cero; entonces, el anterior valor de b (que está guardado en a) será el máximo común divisor.

Estructura **do – while**.

La estructura **do-while** es muy similar a la anterior. La diferencia más sustancial está en que con esta estructura el código de la iteración se ejecuta, **al menos, una vez**. Si después de haberse ejecutado, la condición se cumple, entonces vuelve a ejecutarse, y así hasta que la condición no se cumpla. Puede verse un esquema de su comportamiento en la Figura 10.1.(b), en páginas anteriores.

La sintaxis de la estructura es la siguiente:

```
do sentencia; while(condición);
```

Y si se desea iterar un bloque de sentencias, entonces se agrupan en una sentencia compuesta mediante llaves.

Una estructura muy habitual en un programa es ejecutar unas instrucciones y luego preguntar al usuario, antes de terminar la ejecución de la aplicación, si desea repetir el proceso (cfr. Figura 10.2.).

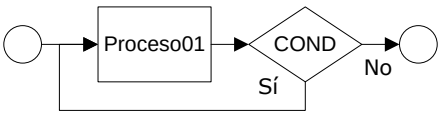


Figura 10.2. Flujograma para repetición de proceso. El proceso, desde luego, puede estar formado por una o por muchas instrucciones.

Supongamos, por ejemplo, que queremos un programa que calcule el factorial de tantos números como desee el usuario, hasta que no quiera continuar. El código ahora requiere de otra estructura de repetición, que vuelva a ejecutar el código mientras que el usuario no diga basta. Una posible codificación de este proceso sería la propuesta en Código 10.5.

La estructura **do–while** repetirá el código que calcula el factorial del entero solicitado mientras que el usuario responda con una `'s'` a la pregunta de si desea que se calcule otro factorial.

Podemos afinar un poco más en la presentación (cfr. Figura 10.3.). Vamos a rechazar cualquier contestación que no sea o `'s'` o `'n'`: si el usuario responde `'s'`, entonces se repetirá la ejecución del cálculo del factorial; si responde `'s'` el programa terminará su ejecución; y si el usuario responde cualquier otra letra, entonces simplemente ignorará la respuesta y seguirá esperando una contestación válida.

El código queda ahora de la forma propuesta en Código 10.6.

Código 10.5.

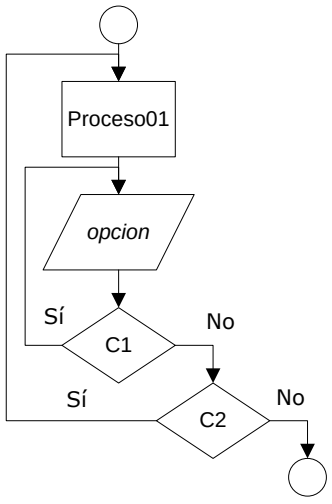
```
#include <stdio.h>
int main(void)
{
      unsigned short n;
      unsigned long Fact;
      char opcion;

      do
      {
            Fact = 1;

            printf("\nValor de n ... ");  scanf(" %hu",&n);
            printf("El factorial de %hu es ... ", n);

            while(n != 0) Fact *= n--;

            printf("%lu.", Fact);
            printf("\n\nCalcular otro factorial (s/n) ");
      }while(opcion = getchar() == 's');
}
```



C1: opcion ≠ 's' | opcion ≠ 'n'.

C2: opcion = 's'

Figura 10.3. Flujograma para repetición de proceso

Código 10.6. Nueva versión del **do-while** de Código 10.5.

```
do
{
      Fact = 1;

      printf("\n\nIntroduzca el entero ... ");
      scanf(" %hu",&n);
      printf("El factorial de %hu es ... ", n);

      while(n != 0) Fact *= n--;
      printf("%lu.", Fact);

      printf("\n\nCalcular otro factorial (s/n) ");

      do    {
            opcion = getchar();
      }while (opcion != 's' && opcion != 'n');

}while(opcion == 's');
```

Estructura **for**.

Una estructura **for** tiene una sintaxis notablemente distinta a la indicada para las estructuras **while** y **do-while**. Pero la actuación que realiza sobre el código es la misma. Con **for** podemos crear estructuras de control que se dicen "**controladas por variable**".

La sintaxis de la estructura **for** es la siguiente:

for(sentencias_1 , expresión ; sentencias_2) sentencias_3;

Donde sentencia_3 es **la sentencia que se itera**, la que queda gobernada por la estructura de control **for**.

Donde sentencias_1 es un grupo de sentencias que se ejecutan antes que ninguna otra en una estructura **for**, y siempre se ejecutan una vez y sólo una vez. Son sentencias, separadas por el operador coma, de **inicialización de variables**.

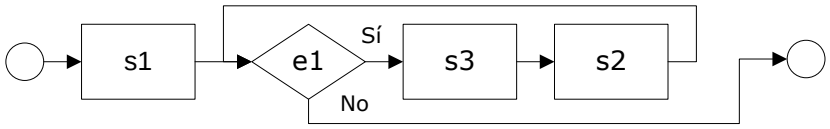


Figura 10.4. Flujograma para la estructura de control **for**.
`for(s1 ; e1 ; s2) s3;`

Donde expresión es la **condición de permanencia** en la estructura **for**. Siempre que se cumpla expresión volverá a ejecutarse la sentencia iterada por la estructura **for**.

Donde sentencias_2 son un grupo de sentencias que se ejecutan **después de la sentencia iterada** (después de sentencias_3).

El orden de ejecución es, por tanto (ver flujograma en Figura 10.4.):

1. Se inicializan variables según el código recogido en s1.
2. Se verifica la condición de permanencia e1. Si e1 es verdadero se sigue en el paso 3; si es falso entonces se sigue en el paso 6.
3. Se ejecuta la sentencia iterada s3.
4. Se ejecutan las sentencias recogidas en s2.
5. Vuelta al paso 2.
6. Fin de la iteración.

Código 10.7. Equivalencia entre estructura **for** y estructura **while**.

```
for(  sentencias_1 ;
      expresión ;
      sentencias_2)
{
      sentencias_3;
}
```

```
sentencias_1;
while(expresión)
{
      sentencia_3;
      sentencias_2;
}
```

Así, una estructura **for** es equivalente a una estructura **while** de la forma indicada en Código 10.7.

Por ejemplo, veamos en Código 10.8. un programa que muestra por pantalla los enteros pares del 1 al 100. Si queremos mejorar la presentación, podemos hacer que cada cinco pares comience una nueva fila: para so se ha añadido la sentencia condicionada marcada con la línea /*01*/.

Ya hemos dicho que en cada uno de los tres espacios de la estructura **for** destinados a recoger sentencias o expresiones, pueden consignarse una expresión, o varias, separadas por comas, o ninguna. Y la sentencia iterada mediante la estructura **for** puede tener cuerpo, o no. Veamos por ejemplo, el cálculo del factorial de un entero mediante una estructura **for**:

```
for(Fact = 1 ; n ; Fact *= n--);
```

Esta estructura **for** no itera más que la sentencia punto y coma. Toda la trasformación que deseamos realizar queda en la expresión del cálculo del factorial mediante la expresión `Fact *= n--`.

Código 10.8.

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

      for(i = 2 ; i <= 100 ; i += 2)
      {
            printf("%5hd",i);
/*01*/      if(i % 10 == 0) printf("\n");
      }

      return 0;
}
```

En este caso especial, el punto y coma debe ponerse: **toda estructura de control actúa sobre una sentencia**. Si no queremos, con la estructura **for**, controlar nada, entonces la solución no es no poner nada, sino poner una sentencia vacía.

Todos los ejemplos que hasta el momento hemos puesto en la presentación de las estructuras **while** y **do – while** se pueden rehacer con una estructura **for**. En algunos casos es más cómodo trabajar con la estructura **for**; en otros se hace algo forzado. En Código 10.9. se recoge el código equivalente al presentado en Código 10.5., con el programa que muestra la tabla de multiplicar. En Código 10.10 se recoge el código equivalente a Código 10.4., para el algoritmo de Euclides en el cálculo del máximo común divisor de dos enteros. Y la sentencia que bloquea la ejecución del programa hasta que se pulse la tecla 'a' queda, con una estructura **for**, de la siguiente forma:

```
for( ; ch != 'a' ; ch = getchar());
```

Código 10.9. Para la Tabla de multiplicar con una estructura **for**.

```
for(i = 0 ; i <= 10 ; i++)
{
      printf("%3hu * %3hu = %3hu\n", i, n , i * n);
}
```

Código 10.10. Algoritmo de Euclides con una estructura **for**.

```
for( ; b ; )
{
      aux = a % b;
      a = b;
      b = aux;
}
printf("%hu", a);
```

Las reglas de la programación estructurada.

Supongamos que deseamos hacer un programa que reciba del usuario un valor entero y muestre entonces por pantalla un mensaje según ese número introducido sea primo o compuesto.

Para determinar si el número N introducido es realmente primo lo que podemos hacer es intentar encontrar algún entero entre 2 y N – 1 que divida a N. Asignamos, pues, a una variable (la llamamos d) el valor 2; y mientras que no se verifique que N % d es igual a cero, vamos incrementando de uno en uno el valor de esa variable d. Si la variable d llega al valor de la variable N entonces es evidente que no hemos encontrado divisor alguno y que nuestro valor de entrada es primo.

Nuestro proceso descrito presenta una iteración (incremento sucesivo de la variable d) y dos condiciones de interrupción: o encontrar un valor de d, comenzando en 2 que verifique que N % d == 0; ó llegar con d al valor de la variable N.

Dese cuenta, por ejemplo, de que si en la primera iteración (para d = 2) tenemos que N % d == 0 es falso, eso sólo significa que N no es par: no puede concluir que el N es primo: debe seguir buscando un posible divisor. Y dese también cuenta de que si en cualquier iteración se verifica que N % d == 0 es verdadero, entonces es cierto que N es compuesto y no es necesario continuar con la búsqueda de nuevos divisores.

Con esta descripción, quizá lo primero que a uno se le ocurre es iniciar el flujograma de forma parecida a como se muestra en la Figura 10.5.

La vía marcada con el número 1 se debería seguir en el caso de que el número N introducido fuera compuesto: se ha encontrado un valor en d para el que se verifica que N % d es falso, es decir, es igual a cero, lo que implica que d es divisor de N y, por tanto, que N es compuesto.

La vía marcada con el número 2 se debería seguir en el caso de que el número N introducido fuera primo: se ha llegado al valor de d igual al

valor de N sin que ningún valor previo hay logrado dividir a N de forma exacta: todos los restos han sido verdaderos, es decir, distintos de cero.

Pero quien haya comenzado así el flujograma está a punto de cometer una violación de las reglas básicas de la programación estructurada ya presentadas en el Capítulo 5. El flujograma de la Figura 10.5. se parece peligrosamente al erróneo de la Figura 5.5. mostrado en aquel Capítulo. ¿Cómo resolvemos entonces ese problema? ¿Cuál es el algoritmo correcto que logra respetar las reglas de la programación estructurada?

Lo veremos más adelante. Antes, planteamos otro problema. Queremos hacer ahora un programa que vaya solicitando al usuario hasta un total de diez valores enteros positivos, y que vaya calculando la suma de todos ellos para mostrar al final la media aritmética. Si el usuario dispone de menos de una decena de datos, entonces puede indicar al programa que se han terminado los valores ingresando el valor cero.

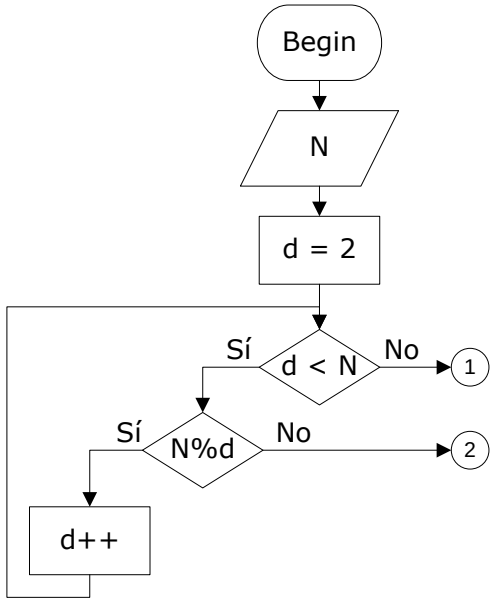


Figura 10.5. Inicio de un posible flujograma que exprese el algoritmo que determina si un valor de entrada N es primo o compuesto.

Este programa, igual que el anterior, requerirá de una estructura de iteración, que vaya solicitando el número de valores de entrada y que termine cuando se hayan introducido 10 valores. De nuevo, y con esta descripción, quizá lo primero que a uno se le podría ocurrir es iniciar el flujograma de forma parecida a como se muestra en la Figura 10.6.

Este flujograma tiene un aspecto parecido al anterior, pero ahora entre los dos diamantes de decisión se ha añadido una nueva sentencia: la de ingreso de un nuevo valor para `N` por teclado. De nuevo el algoritmo inicialmente esbozado con este flujograma inconcluso no es viable, porque viola las reglas de la programación estructurada (está explicado en el Capítulo 5, en torno a esa Figura 5.5. ya referenciada).

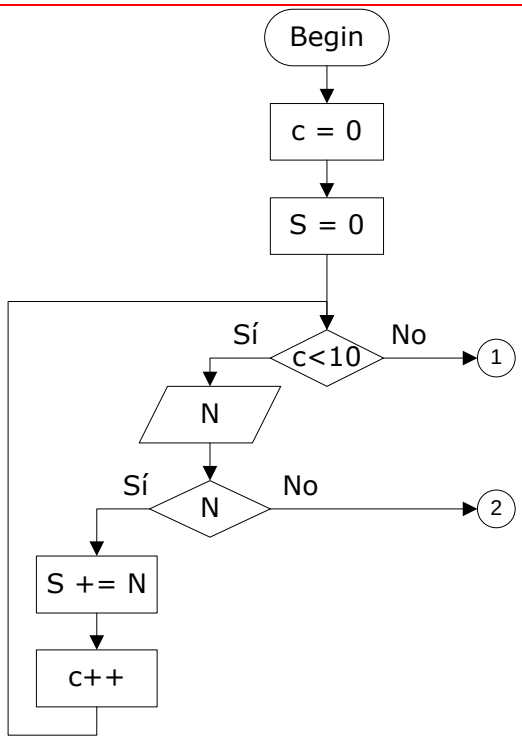


Figura 10.6. Inicio de un posible flujograma que exprese el algoritmo que calcula la media de hasta 10 posibles valores de entrada.

Este problema lo tendremos siempre que en una estructura de iteración haya dos o más condiciones de permanencia que se evalúen en momentos distintos del proceso. Hay que evitar siempre esa situación, buscando juntar la decisión en un sólo punto del algoritmo: vendría a ser como una operación de fusión de diamantes en uno sólo dentro de cada bucle de iteración.

En el caso del programa que determina si un entero introducido por teclado es primo o compuesto, la operación de fusión parece sencilla de realizar, porque ambos diamantes está juntos y no existe operación alguna por medio. Podríamos escribir la siguiente sentencia, una vez que ya tenemos introducido el valor de `N`:

```
for(d = 2 ; d < N && N % d ; d++);
```

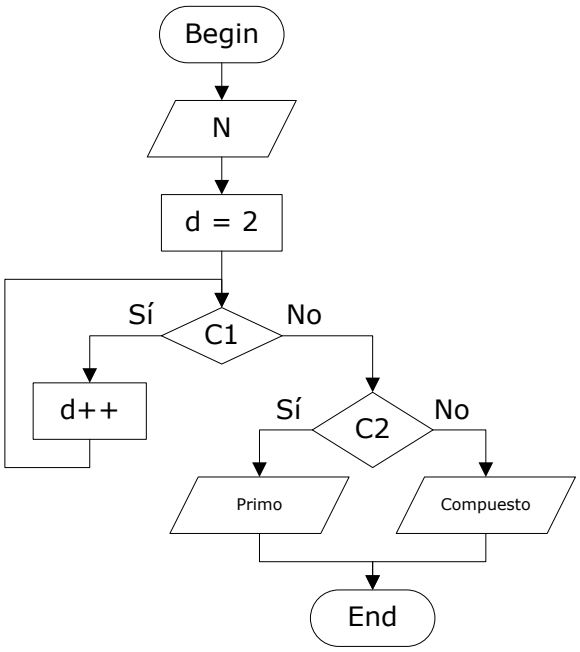


Figura 10.7. Algoritmo para determinar si un entero ingresado por teclado es primo o compuesto.

La condición C1 es: `(d < N && N % d)`. La condición C2 es: `(d == N)`.

Es una estructura de iteración **for** sin cuerpo: por eso termina con una sentencia iterada vacía: el punto y coma sin más. En este código, el programa aumenta de uno en uno el valor de la variable d hasta que se llegue al valor de N, o hasta que se encuentre un divisor. Por ambos motivos el programa abandona la ejecución de esa iteración. El flujograma quedaría entonces como se muestra en la Figura 10.7.

El segundo caso tiene inicialmente más complejidad, porque entre los dos diamantes de decisión hay una sentencia. Pero podemos juntar la decisión si hacemos lo que se propone en el flujograma de la Figura 10.8.

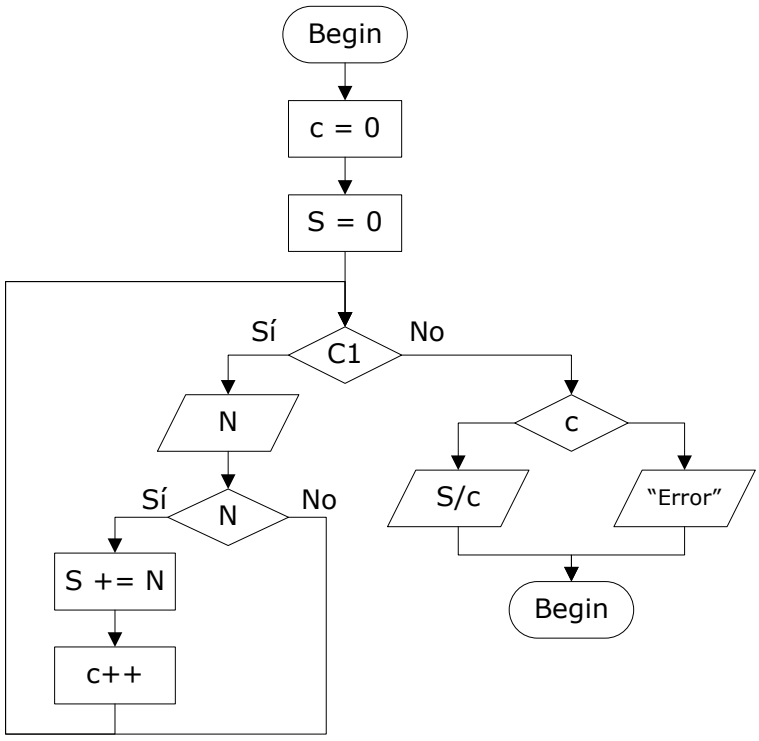


Figura 10.8. Algoritmo para calcular la media de hasta 10 valores introducidos por teclado.

La condición C1 es: `(c < 10 && N)`.

Con estos ejemplos hemos querido mostrar que existe el peligro de no respetar todas las reglas de la programación estructurada, y que con frecuencia es relativamente fácil proponer soluciones que violan esas reglas. En concreto, es importante vigilar que cuando nuestros programas caen en una estructura de iteración, únicamente sea posible abandonar la iteración por un único camino. Siempre que propongamos dos vías de escape del bucle habremos violado las reglas.

Sentencias de salto.

Existen en C dos sentencias que modifican el orden del flujo de instrucciones dentro de una estructura de iteración. Son la sentencia **break**, que presentamos ahora; y la sentencia **continue**, que veremos en el siguiente epígrafe. Ambas sentencias sólo pueden aparecer entre las sentencias de una estructura de iteración; y ambas sentencias deben aparecer siempre condicionadas. No es fácil, como veremos, justificar su uso desde la perspectiva de la programación estructurada. No aprenda, pues, a hacer de estas dos sentencias un uso indiscriminado.

Existe en C una tercera sentencia de salto, que puede insertarse en cualquier parte del programa: es la sentencia **goto**, que ni veremos ni conviene que usted la vea por su cuenta, porque está prohibida pues viola abiertamente las reglas de la programación estructurada.

Mire, por ejemplo, las siguientes tres recomendaciones que se recogen en "*MISRA-C 2004. Guidelines for the use of the C language in critical systems*"[1], sobre el uso de esas tres sentencias y que son palabras reservadas del Lenguaje C:

- "*Rule 14.4. (required): The* **goto** *statement shall not be used.*"
- "*Rule 14.5. (required): The* **continue** *statement shall not be used.*"

[1] La misión de MISRA (*The Motor Industry Software Reliability Association*), según se declara en su sitio web (http://www.misra.org.uk/), es: "*To provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software.*"

- "*Rule 14.6. (required): For any iteration statement there shall be at most one* **break** *statement used for loop termination.*"

Así que ya ve cómo se nos recomienda no hacer uso en ningún caso de la sentencia **continue**; y cómo se nos recomienda hacer un uso muy restringido de la sentencia **break**. Respecto a la sentencia goto la prohibición es mucho más severa como verá más adelante.

Sentencia de salto **break**.

Una sentencia **break** dentro de un bloque de instrucciones de una estructura de iteración interrumpe la ejecución de las restantes sentencias iteradas y colocadas más abajo y abandona la estructura de control, asignando al contador de programa la dirección de la siguiente sentencia posterior a la llave que cierra el bloque de sentencias de la estructura de control.

Es muy fácil violar, con esta sentencia, las reglas de la programación estructurada. Vea, por ejemplo, una posible solución a los dos problemas planteados un poco más arriba en este capítulo: el primero, en Código 10.11., es un programa que determina si un entero introducido por teclado es primo o compuesto; el segundo, en Código 10.12., es un programa que calcula la media de los valores introducidos por el usuario hasta que introduce el valor 0 y nunca más de 10.

Ambas soluciones aquí propuestas son sintácticamente correctas, pero presentan una violación de reglas: ofrecen dos salidas dentro del bucle de iteración. Como de hecho no existe una prohibición dictada por ningún estándar internacional, damos por bueno su uso. De todas formas, sí existen recomendaciones, como la de MISRA-C anteriormente mostrada que recomiendan no incluir más de una sentencia **break** dentro de una iteración.

Código 10.11. Primo o Compuesto.

```
#include <stdio.h>
#include <stdint.h>
int main(void)
{
      uint32_t d, n;

      printf("Valor de n ... ");    scanf(" %ld", &n);
      for(d = 2 ; d < n ; d++)
      {
            if(n % d == 0) break;
      }

      if(d == n)        printf("PRIMO\n");
      else              printf("COMPUESTO\n");
      return 0;
}
```

Código 10.12. Cálculo de la media.

```
#include <stdio.h>
#include <stdint.h>
int main(void)
{
      uint16_t c , n;
      uint32_t s;

      for(s =0 , c = 0 ; c < 10 ; c++)
      {
            printf("Entrada ... ");    scanf(" %hu", &n);
            if(!n) break;
            s += n;
      }

      if(c)       printf("Media: %lf\n", (double)s / c);
      else        printf("No ha introducido valores\n");
      return 0;
}
```

Vaya pues con cuidado al usar este tipo de estructuras: no siempre es afortunado hacerlo. Habitualmente esta sentencia **break** siempre podrá evitarse con un diseño distinto de la estructura de control. Según en qué ocasiones, el código adquiere mayor claridad si se utiliza el **break**, otras veces es mejor pensar un poco más otra forma de llegar a la solución, sin violar las reglas.

De todas formas, y al margen de estas restricciones impuestas por las reglas de la programación estructurada, sí existe una estructura derivada básica en los flujogramas estructurados que necesita de esta sentencia para crear el correspondiente código en C: son las estructuras híbridas (cfr. Figura 5.3.(b), en el Capítulo 5). Son iteraciones que ni son de tipo While (que primero evalúan la condición y luego ejecutan las sentencias del bucle), ni de tipo Do-While (que primero ejecutan las sentencias iteradas y sólo después evalúan la condición de permanencia). Las híbridas son estructuras que ejecutan sentencias antes y después de evaluar la permanencia en la iteración.

Con la sentencia **break** es posible definir estructuras de control sin condición de permanencia, o con una condición que es siempre verdadera. Así se logra definir una estructura de iteración que no introduce una posible salida del bucle en el inicio o al final de las sentencias del bloque iterado. Desde luego, una estructura iterada sin condición de permanencia crea un bucle infinito del que no se podría salir, salvo que entre las sentencias iteradas se encontrase una sentencia **break** condicionada. Se logra así insertar un camino de salida dentro de la iteración, sin llegar por ello a tener dos vías de escape, técnicamente prohibidas en las reglas de programación estructurada.

Por ejemplo, en Código 10.13 se recoge unas sentencias que solicitan al usuario valores enteros pares hasta que éste introduce un impar, y entonces muestran por pantalla la suma de todos esos valores pares introducidos.

Código 10.13. Suma de los valores pares introducidos

```
long int suma = 0, num;
do
{
      printf("Introduzca un nuevo sumando ... ");
      scanf(" %ld",&num);
      if(num % 2 != 0) break;
      suma += num;
}while(1);

printf("La suma es ... %ld\n", suma);
```

En esta estructura **do-while**, la condición de permanencia es verdadera siempre, por definición, puesto que es un literal diferente de cero. Pero hay una sentencia **break** condicionada dentro del bloque iterado. El código irá guardando en la variable suma la suma acumulada de todos los valores introducidos por teclado mientras que esos valores sean enteros pares. En cuanto se introduzca un entero impar se abandona la estructura de iteración y se muestra el valor sumado.

El flujograma de este último ejemplo queda recogido en la Figura 10.9. Vea que, en realidad, estamos implementando la estructura derivada híbrida. En este caso, por tanto, la sentencia **break** sí respeta las reglas de la programación estructurada.

Código 10.14. Estructura **for** sin condición de permanencia.

```
for( ; ; )
{
      ch = getchar();
      printf("Esto es un bucle infinito\n");
      if(ch == 'a') break;
}
```

También se pueden tener estructuras **for** sin ninguna expresión recogida entre sus paréntesis. Puede ver un ejemplo en Código 10.14., que repetirá la ejecución del código hasta que se pulse la tecla `'a'`, y que ocasionará la ejecución del **break**.

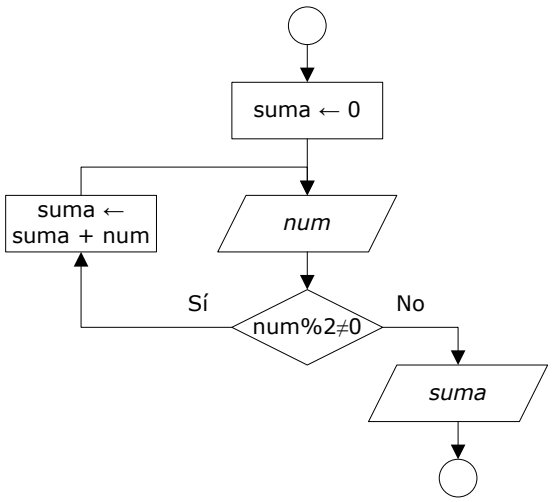


Figura 10.9. Flujograma con una estructura híbrida.

Sentencia de salto **continue**.

Ya ha quedado recogida la recomendación sobre el no uso de esta sentencia (MISRA-C). De todas formas, como esa prohibición no supone un estándar universalmente aceptado, convendrá presentar brevemente esta sentencia, porque su uso es técnicamente permitido.

Una sentencia **continue** dentro de un bloque de instrucciones de una estructura de iteración interrumpe la ejecución de las restantes sentencias iteradas y vuelve al inicio de las sentencias de la estructura de control, si es que la condición de permanencia así lo permite.

Veamos, por ejemplo, el programa antes presentado que muestra por pantalla los 100 primeros enteros pares positivos. Otro modo de

resolver ese programa podría ser el mostrado en Código 10.15. Cuando el resto de la división entera entre el contador `i` y el número 2 es distinto de cero, entonces el número es impar y se solicita que se ejecute la sentencia **`continue`**. Entonces se abandona la ejecución del resto de las sentencias del bloque iterado y, si la condición de permanencia en la iteración así lo permite, vuelve a comenzar la iteración por su primera sentencia del bloque iterado.

Código 10.15.

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

      for(i = 1 ; i <= 100 ; i++)
      {
            if(i % 2) continue;
            printf("%4hd\t" , i);
      }

      return 0;
}
```

El uso de la sentencia **`continue`** introduce una diferencia entre el código expresado con una estructura **`for`** y el mismo código aparentemente bien expresado con una estructura **`while`**. Mostrábamos la equivalencia de ambas estructuras en Código 10.7. Pero esa equivalencia ya no es válida cuando dentro del bloque de código iterado se encuentra una sentencia de salto **`continue`**. Vea, sino, los dos bloques iterados de Código 10.16., que copia la iteración de Código 10.15. (con una estructura **`for`**) y luego lo repite con una estructura **`while`**. ¿Qué ocurre, en el ejemplo de Código 10.16.(b), cuando se ejecuta la sentencia **`continue`**?: pues que no se permite, en esa iteración, la ejecución de la

sentencia que incrementa en uno el valor de `i`. Quedamos, entonces, atrapados en un bucle infinito del que no hay escapatoria. Porque la estructura **`for`** tiene una sentencia (`i++`) que se ejecuta en cualquier caso; pero no ocurre así con la estructura **`while`**, que no tiene un espacio reservado que inserta un último bloque de sentencias a ejecutar después de todo el bloque de iteración.

Código 10.16. ¿Equivalencia? entre estructura **`for`** y estructura **`while`**.

(a)

```
for(i = 1 ; i <= 100 ; i++)
{
      if(i % 2) continue;
      printf("%4hd\t" , i);
}
```

(b)

```
i = 1;
while(i < 100)
{
      if(i % 2) continue;
      printf("%4hd\t" , i);
      i++;
}
```

Palabra reservada **goto**.

Edsger W. Dijkstra (Premio Turing en 1972) publicó en 1968 una breve comunicación que ha hecho fortuna y que se ha visto como el paso definitivo hacia el rechazo de la sentencia **goto**, presente en casi todos los lenguajes de programación. El mismo título de esa breve carta deja clara su postura. Reza así: "*Go To Statement Considered Harmful*". Siempre hay programadores que la usan, e incluso hay autoridades indiscutidas del mundo de la programación que aceptan su uso en determinadas condiciones excepcionales. Pero en términos generales se puede afirmar que la sentencia **goto** no debe ser usada.

La sentencia de salto **goto** no respeta las reglas de la programación estructurada. Todo código que se resuelve empleando una sentencia **goto** puede encontrar una solución mejor con una estructura de iteración.

Si alguna vez ha programado con esta palabra, la recomendación es que se olvide de ella. Si nunca lo ha hecho, la recomendación es que la ignore y no se la aprenda.

Y, eso sí: hay que acordarse de que esa palabra es clave en C: no se puede generar un identificador con esa cadena de letras.

Variables de control de iteraciones.

Hasta el momento, hemos hecho siempre uso de las variables para poder almacenar valores concretos. Pero pueden tener otros usos. Por ejemplo, podemos hacer uso de ellas como chivatos o centinelas: su información no es el valor concreto numérico que puedan tener, sino la de una situación del proceso.

Como vamos viendo en los distintos ejemplos que se presentan, el diseño de bucles es tema delicado: acertar en la forma de resolver un problema nos permitirá llevar solución a muy diversos problemas. Pero, como estamos viendo, es importante acertar en un correcto control del bucle: decidir bien cuando se ejecuta y cuando se abandona.

Existen dos formas habituales de controlar un bucle o iteración:

1. **Control mediante variable contador**. En ellos una variable se encarga de contar el número de veces que se ejecuta el cuerpo del bucle. Esos contadores requieren una inicialización previa, externa al bucle, y una actualización en cada iteración para llegar así finalmente a un valor que haga falsa la condición de permanencia. Esa actualización suele hacerse al principio o al final de las sentencias iteradas. Hay que garantizar, cuando se diseña una iteración, que se llega a una situación de salida.
2. **Control por suceso**. Este tipo de control acepta, a su vez, diferentes modalidades:

   2.1. **Consulta explícita**: El programa interroga al usuario si desea

continuar la ejecución del bucle. La contestación del usuario normalmente se almacena en una variable tipo ***char*** o ***int***. Es el usuario, con su contestación, quien decida la salida de la iteración.

2.2. **Centinelas**: la iteración se termina cuando la variable de control toma un valor determinado. Este tipo de control es usado habitualmente para introducir datos. Cuando el usuario introduce el valor que el programador ha considerado como valor de fin de iteración, entonces, efectivamente, se termina esa entrada de datos. Así lo hemos visto en el ejemplo de introducción de números, en el programa del cálculo de la media, en el que hemos considerado que la introducción del valor cero era entendido como final de la introducción de datos.

2.3. **Banderas**: Es similar al centinela, pero utilizando una variable lógica que toma un valor u otro en función de determinadas condiciones que se evalúan durante la ejecución del bucle.

Normalmente la bandera siempre se puede sustituir por un centinela, pero se emplea en ocasiones donde la condición de terminación resulta compleja y depende de varios factores que se determinan en diferentes puntos del bucle.

En los ejercicios planteados al final del capítulo podrá encontrar algunos ejemplos de variables bandera y centinela.

Recapitulación.

Hemos presentado las estructuras de control iterativas creadas con las estructuras **`for`**, **`while`** y **`do – while`**, y las modificaciones a las estructuras que podemos introducir gracias a las palabras reservadas **`break`** y **`continue`**.

Ejercicios.

10.1. *Calcular la suma de los pares positivos menores o igual a 200.*

Con una estructura **for**, incrementando de dos en dos el valor de la variable de control a la que se le asigna un valor inicial igual a 2, el código resulta sencillo. Simplemente debemos definir una variable que guarde la suma acumulada de todos los valores que va tomando la variable de control.

El código puede ser algo así como el que se muestra en Código 10.17.

Código 10.17. Solución al problema 10.1.

```
#include <stdio.h>
int main(void)
{
	long suma;
	short i;

	for(suma = 0 , i = 2 ; i <= 200 ; i += 2)
		suma += i;

	printf("esta suma es ... %ld.\n" , suma);

	return 0;
}
```

10.2. *Indique la salida que, por pantalla, ofrecen las sentencias de Código 10.18.*

Código 10.18. Enunciado al problema 10.2.

```
short a;
while(a) scanf(“ %hd", &a);
printf("El valor de a es ... %hd", a);
```

La salida por pantalla es: **El valor de a es ... 0**

10.3. ***Indique la salida que, por pantalla, ofrecen las sentencias de Código 10.19.***

Código 10.19. Enunciado al problema 10.3.

```
#include <stdio.h>
int main(void)
{
	short a = 210, b = 2;
	while(b <= a)
	{
		if(a % b) b++;
		else
		{
			printf("%hd\t" , b);
			a /= b;
		}
	}
	return 0;
}
```

Este programa lo que hace es obtener los factores primos del valor entero asignado a la variable a. En este caso, pues, la salida por pantalla será la secuencia de enteros:

2 3 5 7.

Para ver el proceso que sigue un programa es muy útil seguir las trazas de las variables. En este caso, las variables toman los valores iniciales a = 210 y b = 2. Y mientras que b sea menor o igual que a se realiza la siguiente operación: si el valor de b NO divide al valor de a (es decir, si el resto es distinto de cero, lo que en C se interpreta como verdadero), entonces se incrementa el valor de b. Si SÍ lo divide, entonces se muestra ese valor de b por pantalla y se cambia el valor de a, que pasa a ser ahora el cociente entre a y b: no se incrementa, en ese caso, en valor de b.

La traza de las variables será, por tanto, la siguiente:

a	210	105	105	35	35	7	7	1
b	2	2	3	3	5	5	7	7
	*		*		*		*	

Donde hemos marcado con un asterisco los valores en los que se ejecuta la función `printf`. El proceso termina cuando se realiza la última división y el valor de a (1) pasa a ser menor que el valor de b (7).

10.4. Muestre la salida que, por pantalla, ofrecen las sentencias de Código 10.20.

Código 10.20. Enunciado al problema 10.4.

```
#include <stdio.h>
int main(void)
{
      short a = 29;
```

Código 10.20. (Cont.)

```
      while(a)
      {
            a /= 2;
            printf("%5hd", a);
      }

      return 0;
}
```

La salida por pantalla será la siguiente:

```
   14    7    3    1    0
```

10.5. ***Hacer un programa que calcule la media de todos los valores que introduzca el usuario por consola. El programa debe dejar de solicitar valores cuando el usuario introduzca el valor 0.***

Posible solución en Código 10.21.

Código 10.21. Posible solución al problema 10.5.

```
#include <stdio.h>
int main(void)
{
      long suma;
      short i, num;

      for(i = 0, suma = 0 ; ; i++)
      {
            printf("Introduzca número ... ");
            scanf(“ %hd",&num);
```

Código 10.21. (Cont.).

```
            suma += num;
            if(num == 0) break;
      }

      if(i == 0)
      {
            printf("No se han introducido enteros.");
      }
      else
      {
            printf("La media es ... %.2f.",(float)suma / i);
      }

      return 0;
}
```

En la estructura **for** se inicializan las variables `i` y `suma`. Cada vez que se introduce un nuevo entero se suma al acumulado de todas las sumas de los enteros anteriores, en la variable `suma`. La variable `i` lleva la cuenta de cuántos enteros se han introducido; dato necesario para calcular, cuando se termine de introducir enteros, el valor de la media. La estructura **for**, como puede ver, no tiene una condición de permanencia: hemos creado una estructura híbrida.

10.6. Mostrar por pantalla todos los caracteres ASCII.

Una posible solución sería la propuesta en Código 10.22. El programa propuesto va mostrando todos los caracteres, uno por uno, junto con su código ASCII en hexadecimal y en decimal.

Código 10.22. Posible solución al problema 10.6. (¿Descubre algo peligroso en la condición de permanencia?)

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

	for(a = 0 ; a <= 255 ; a++)
		printf("%3c - %hX - %hd\n" , a , a , a);

	return 0;
}
```

Advertencia importante: De forma intencionada hemos dejado el código de este problema con un error grave. Aparentemente todo está bien. De hecho no existe error sintáctico alguno. El error se verá en tiempo de ejecución, cuando se compruebe que se ha caído en un bucle infinito.

Para comprobarlo basta observar la condición de permanencia en la iteración gobernada por la estructura **for**, mediante la variable que hemos llamado a: `a <= 255`. Si se tiene en cuenta que la variable a es de tipo **unsigned char**, no es posible que la condición indicada llegue a ser falsa, puesto que, en el caso de que a alcance el valor 255, al incrementar en 1 su valor, incurrimos en *overflow*, y la variable cae en el valor cero: $(255)_{10} = (1111\,1111)_2$; si solo tenemos ocho dígitos, entonces… $(1111\,1111)_2 + (1)_2 = (0000\,0000)_2$, pues no existe el bit noveno, donde debería haber quedado codificado un dígito 1.

Sirva este ejemplo para advertir de que a veces puede aparecer un bucle infinito de la manera más insospechada. La tarea de programar no está exenta de sustos e imprevistos que a veces obligan a dedicar un tiempo no pequeño a buscar la causa de un error.

Un modo correcto de codificar este bucle queda en Código 10.22.bis.

Código 10.22. bis.

```
for(a = 0 ; a < 255 ; a++)
      printf("%3c - %hX - %hd\n" , a , a , a);
printf("%3c - %hX - %hd\n" , a , a , a);
```

Y así, cuando llegue al valor máximo codificable en la variable a, abandona el bucle e imprime, ya fuera de la iteración, una última línea de código, con el valor último.

10.7. Solicitar al usuario un valor entero por teclado y mostrar entonces, por pantalla, el código binario en la forma en que se guarda el número en la memoria.

Solución propuesta en Código 10.23. La variable `Test` se inicializa al valor hexadecimal 8000 0000 (con un 1 en el bit más significativo y en cero en el resto). Posteriormente sobre esta variable se va operando un desplazamiento a derecha de un bit cada vez. Así, siempre tenemos localizado un único bit en la codificación de la variable `Test`, hasta que este bit se pierda por la parte derecha en un último desplazamiento.

Antes de cada desplazamiento, se realiza la operación and a nivel de bit entre la variable `Test`, de la que conocemos donde está su único bit, y la variable de la que queremos conocer su código binario.

En el principio, tendremos así las dos variables:

```
Test         1000 0000 0000 0000 0000 0000 0000 0000
a            xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
Test & a     x000 0000 0000 0000 0000 0000 0000 0000
```

Tenemos certeza de que la operación and dejará un cero en todos los bits (menos el más significativo) de `Test & a`, porque `Test` tiene todos

esos bits a cero. La operación dará un valor distinto de cero únicamente si en ese bit se encuentra un 1 en la variable a.

Código 10.23. Posible solución al problema 10.7.

```
#include <stdio.h>
int main(void)
{
	signed long a;
	unsigned long Test;
	char opcion;
	do
	{
		Test = 0x80000000;
		printf("\n\nIndique el entero ... ");
		scanf(“ %ld", &a);
		while(Test)
		{
			Test & a ? printf("1") : printf("0");
			Test >>= 1;
		}
		printf("\n¿Desea introducir otro entero? ... ");
		do
			opcion = getchar();
		while (opcion != 's' && opcion != 'n');
	}while(opcion == 's');

	return 0;
}
```

Al desplazar Test un bit a la derecha tendremos la misma situación que antes, pero ahora el bit de a testeado será el segundo por la derecha.

```
Test         0100 0000 0000 0000 0000 0000 0000 0000
a            xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
Test & a     0x00 0000 0000 0000 0000 0000 0000 0000
```

Y así sucesivamente, imprimimos un 0 cuando la operación and sea 0, e imprimiremos un 1 cuando la operación and dé un valor distinto de 0.

El programa que hemos presentado permite al usuario ver el código de tantos números como quiera introducir por teclado: hay una estructura **do-while** que anida todo el proceso.

10.8. ***Escriba un programa que solicite al usuario un entero positivo e indique si ese número introducido es primo o compuesto.***

El procedimiento más usado para determinar esa propiedad de los números, cuando estos son menores de diez millones, es la búsqueda exhaustiva de un entero que resulte ser divisor propio del número analizado.

Este problema ya ha sido tratado en páginas previas de este capítulo. En Código 10.24. mostramos una posible implementación.

Código 10.24. Posible solución al problema 10.8.

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

int main(void)
{
      uint32_t n;
      uint16_t d;

      printf("Valor de n ... ");    scanf("%ld", &n);

      for(d = 2 ; d <= sqrt(n) && n % d ; d++);

      printf("%El entero ld: %s.\n\n" ,
                  n , n % d ? "PRIMO" : "COMPUESTO");
      return 0;
}
```

La condición de permanencia del **for** requiere una explicación. Todo entero compuesto verifica que al menos uno de esos divisores es menor que su raíz cuadrada. Eso es sencillo de demostrar por reducción al absurdo: supongamos que tenemos un entero n y que tiene dos factores distintos de 1 y de n; por ejemplo a y b, es decir, $n = a \times b$. Supongamos que ambos factores son estrictamente mayores que la raíz cuadrada de n. ($a < \sqrt{n}$ y $b < \sqrt{n}$) Tendremos entonces que $n = a \times b$ es estrictamente menor que $\sqrt{n} \times \sqrt{n}$ que es igual a n. Y así, habríamos llegado al absurdo de que n es estrictamente menor que n.

Por lo tanto, para saber si un entero n es primo o compuesto, basta buscar divisores entre 2 y $\sqrt{n}$. Si en ese rango no los encontramos, entonces podemos concluir que el entero es primo.

10.9. ***Presentar un algoritmo que reciba un entero y muestre todos los enteros que lo dividen. Tenga en cuenta que todos los divisores de un entero excepto él mismo, se encuentran entre el 1 y la mitad del entero del que se buscan los divisores.***

Con las indicaciones del enunciado, el algoritmo tiene fácil presentación:

ENUNCIADO: Mostrar los divisores de un entero n.

```
1.    Inicializar Variables:
      1.1.  El usuario del programa introduce el valor de n.
      1.2.  div ← 2.
2.    Mostrar 1
3.    WHILE div ≤ n/2
      3.1.  IF n mod div = 0
            THEN
                  Mostrar div
            END IF
      3.2.  i ← i + 1.
      END WHILE
4.    Mostrar N
```

Código 10.25. Posible solución al problema 10.9.

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

int main(void)
{
	long N;
	long div;

	printf("Valor de N ... ");	scanf(" %ld", &N);
	printf("Los divisores del numero %ld son ...\n", N);
	printf("%8ld", 1);
	for(div = 2 ; div <= N / 2 ; div ++)
		if(!(N % div)) printf("%8ld", div);
	printf("%8ld", N);

	return 0;
}
```

Código 10.25. recoge el código del algoritmo. La condición recogida en la estructura condicional **if** es `!(N % div)`: es decir, `N % div == 0`: si esta última expresión es verdadera, entonces `N % div` será igual a cero, es decir, será falso: y por tanto, su negación será verdadero.

10.10. ***Se llama número perfecto a aquel que es igual a la suma de todos sus divisores excepto él mismo. Por ejemplo, el número 6 tiene como divisores los enteros 1, 2 y 3, que, efectivamente, suman 6; el número 28 tiene como divisores el 1, 2, 4, 7 y 14, que de nuevo efectivamente, suman 28; el 18 tiene como divisores el 1, 2, 3, 6 y 9, que suman 21, por lo que este último número no es perfecto; los dos primeros sí lo han sido.***

Presentar un algoritmo que muestre los números perfectos entre el 1 y un millón.

Este algoritmo requiere buscar todos los divisores de un entero. Ya hemos visto más arriba cómo se hace esto. Ahora, además de buscar los divisores, lo que debemos hacer es sumarlos.

Aunque es evidente, conviene quizá advertir que este programa no requiere la interacción con ningún usuario: todos los datos del problema está claros desde su inicio, y no necesitamos, de acuerdo con el enunciado propuesto, ninguna información adicional que nos haya de facilitar nadie.

Código 10.26. Posible solución al problema 10.10.

```
#include <stdio.h>
#define _LIMITE 1000000
int main(void)
{
	long suma;
	long i, num;
	for(num = 2 ; num < _LIMITE ; num++)
	{
		for(i = 1, suma = 0 ; i <= num / 2 ; i++)
			if(num % i == 0) suma += i;
		if(num == suma)
			printf("%hd es perfecto.\n" , num);
	}
	return 0;
}
```

El Flujograma de este algoritmo puede verse en la Figura 10.10. La implementación en C puede verse en código 10.26.

La variable num recorre todos los enteros entre 2 y _LIMITE en busca de aquellos que sean perfectos. Esa búsqueda se realiza con el **for** más externo. Para cada valor de num la variable suma se inicializa a cero y se calcula, en el **for** anidado la suma de todos sus divisores.

Después del cálculo de cada suma, si su valor es el mismo que el entero inicial, entonces ese número será perfecto (esa es su definición) y así se mostrará por pantalla.

La búsqueda de divisores se hace desde el 1 (que siempre interviene) hasta la mitad de num: ninguno de los divisores de num puede ser mayor que su mitad. Desde luego, se podría haber inicializado la variable suma al valor 1, y comenzar a buscar los divisores a partir del 2, porque, efectivamente, el entero 1 es divisor de todos los enteros.

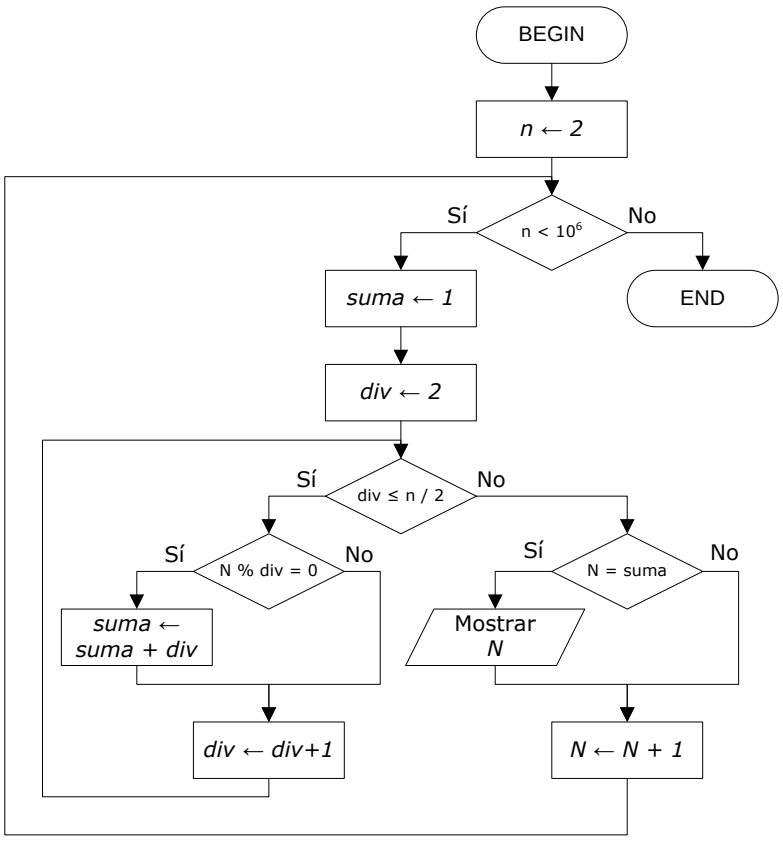


Figura 10.10.: Algoritmo correspondiente al enunciado 10.11.

10.11. ***Escriba un programa que muestre por pantalla un término cualquiera de la serie de Fibonacci.***

La serie de Fibonacci está definida de la siguiente manera: el primer y el segundo elementos son iguales a 1. A partir del tercero, cualquier otro elemento de la serie es igual a la suma de sus dos elementos anteriores.

Tiene el flujograma en la Figura 5.16. y su implementación en Código 5.7.

10.12. ***Escriba un programa que indique si un número introducido por teclado es cuadrado perfecto.***

Ya sabe que un cuadrado perfecto es un entero que se puede expresar como potencia de dos de otro entero; por ejemplo, 1, 4, 9, 16, 25, etc.

Código 10.27. Posible solución al problema 10.12.

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

int main(void)
{
      long a;

      printf("Valor de a ... ");    scanf(" %ld", &a);
      if((long)sqrt(a) * (long)sqrt(a) == a)
            printf("SÍ ES CUADRADO PERFECTO\n");
      else
            printf("NO ES CUADRADO PERFECTO\n");
      return 0;
}
```

Hay muchas formas de dar respuesta a esta pregunta. La construcción de un programa que resuelva esta cuestión es muy sencilla; todo el problema está en saber cómo averiguarlo, al margen de la programación.

Una posibilidad es, por ejemplo, verificar que al elevar al cuadrado su raíz cuadrada, expresada como entero, obtenemos el mismo número de inicio. Lo tenemos en Código 10.27.

Otra forma es verificar que la expresión entera de la raíz cuadrada es igual a la raíz cuadrada. Simplemente cambiaría la condición del **if**:

```
if((long)sqrt(a) == sqrt(a))
```

Otra tercera opción sería buscar, entre los enteros menores que el valor de entrada, uno que, al multiplicarlo por sí mismo, ofrece como resultado el valor de entrada. Lo tenemos en Código 10.28.

Código 10.28. Posible solución al problema 10.12.

```
#include <stdio.h>

int main(void)
{
      long a, b;

      printf("Valor de a ... ");    scanf(" %ld", &a);
      for(b = 1 ; b * b < a ; b++);
      printf("%s ES CUADRADO PERFECTO\n" ,
                  b * b == a ? "SI" : "NO");
      return 0;
}
```

10.13. ***Escriba un programa que calcule cuántos enteros primos hay entre los números introducidos por teclado (ambos inclusive).***

Ya hemos hecho algunos ejercicios trabajando con números primos. Ahora hemos de recorrer todos los enteros del intervalo y aumentar un contador cada vez que hallemos un nuevo entero primo. Una posible solución al problema planteado podría ser la propuesta en Código 10.29.

El bloque de sentencias condicionado con el primer ***if***, situado inmediatamente después de la entrada de datos, verifica que el intervalo está marcado por dos enteros donde el primero es menor o igual que el segundo: si no es así, intercambia esos valores que marcan los límites del intervalo.

Código 10.29. Posible solución al problema 10.13.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
      long a , b , N;
      short div , primos;

      printf("Intervalo. Primer valor.... ");
      scanf(" %ld", &a);
      printf("Intervalo. Segundo valor... ");
      scanf(" %ld", &b);

      if(a > b) { a ^= b ; b ^= a ; a ^= b; }

      for(N = a ; N <= b ; N++)
      {
            raiz = sqrt(N);
            for(div = 2 ; div <= raiz && N % div ; div++);
            if(div > sqrt(N)) primos++;
      }
      printf("%hd pr. entre %ld y %ld" , primos , a , b);

      return 0;
}
```

10.14. ***Escriba un programa que sume los números primos menores que mil.***

Una posible solución podría ser la sugerida en Código 10.30.

Código 10.30. Posible solución al problema 10.14.

```
#include <stdio.h>
#include <math.h>
#define LIMITE 1000
int main(void)
{
      long suma = 3;    // Suma de primos: 1 + 2 + ...
      long N;           // Número candidato a primo
      short d;          // sucesivos posibles divisores...
      short raiz;

      for(N = 3 ; N < LIMITE ; N += 2)
      {
            for(raiz = sqrt(N), d = 3 ;
                d <= raiz && N % d ;
                d += 2);

            if(d > raiz) suma += N;
      }

      printf("La suma es %ld", suma);
      return 0;
}
```

10.15. ***Escriba un programa que solicite al usuario valores enteros, uno tras otro, hasta que introduzca uno que sea Primo.***

Un posible código podría ser el propuesto en Código 10.31.

Código 10.31. Posible solución al problema 10.15.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
      long a;
      short d;

      do
      {
            printf("Valor de a ... ");    scanf(“ %ld", &a);
            for(d = 2 ; d <= sqrt(a) && a % d ; d++);
      }while(d <= sqrt(a) || !a);

      return 0;
}
```

10.16. ***Muestre por pantalla todos los enteros primos comprendidos entre dos enteros introducidos por teclado.***

Un posible código podría ser el propuesto en Código 10.32.

Código 10.32. Posible solución al problema 10.16.

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

int main(void)                        {
      long a, b, div, num;
      short primos;
```

Código 10.32. (Cont.)

```
      printf("Límite inferior ... ");    scanf(" %ld",&a);
      printf("Límite superior ... ");    scanf(" %ld",&b);
      printf("Los primos entre %ld y %ld son ...\n\t",
                        a, b);
      for(num = a, primos = 0 ; num <= b ; num++)
      {
            for(div = 2 ;
                div <= sqrt(num) && num % div ;
                div++);

            if(num % div)
            {
                  if(primos != 0 && primos % 10 == 0)
                        printf("\n\t");
                  primos++;
                  printf("%6ld,",num);
            }
      }
      return 0;
}
```

La salida por pantalla del programa podría ser la siguiente:

```
Límite inferior ... 123
Límite superior ... 264
Los primos entre 123 y 264 son ...

127,   131,   137,   139,   149,   151,   157,   163,   167,   173,
179,   181,   191,   193,   197,   199,   211,   223,   227,   229,
233,   239,   241,   251,   257,   263,
```

10.17. ***Escriba un programa que calcule el número π,sabiendo que este número verifica la siguiente relación:***

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2 \cdot k + 1}$$

En el capítulo 5 ha quedado ya recogido este enunciado. Código 10.33. propone una posible solución.

Código 10.33. Posible solución al problema 10.17.

```
#include <stdio.h>
#define LIMITE 100000

int main(void)
{
      double PI = 0;

      for(int i = 1 , e = 4 ; i < LIMITE ; i += 2 , e = -e)
            PI += e / (double)i;

      printf("El valor de PI es ... %lf.\n\n" , PI);
      return 0;
}
```

La variable PI se inicializa a cero. En cada iteración irá almacenando la suma de todos los valores calculados. En lugar de calcular $\pi/4$, calculamos directamente el valor de π: por eso el numerador (variable que hemos llamado e) no varía entre -1 y +1, sino entre -4 y +4. El valor de la variable `i` aumenta de dos en dos, y va tomando los diferentes valores del denominador 2·k + 1. Es necesario forzar el tipo de la variable `i` a **`double`**, para que el resultado de la operación cociente no sea un entero, que a partir de `i` igual a cero daría como resultado el valor cero.

10.18. ***Escriba un programa que calcule el número π, sabiendo que este número verifica la siguiente relación:***

$$\frac{\pi}{2} = \frac{2}{1}\times\frac{2}{3}\times\frac{4}{3}\times\frac{4}{5}\times\frac{6}{5}\times\frac{6}{7}\times\frac{8}{7}\times\frac{8}{9}\times\ldots$$

De nuevo el cálculo del valor del número pi. Estos ejercicios son muy sencillos de buscar (Internet está llena de definiciones de propiedades del número pi) y siempre es fácil comprobar si hemos realizado un buen código: basta ejecutarlo y comprobar si sale el famoso 3.14.

La implementación podría tomar la forma propuesta en código 10.34.

Código 10.34. Posible solución al problema 10.18.

```
#include <stdio.h>
#define LIMITE 100000

int main(void)
{
	double PI = 2;
	long num = 2, den = 1, i = 1;

	for( ; i < LIMITE ; i++ , i % 2 ? num+=2 : den+=2)
		PI *= num / (double)den;

	printf("El valor de PI es ... %lf.",PI);
	return 0;
}
```

10.19. ***Defina un algoritmo que recibe por teclado las calificaciones de cada uno de los alumnos de una clase. Las notas deber ser siempre comprendidas entre el cero y el diez. Cualquiera otra nota no la tomará en consideración. La introducción de datos terminará cuando se introduzca la nota menos 1 (-1).***

Al terminar la introducción de datos, el programa debe mostrar por pantalla la siguiente información:

- ***Número de aprobados (nota igual o mayor que cinco) y porcentaje del total de evaluados.***

- ***Número de suspensos y porcentaje del total de evaluados.***
- ***Nota media entre los aprobados.***
- ***Nota media entre los suspendidos.***

Un posible algoritmo de respuesta al problema podría ser el siguiente:

ENUNCIADO: Diversos cálculos con una colección de nota que el usuario va introduciendo por pantalla.

```
1.    Inicializar Variables:
      Nap ← 0.   Nsus ← 0.   Sap ← 0.    Ssus ← 0.
2.    DO
      2.1.  Leer nota.
      2.2.  IF 0 ≤ nota < 5                          Condición C1
            THEN
                  Nsus ← Nsus + 1
                  Ssus ← Ssus + nota
            ELSE
                  IF 5 ≤ nota ≤ 10                   Condición C2
                  THEN
                        Nap ← Nap + 1
                        Sap ← Sap + 1
                  END IF
            END IF
      WHILE nota ≠ -1                                Condición C3
      END DO
3.    Mostrar número de aprobados: Nap.
4.    Mostrar número de suspensos: Nsus
5.    IF Nap + Nsus  ≠ 0                             Condición C4
      THEN
            Mostrar % de aprobados:   Nap·100 / (Nap + Nsus)
            Mostrar % de suspensos:   Nsus·100 / (Nap + Nsus)
      END IF
6.    IF Nap ≠ 0                                     Condición C5
      THEN
            Mostrar media de los aprobados: Sap / Nap
      END IF
7.    IF Nsus ≠ 0                                    Condición C6
      THEN
            Mostrar media de los suspensos: Ssus / Nsus
      END IF
```

Donde hemos llamado: Nap la variable que cuenta el número de aprobados; Nsus a la que cuenta el número de suspensos; Sap a la variable que suma las notas de los aprobados; y Ssus a la que suma las notas de los suspensos.

El Flujograma de este algoritmo puede verse en la Figura 10.11. Queda pendiente que redacte el programa en lenguaje C.

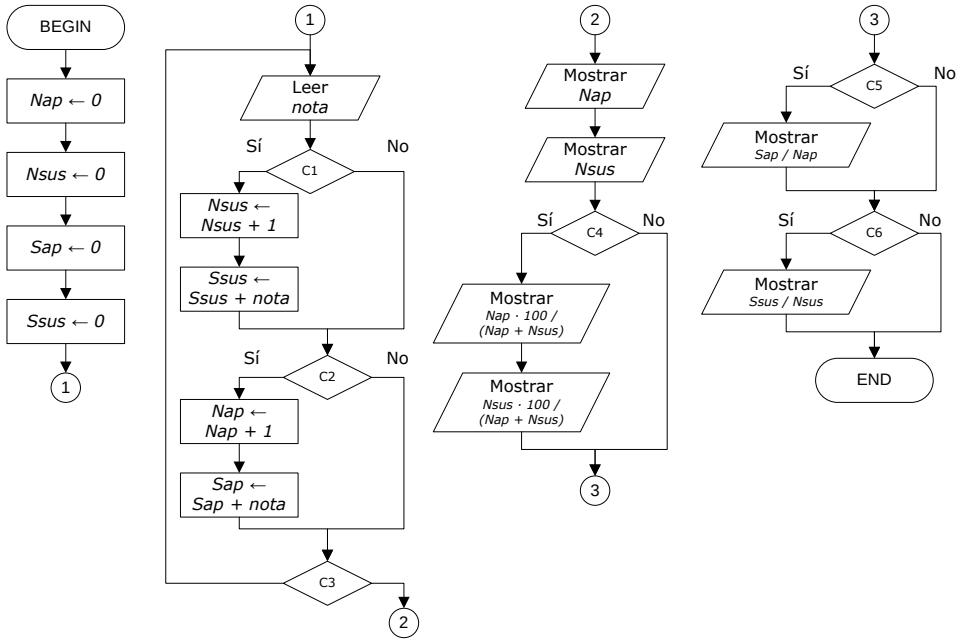


Figura 10.11: Algoritmo correspondiente al enunciado 10.19.

10.20. ***Calcule el valor del número e. sabiendo que verifica la siguiente relación:***

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Código 10.35. recoge una posible implementación.

Código 10.35. Posible solución al problema 10.20.

```
#include <stdio.h>
int main(void)                                  {
      double p = 1, e = 1;
      short n = 10000, i;

      for(i = 1 ; i < n ; i++)                  {
            p *= 1.0 / i;
            e += p;
      }

      printf("El numero e es ... %20.17lf.",e);
      return 0;
}
```

Debe tener la precaución de no pretender calcular los sucesivos valores del factorial para obtener, a continuación, si valor inverso: ya el factorial de 14 no cabe en una variable entera de 32 bits. En la implementación sugerida lo que se calcula es, directamente, el inverso del factorial. Tenga en cuenta que $1 / p! = 1 / p \times 1 / (p - 1)!$

10.21. ***El número áureo (Φ) verifica muchas curiosas propiedades. Por ejemplo:***

$\Phi^2 = \Phi + 1$ $\quad$ $\Phi - 1 = 1 / \Phi$ $\quad$ $\Phi^3 = (\Phi + 1) / (\Phi - 1)$

$\Phi = 1 + 1/\Phi$ $\quad$ $\Phi = \sqrt[2]{1 + \Phi}$ $\quad$ **y otras…**

La penúltima expresión presentada ($\Phi = 1 + 1 / \Phi$) muestra un camino curioso para el cálculo del número áureo:

$$\Phi = 1 + \frac{1}{\Phi} \Rightarrow \Phi = 1 + \frac{1}{1 + \frac{1}{\Phi}} \Rightarrow \Phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\Phi}}}} \ldots$$

Y, entonces un modo de calcular el número áureo es el siguiente:

$$\Phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots \cfrac{1}{1 + \frac{1}{1}}}}}}$$

Es decir, inicializando Φ al valor 1, se puede ir afinando en el cálculo del número áureo a base de repetir muchas veces `Φ = 1 + 1 / Φ`***.***

Presente el algoritmo para calcular el número áureo mediante este procedimiento haciendo, por ejemplo, la sustitución `Φ = 1 + 1 / Φ` ***mil veces. Mostrar luego el resultado por pantalla.***

El enunciado es algo aparatoso. Pero el algoritmo que lo resuelve es muy sencillo. Hay que inicializar la variable `Phi` a 1 y luego realiza 1000 veces la sustitución `Phi = 1 + 1 / Phi`. El algoritmo queda como sigue:

ENUNCIADO: Cálculo del número áureo.

1. Inicializar Variables:
   1.1. `Phi ← 1.`
2. **WHILE** `Phi < 1000`
   2.1. `Phi ← 1 + 1 / Phi`
   END WHILE
3. Resultado: `Phi`.

Siguiendo con el mismo enunciado, también podemos plantearnos calcular el número áureo a partir de la relación $\Phi = \sqrt[2]{1 + \Phi}$. De nuevo tenemos:

$$\Phi = \sqrt[2]{1 + \Phi} = \sqrt[2]{1 + \sqrt[2]{1 + \Phi}} = \sqrt[2]{1 + \sqrt[2]{1 + \sqrt[2]{1 + \Phi}}} = \ldots$$

$$\ldots = \sqrt[2]{1 + \sqrt[2]{1 + \sqrt[2]{1 + \sqrt[2]{1 + \sqrt[2]{1 + \ldots \sqrt[2]{1 + \Phi}}}}}}$$

eso nos brinda otro algoritmo, casi idéntico al anterior, donde ahora cambiamos la operación cociente por la operación raíz cuadrada.

ENUNCIADO: Cálculo del número áureo.

1. Inicializar Variables:
   1.1. `Phi` ← `1`.
2. **WHILE** `Phi < 1000`
   2.1. `Phi` ← $\sqrt[2]{1 + Phi}$.
   END WHILE
3. Resultado: `Phi`.

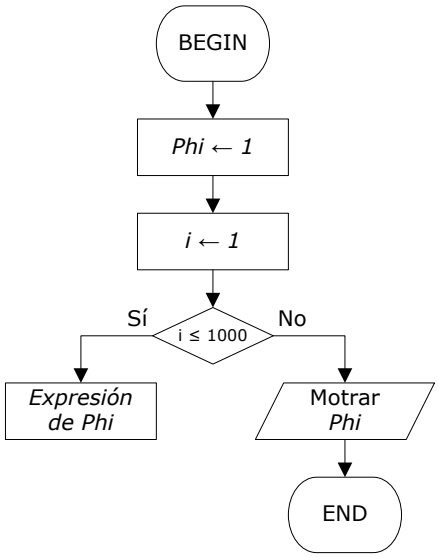


Figura 10.12.: Número Áureo

Ambos algoritmos quedan recogidos en la Figura 10.12. Si Expresión es $\Phi = 1 + 1 / \Phi$, entonces estamos en el primer algoritmo; si Expresión es $\Phi = \sqrt[2]{1 + \Phi}$, entonces estamos en el segundo algoritmo.

Código 10.36. Posible solución al problema 10.21.

```
#include <stdio.h>
#define LIMITE 1000

int main(void)
{
	double au = 1;
	int i;

	for(i = 0 ; i < LIMITE ; i++)	au = 1 + 1 / au;

	printf("El número áureo es ..... %lf.\n",au);
	printf("Áureo al cuadrado es ... %lf.\n",au * au);

	return 0;
}
```

En Código 10.36. se recoge una posible implementación.

No se ha hecho otra cosa que considerar lo que sugiere el enunciado: inicializar el número áureo a 1, y repetir mil veces la iteración $\Phi = 1 + 1/\Phi$. Al final mostramos también el cuadrado del número áureo: es un modo rápido de verificar que, efectivamente, el número hallado es el número buscado: el número áureo verifica que su cuadrado es igual al número incrementado en uno.

La salida que ofrece este código por pantalla es la siguiente:

```
El número áureo es ..... 1.618034.
Áureo al cuadrado es ... 2.618034.
```

Con la expresión de las raíces cuadradas sucesivas, la implementación queda como se propone en Código 10.37. Este programa ofrece una salida idéntica a la anterior. Y el código es casi igual que el anterior: únicamente cambia la definición de la iteración.

Código 10.37. Posible solución al problema 10.21.

```
#include <stdio.h>
#include <math.h>
#define LIMITE 100000

int main(void)
{
      double au = 1;
      int i;

      for(i = 0 ; i < LIMITE ; i++)      au = sqrt(1 + au);

      printf("El número áureo es ..... %lf.\n",au);
      printf("Áureo al cuadrado es ... %lf.\n",au * au);

      return 0;
}
```

10.22. ***La función sen x (x es una variable real que guarda el valor del ángulo en radianes) puede ser expresada, utilizando sus series de Taylor alrededor de cero así:***

$$\text{sen } x = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$$

Escriba un programa que solicite del usuario el valor de x ***y muestre por pantalla su seno, calculado de acuerdo con la expresión arriba indicada.***

Podemos usar la función pow, cuyo prototipo es

double pow(**double, double**);

que recibe la base como primer parámetro, y el exponente como segundo parámetro y que está declarada en el archivo de cabercera math.h. Código 10.38. recoge una posible solución.

Código 10.38. Posible solución al problema 10.22.

```
#include <stdio.h>
#include <math.h>
#define PI 3.141596

int main(void)
{
      double x;
      double seno = 0, invF = 1.0;
      long i, sw;

      printf("Valor de x ... ");    scanf(" %lf", &x);
      x *= PI / 180;          // Pasamos a radianes.

      for(i = 1 ,sw = 1 ; i < 10000 ; sw = -sw , i += 2)
      {
            seno += sw * pow(x, i) * invF;
            invF *= 1.0 /((i + 1) * (i + 2));
      }

      printf("El seno de %lf es %lf\n", x, seno);

      return 0;
}
```

10.23. ***La integral definida de la función f(x) = x, en el intervalo [0, 10] es igual a 50 ($[x^2 / 2]_0^{10} = 100 / 2 - 0$)***

Para realizar un cálculo aproximado de esta integral de forma analógica, bastaría realizar las sucesivas sumas de los rectángulos (tan estrechos como podamos) que van desde el eje de las abscisas hasta la gráfica f(x) = x..

Se podría dividir el intervalo de 0 a 10 en cientos o miles de segmentos; calcular de cada uno de ellos la altura de la imagen del punto medio del segmento. Realice un programa que calcule la integral de la función f(x) = x en el intervalo

comprendido entre 0 y 10. Calcule la superficie de los rectángulos en intervalos de milésimas: mil intervalos entre cada entero.

La dificultad, de nuevo, no está en la sintaxis de C, sino en saber con certeza qué es lo que vamos a querer expresar. Código 10.39. recoge una posible solución.

Código 10.39. Posible solución al problema 10.23.

```
#include <stdio.h>

#define TRAMOS 1000     // Intervalos por unidad
#define RANGO 10        // Límite superior de la integral

int main(void)
{
      double Sup = 0;
      long i;

      for(i = 1 ; i <= RANGO * TRAMOS ; i++)
            Sup += (double)i / (TRAMOS * TRAMOS);

      printf("La superficie es ... %lf", Sup);
      return 0;
}
```

10.24. ***Escriba un programa que muestre por pantalla todas las ternas de tres enteros menores que 100 que verifiquen la relación del teorema de Pitágoras. Por ejemplo: 3 - 4 - 5; 5 - 12 - 13; …***

Código 10.40. recoge una posible solución.

Código 10.40. Posible solución al problema 10.24.

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

      for(a = 1 ; a < 100 ; a++)
            for(b = a ; b < 100 ; b++)
                  for(c = b ; c < 100 ; c++)
                        if(c * c == a * a + b * b)
                              printf("%ld-%ld-%ld\n",a,b,c);

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

10.25. ***Decimos que dos números enteros positivos a y b son números amigos si se verifica que a es la suma de todos los divisores de b (excepto el mismo valor de b) y b es la suma de los divisores de a (excepto el mismo valor de a).***

Un ejemplo es el par (220, 284), ya que: los divisores propios de 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110, que suman 284; y los divisores propios de 284 son 1, 2, 4, 71 y 142, que suman 220.

Haga un programa que determine, cuáles de los enteros del intervalo [2,10000] tienen un entero amigo.

De nuevo trabajamos con un algoritmo que ya hemos implementado: la búsqueda de los divisores de un entero. Pero ahora esa búsqueda sólo es parte de un proceso más amplio. Debemos, para cada candidato del intervalo, buscar sus divisores y sumarlos; y con la suma obtenida hacer de nuevo el cálculo de la suma de sus divisores, para verificar si esa

suma es igual al entero candidato que teníamos inicialmente. Explicado de otra manera: Dentro del intervalo indicado $i \in [2,10000]$, se calcula la suma de los divisores propios de cada `i` (`s1`); para el valor de cada suma obtenida se calcula la suma de sus correspondientes divisores propios (`s2`); cuando el valor de `i` coincide con esta segunda suma (`i` es igual a `s2`) tenemos un par de enteros amigos.

Código 10.41. recoge una posible solución.

Código 10.41. Posible solución al problema 10.25.

```
#include <stdio.h>
int main(void)
{
	short i, d, s1, s2;

	for(i = 2 ; i < 10000 ; i++)
	{
		for(d = 1 , s1 = 0 ; d <= i / 2 ; d++)
			if(i % d == 0) s1 += d;

		for(d = 1 , s2 = 0 ; d <= s1 / 2 ; d++)
			if(s1 % d == 0) s2 += d;

		if(s2 == i && i != s1 && i < s1)
			printf("%5hd y %5hd.\n", i, s1);
	}

	return 0;
}
```

La condición final elimina, de entre los posibles pares amigos, aquellos que son números perfectos (la suma de sus divisores es igual a él mismo), y elimina las redundancias: 220 es amigo de 284 y 284 es amigo de 220.

10.26. ***Cinco marineros llegan, tras un naufragio, a una isla desierta con un gran número de cocoteros y un pequeño mono. Dedican el primer día a recolectar cocos, pero ejecutan con tanto afán este trabajo que acaban agotados, por lo que deciden repartirse los cocos al día siguiente.***

Durante la noche un marinero se despierta y, desconfiando de sus compañeros, decide tomar su parte. Para ello, divide el montón de cocos en cinco partes iguales, sobrándole un coco, que regala al mono. Una vez calculada su parte la esconde y se vuelve a acostar.

Un poco más tarde otro marinero también se despierta y vuelve a repetir la operación, sobrándole también un coco que regala al mono. En el resto de la noche sucede lo mismo con los otros tres marineros.

Al levantarse por la mañana procedieron a repartirse los cocos que quedaban entre ellos cinco, no sobrando ahora ninguno.

¿Cuántos cocos habían recogido inicialmente? Mostrar todas las soluciones posibles menores de 1 millón de cocos.

Vamos a presentar la solución más general: Supondremos que tenemos `N` marineros y que, cada vez que uno de ellos se levanta durante la noche para hacer `N` partes sobran `m` cocos que se le dan al mono. Así, el caso del enunciado no es más que un caso particular de la solución que aquí mostramos.

Este problema tiene poco de informática y bastante de pensar cómo dar con la solución. Su programación es sencilla… una vez se ha logrado saber cómo dar con la respuesta. Por eso, sería útil que quien quiera aprender a programar una aplicación que resuelva este ejercicio, antes lo intentara resolver en un papel.

El número de cocos (lo llamaremos C) debe ser tal que al llegar el primer marinero ocurra que haga N montones y sobren m cocos para el mono. Es decir, C es tal que $C \bmod N = m$.

Cuando el primer marinero se acuesta de nuevo, se ha llevado y escondido una porción de los cocos. Además ha entregado m al mono. Por lo tanto, los cocos que quedan son $C \leftarrow (N - 1) \cdot (C - m) / N$.

Ahora se levanta el segundo marinero. Y se repite la faena: Vuelve a hacer N montones, vuelven a sobrar m cocos, que se los da al mono, y se lleva su parte, dejando los restantes en el menguado montón inicial. Quedan, por tanto, $C \leftarrow (N - 1) \cdot (C - m) / N$ cocos, donde ahora C es el número de cocos que ha dejado el marinero primero, y no el número total de cocos recolectado.

Y le toca ahora el turno al tercer marinero. Y vuelve a hacer N montones, vuelven a sobrar m cocos, que se los da al mono, y se lleva su parte, dejando los restantes en el cada vez más menguado montón inicial. Quedan, por tanto, $C \leftarrow (N - 1) \cdot (C - m) / N$ cocos, donde ahora C es el número de cocos que ha dejado el marinero segundo, y no el número total de cocos recolectado.

Y así, van desfilando todos los marineros. Y todos realizan la misma operación. Y siempre ocurre que sobran, después de hacer N montones, los m cocos para el mono.

Al final, cuando ya han pasado los N marineros, se hace de día, y todos se levantan como si tal cosa, y reparten los cocos en N montones, y esta vez el mono se queda sin coco alguno. Es decir, al final, después de N veces en que se verifica que $C \bmod N = m$, y después de menguar el valor de C de acuerdo con la expresión $C \leftarrow (N - 1) \cdot (C - m) / N$, ahora se cumple que $C \bmod N = 0$.

Vamos a expresar esto con un algoritmo:

ENUNCIADO: Buscar soluciones al problema de los N marineros y el modo en la distribución de cocos. Como explica el enunciado, cada vez

que un marinero hace una distribución sobran m cocos que se le dan al mono.

Inicializar Variables:
    El usuario indica los valores de N y de m.
    cocos ← 1
WHILE cocos < 10^6 Condición C1
    C ← cocos
    i ← 1
    WHILE i ≤ N AND C mod N = m Condición C2
        C ← (N – 1) · (C – m) / N
        i ← i + 1
    END WHILE
    IF C mod N = 0 AND i > N Condición C3
    THEN
        Mostrar valor cocos.
    END IF
    cocos ← cocos + 1
END WHILE

Éste es el algoritmo queda representado en la Figura 10.13.

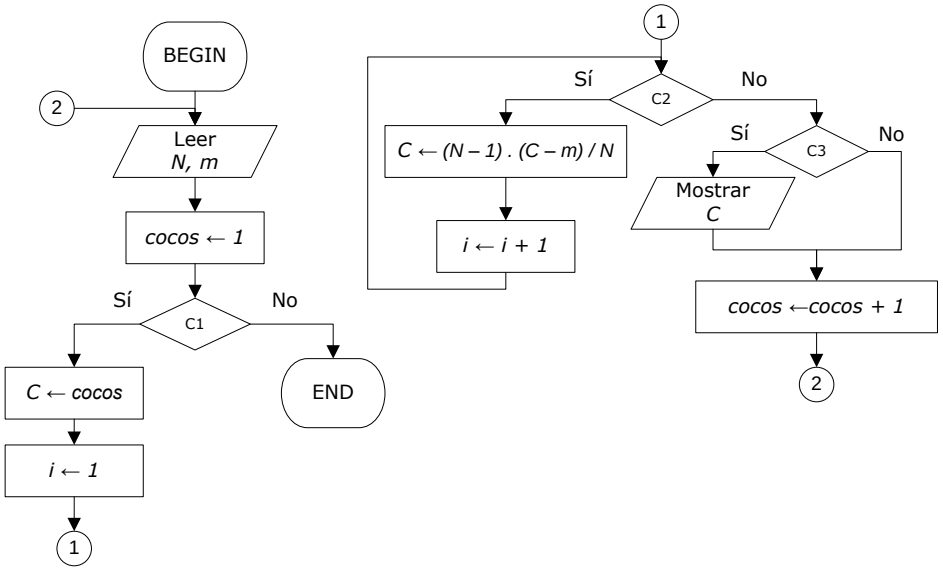


Figura 10.13.: Flujograma del algoritmo que resuelve el problema de los marineros, cocos y el mono.

10.27. ***Juego de las 15 cerillas: "Participan dos jugadores. Inicialmente se colocan 15 cerillas sobre una mesa y cada uno de los dos jugadores toma, alternativamente 1, 2 o 3 cerillas pierde el jugador que toma la última cerilla".***

Buscar el algoritmo ganador para este juego, generalizando: inicialmente hay *N* ***cerillas y cada vez se puede tomar hasta un máximo de*** *k* ***cerillas. Los valores*** *N* ***y*** *k* ***son introducidos por teclado y decididos por un jugador (que será el jugador perdedor), el otro jugador (que será el ordenador, y que siempre debe ganar) decide quien empieza el juego.***

El algoritmo que permite programar este juego es muy sencillo. Pero hay que saber llegar hasta él.

A lo largo de la explicación del juego llamaremos jugador G al ganador, es decir, al ordenador; y jugador P al perdedor, es decir, al usuario.

Supongamos que hemos tomado un valor k = 3: es decir, en cada jugada, el jugador puede retirar 1, ó 2, ó 3 cerillas. En la Tabla 10.2. queda recogido la cantidad de cerillas que se pueden llegar a retirar cada vez que juega un turno los jugadores G y P.

		Jugador P		
		1	**2**	**3**
Jugador G	**1**	2	3	4
	2	3	4	5
	3	4	5	6

Tabla 10.2. Cerillas que se retiran después de un turno de juego.

Está claro, y se ve reflejado en el cuadro, que sea cual sea el número de cerillas que retire el primer jugador (el jugador P), el segundo (el jugador G) siempre puede retirar un número tal que en todos los turnos de juego siempre se retire el mismo número de cerillas. Entre las cantidades de cerillas que se pueden retirar en un turno hay un valor que se repite en todas las filas y columnas: el 4. Ése es el valor al que debe jugar el que retira cerillas en segundo lugar (que será el jugador G). Si el primero (jugador P) ha retirado 1, entonces el ganador retira 3; si el primero retira 2 cerillas, entonces el ganador retira también 2. Si el primero retira 3 cerillas, entonces el ganador retira sólo una. Y, en general, si tenemos k cerillas, el número de cerillas que siempre se podrán retirar en un turno de jugada es k + 1. Si el jugador P retira c ≤ k cerillas, entonces el jugador G deberá retirar, si de verdad quiere ser el ganador, k + 1 - c cerillas.

Por lo tanto, la primera regla del jugador ganador es retirar las cerillas siempre después del jugador que va a perder, y retirar siempre tantas cerillas como sean necesarias para lograr que en cada turno se retiren k + 1 cerillas.

Si, por ejemplo, tenemos N = 15 y k = 3, entonces hemos quedado que en cada turno el ganador logrará que se retiren 4 cerillas. En la Figura 10.15. se ven las cerillas agrupadas en bloques de 4.

El objetivo del juego es lograr que después de la jugada del ganador quede sobre la mesa únicamente una cerilla. Si el jugador ganador va logrando que después de cada turno se retiren k + 1 cerillas (es decir, 4 cerillas), lo que importa es que al inicio del juego sobre la mesa haya una cantidad de cerillas múltiplo de k + 1, más una cerilla más. Pero en nuestro juego no ocurre así: de hecho hay una cantidad múltiplo de 4 más TRES cerillas más.

Esa es la segunda regla del algoritmo ganador: el jugador ganador (jugador G) debe decidir quién comienza a jugar. Y lo hace de la siguiente manera:

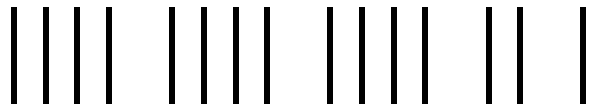


Figura 10.14. Cerillas. N = 15 y k = 3.

1. Calcula el resto de dividir (N + 1) por (k + 1): r ← (N – 1) mod (k + 1). Ya hemos dicho que el objetivo es que al principio el número de cerillas sea tal que (N + 1) sea múltiplo de (k + 1).
2. Si r = 0, entonces el jugador ganador cede gentilmente el turno al otro jugador, que será quien comenzará la partida. Si r ≠ 0 entonces será el jugador ganador quien comenzará a jugar, retirando r cerillas. A partir de ese momento tiene una cantidad de cerillas adecuada, y sin más que lograr retirar en cada turno k + 1 llevará inevitablemente a la derrota a su contrincante. En el ejemplo de la Figura 10.14. se obtiene el valor r = 2, y entonces el jugador G lo que debe hacer, si desea ganar, es empezar retirando esas dos cerillas. Y a partir de este momento, el jugador G juega siempre detrás del jugador P, y retira siempre k + 1 – c = 4 - c cerillas, donde c es el número de cerillas que ha retirado en la última jugada el jugador P.

Quien deduce estas reglas del juego, ya tiene resuelto el algoritmo, que toma la siguiente forma:

ENUNCIADO: Implementar el algoritmo ganador del juego de los N palillos que se retiran de k en k.

```
1.    Inicializar Variables: Entrada de los parámetros N y k.
2.    r ← (N + 1) mod (k + 1).
3.    IF r ≠ 0
      THEN
      3.1.  [Turno ordenador] rg ← r
            (rg: cerillas que retira el ganador)
```

```
        3.2.  N ← N – rg.
              (hay que descontar esas cerillas)
        3.3.  Mostrar rg y N.
        END IF
4.      WHILE N > 1
        4.1.  [Turno usuario]: u ← 1
        4.2.  [Lectura]: Leer rp (cerillas que retira el perdedor)
        4.3.  N ← N – rp (hay que descontar esas cerillas)
        4.4.  [Mostrar información]: Mostrar N
        4.5.  IF N > 1
              THEN
              4.5.1. [Turno ordenador]: u ← 2.
              4.5.2. rg ← (k + 1) – rp
              4.5.3. N ← N – rg (hay que descontar esas cerillas)
              4.5.4. [Mostrar información]: Mostrar rg y N.
              END IF
        END WHILE
5.      IF u = 1
        THEN
              [Mostrar ganador]: Mostrar "Ha ganado el usuario."
        ELSE
              [Mostrar ganador]: Mostrar "Ha ganado el PC."
        END IF
```

En el algoritmo hemos utilizado una variable (que hemos llamado u) que nos indica quién está jugando en cada momento. Si u = 1, juega el usuario, es decir, el jugador P; si u = 2, entonces juega el ordenador, es decir, el jugador G.

El Flujograma del algoritmo queda recogido en la Figura 10.15. Intente implementar el código, aunque quizá pueda ser más largo que los que hasta ahora ha visto.

El programa ha quedado un poco más largo que los anteriores. Se ha dedicado un poco de código a la presentación en el momento de la ejecución. Más adelante, cuando se haya visto el modo de definir funciones, el código quedará visiblemente reducido. Por ejemplo, en cuatro ocasiones hemos repetido el mismo código destinado a visualizar por pantalla las cerillas que quedan por retirar.

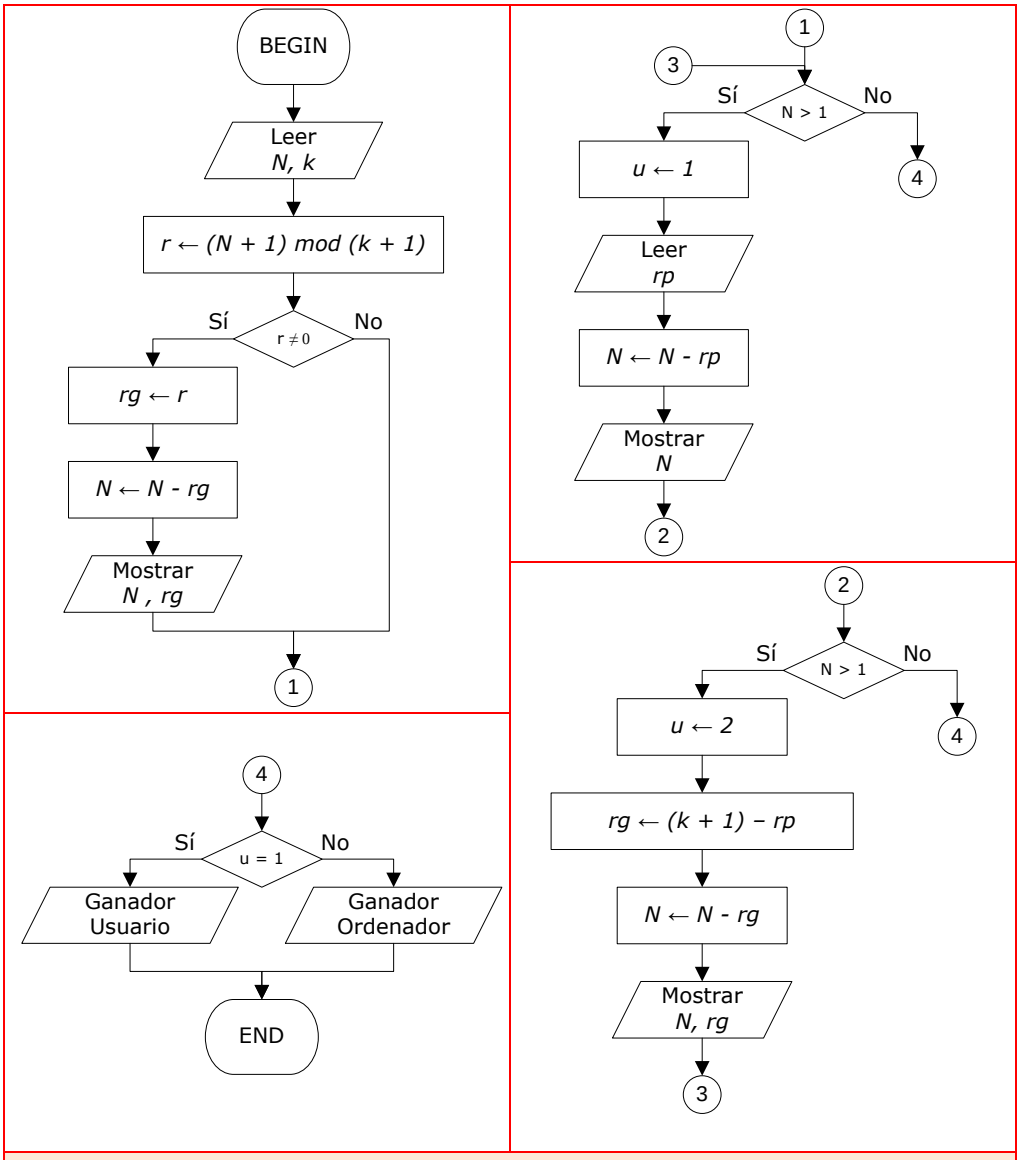


Figura 10.15.: Juego de las cerillas.

CAPÍTULO 11

ARRAYS NUMÉRICOS: VECTORES Y MATRICES.

Hasta el momento hemos trabajado con variables, declaradas una a una en la medida en que nos han sido necesarias. Pero pudiera ocurrir que necesitásemos un bloque de variables grande, por ejemplo para definir los valores de una matriz numérica, o para almacenar los distintos valores obtenidos en un proceso de cálculo del que se obtienen numerosos resultados del mismo tipo. Supongamos, por ejemplo, que deseamos hacer un programa que ordene mil valores enteros: habrá que declarar entonces mil variables, todas ellas del mismo tipo, y todas ellas con un nombre diferente.

Es posible hacer una declaración conjunta de un grupo de variables. Eso se realiza cuando todas esas variables son del mismo tipo. Esa es, intuitivamente, la noción del array. Y ese es el concepto que introducimos en el presente capítulo

Noción y *declaración* de array.

Un **array** (también llamado **vector**) es una colección de variables del mismo tipo todas ellas referenciadas con un nombre común.

La sintaxis para la declaración de un vector es la siguiente:

`tipo nombre_vector[dimensión];`

Donde `tipo` define el tipo de dato de todas las variables creadas, y `dimensión` es un literal que indica cuántas variables de ese `tipo` se deben crear. En ningún caso (estándares C89 y C90) está permitido introducir el valor de la dimensión mediante una variable. El compilador reserva el espacio necesario para almacenar, de forma contigua, tantas variables como indique el literal `dimensión`: reservará, pues, tantos bytes como requiera cada una de esas variables, multiplicado por el número de variables a crear.

Por ejemplo, la sentencia **`short int`** `mi_vector[1000];` reserva dos mil bytes de memoria consecutivos para mil variables de tipo **`short`**.

Cada una de las variables de un array tiene un comportamiento completamente independiente de las demás. Su única relación con todas las otras variables del array es que están situadas todas ellas de forma correlativa en la memoria. Cada variable tiene su propio modo de ser llamada: desde `nombre_vector[0]` hasta `nombre_vector[dimensión – 1]`. En el ejemplo anterior, tendremos 1000 variables que van desde `mi_vector[0]` hasta `mi_vector[999]`.

C no comprueba los límites del vector. Es responsabilidad del programador asegurar que no se accede a otras posiciones de memoria contiguas del vector. Por ejemplo, si hacemos referencia al elemento `mi_vector[1000]`, el compilador no dará como erróneo ese nombre, aunque de hecho no exista tal variable.

La variable `mi_vector[0]` está posicionada en una dirección de memoria determinada: no sabemos a priori cuál sea ésta. Pero a partir de ella,

todas los demás elementos del array tienen una ubicación conocida determinada. La variable `mi_vector[1]` está situada dos bytes más adelante, porque `mi_vector` es un array de tipo **`short`** y por tanto `mi_vector[0]` ocupa 2 bytes (como todos los demás elementos del vector), y porque `mi_vector[1]` es consecutiva en la memoria, a `mi_vector[0]`. Esa sucesión de ubicaciones sigue en adelante, y la variable `mi_vector[999]` estará 1998 bytes por encima de la posición de `mi_vector[0]`. Si hacemos referencia a la variable `mi_vector[1000]` entonces el compilador considera la posición de memoria situada 2000 bytes por encima de la posición de `mi_vector[0]`. Y de allí tomará valor o escribirá valor si así se lo indicamos. Pero realmente no está reservado el espacio para esta variable, y no sabemos qué estaremos realmente leyendo o modificando. Este tipo de errores son muy graves y a veces no se detectan hasta después de varias ejecuciones. Es lo que se denomina **violación de memoria**.

El recorrido del vector se puede hacer mediante índices. Por ejemplo:

```
short mi_vector[1000], i;
for(i = 0 ; i < 1000 ; i++) mi_vector[i] = 0;
```

Este código recorre el vector e inicializa a cero todas las variables.

Tenga cuidado, por ejemplo, con el recorrido del vector, que va desde el elemento 0 hasta el elemento 999. Un error habitual es escribir:

```
short mi_vector[1000], i;
for(i = 0 ; i <= 1000 ; i++) mi_vector[i] = 0; // ERROR !!!
```

Donde se hará referencia a la posición 1000 del vector, que no es válida.

Existe otro modo de inicializar los valores de un vector o array, sin necesidad de recorrerlo con un índice. Se puede emplear para ello el operador asignación, dando, entre llaves y separados por comas, tantos valores como dimensión tenga el vector. Por ejemplo;

```
short mi_vector[10] = {10,20,30,40,50,60,70,80,90,100};
```

que es equivalente al siguiente código:

```
short mi_vector[10], i;
for(i = 0 ; i < 10 ; i++) mi_vector[i] = 10 * (i + 1);
```

Cuando se inicializa un vector mediante el operador asignación en su declaración, se puede declarar ese vector sin especificar el número de variables que se deben crear. Por ejemplo:

```
short mi_vector[] = {10,20,30,40,50,60,70,80,90,100};
```

Si al declarar un array y asignarle valores se introducen menos valores que elementos tiene el array (por ejemplo, `short a[5] = {1, 2, 3};`), entonces los primeros elementos del array irán tomando los valores indicados; y cuando ya no queden valores, el resto del array queda inicializado con el valor 0. Una forma, por tanto, de declarar un array y asignar a todos sus elementos el valor cero es:

```
double array[100] = {0};
```

Donde al primer elemento tendré el valor cero, porque ese es el valor que expresamente se le asigna, y el resto de elementos valdrá cero, porque así lo hace la sintaxis.

Noción y declaración de array de dimensión múltiple, o matrices.

Es posible definir arrays de más de una dimensión. El comportamiento de esas variables vuelve a ser conceptualmente muy sencillo. La sintaxis de esa declaración es la siguiente:

```
tipo nombre_matriz[dim_1][dim_2]...[dim_N];
```

Donde los valores de las dimensiones son todos ellos literales.

Por ejemplo podemos crear una matriz tres por tres:

```
float matriz[3][3];
```

que reserva 9 bloques de cuatro bytes cada uno para poder almacenar valores tipo **float**. Esas variables se llaman también con índices, en

este caso dos índices (uno para cada dimensión) que van desde el 0 hasta el valor de cada dimensión menos uno.

Por ejemplo:

```
long matriz[5][2], i, j;
for(i = 0 ; i < 5 ; i++)
      for(j = 0 ; j < 2 ; j++)
            matriz[i][j] = 0;
```

donde tenemos una matriz de cinco filas y dos columnas, toda ella con los valores iniciales a cero. También se puede inicializar la matriz mediante el operador asignación y llaves. En este caso se haría lo siguiente:

```
long int matriz[5][2] = {{1,2},{3,4},{5,6},{7,8},{9,10}};
```

que es lo mismo que escribir

```
long matriz[5][2], i, j, k;
for(i = 0 , k = 1; i < 5 ; i++)
{
      for(j = 0 ; j < 2 ; j++)
      {
            matriz[i][j] = k;
            k++;
      }
}
```

Y de nuevo hay que estar muy vigilante para no sobrepasar, al utilizar los índices, la dimensión de la matriz. También, igual que antes con los arrays monodimensionales, si se asignan menos valores que elementos tiene el array, entonces el resto de elementos no explícitamente inicializados quedan asignados a cero.

Arrays en el estándar C99.

El estándar C99 ha introducido cambios sustanciales en la declaración de los arrays. Principalmente porque permite declarar el array de longitud variable: si C90 exigía que la dimensión del array estuviera definido

mediante un valor literal entero o una expresión constante que pueda ser calculada en tiempo de compilación, ahora C99 relaja esa exigencia o restricción y permite que el tamaño del array venga indicado mediante una expresión, no necesariamente constante, y no necesariamente evaluable en tiempo de compilación. Esto permite declarar un array de longitud variable, o un array cuyo tamaño queda determinado en tiempo de ejecución. Por ejemplo:

```
int main(void)
{
      short size;
      printf("Dimension del array ... ");
      scanf(" %hd", &size);
      double m[size];
      // . . .
      return 0;
}
```

Así, el array `m` tendrá, en el momento de su declaración, la dimensión que haya deseado el usuario al introducir un valor para la variable `size`. Es importante que ese valor no sea menor o igual que 0.

Cuando un array es declarado con un tamaño que no se conoce en tiempo de compilación, no se le pueden asignar valores en la declaración. El siguiente código dará un error porque no se puede inicializar, en tiempo de compilación, un array de tamaño variable.

```
short dimensión;
printf("dimensión ... ");    scanf(" %hd", &dimension);
double array[dimension] = {0};           // ERROR!!!
```

Ejercicios.

11.1. ***Escriba el código necesario para crear una matriz identidad (todos sus valores a cero, excepto la diagonal principal) de dimensión 3.***

En Código 11.1. se muestran dos formas de hacerlo.

Código 11.1. Sentencias que resuelven el Ejercicio 11.1.

```
short identidad[3][3] = {{1,0,0},{0,1,0},{0,0,1}};
```

```
short identidad[3][3], i, j;
for(i = 0 ; i < 3 ; i++)
      for(j = 0 ; j < 3 ; j++)
/*01*/      identidad[i][j] = i == j ? 1 : 0;
```

Como el lenguaje C devuelve el valor 1 cuando una expresión se evalúa como verdadera, hubiera bastando con que la línea marcada con `/*01*/` en Código 11.1. fuese: `identidad[i][j] = i == j;`

Para mostrar la matriz por pantalla el código es siempre más o menos el mismo (cfr. Código 11.2.)

Código 11.2. Para mostrar por pantalla los valores de una matriz.

```
for(i = 0 ; i < 3 ; i++)
{
      printf("\n\n");
      for(j = 0 ; j < 3 ; j++)
            printf("%5hd",identidad[i][j]);
}
```

11.2. ***Escriba un programa que solicite al usuario los valores de una matriz de tres por tres y muestre por pantalla la traspuesta de esa matriz introducida.***

Puede ver una posible solución en Código 11.3. Primero muestra la matriz introducida por el usuario, y más abajo muestra su traspuesta.

Código 11.3. Posible solución a Ejercicio 11.2.

```
#include <stdio.h>
int main(void)                {
      short matriz[3][3] , i , j;
      for(i = 0 ; i < 3 ; i++)
            for(j = 0 ; j < 3 ; j++)      {
                  printf("matriz[%hd][%hd] = ", i, j);
                  scanf(" %hd",&matriz[i][j]);
            }
      for(i = 0 ; i < 3 ; i++)      {
            printf("\n\n");
            for(j = 0 ; j < 3 ; j++)
                  printf("%5hd",matriz[i][j]);
      }

      printf("\n\n\n");
      for(i = 0 ; i < 3 ; i++)      {
            printf("\n\n");
            for(j = 0 ; j < 3 ; j++)
                  printf("%5hd",matriz[j][i]);
      }
      return 0;
}
```

11.3. *Escriba un programa que solicite al usuario los valores de dos matrices de tres por tres y muestre por pantalla cada una de ellas, y las matrices suma y producto.*

Este ejercicio no es sencillo para quien está empezando a programar y, ahora en concreto, para quien está comenzando con el manejo de vectores y de matrices. Pero en realidad no tiene ninguna dificultad

conceptual, y es un ejercicio muy clásico. Puede ser un buen ejercicio para aprender a manejar con soltura la operatoria de índices de matrices.

Se propone aquí las operaciones con matrices cuadradas: es el caso más sencillo. Más adelante, entre los ejercicios propuestos en este mismo capítulo, se le pedirá que resuelva el caso más general, de matrices no necesariamente cuadradas.

Supongamos que tenemos las matrices A y B, con sus valores $a_{i,j}$ y $b_{i,j}$. Cada coeficiente de la matriz suma S será $s_{i,j} = a_{i,j} + b_{i,j}$, para todos los valores $i, j \in \{1, 2, 3\}$.

Para los elementos de la Matriz producto, P, la expresión trae un poco más de matemática: el elemento $p_{i,j}$ es igual a la suma de los productos de los elementos de la fila i de la matriz A y de la columna j de la matriz B. Es decir

$$p_{i,j} = \sum_{k=1}^{3} a_{i,k} \times b_{k,j}$$

No es este manual el lugar para explicar esta expresión, en principio bien conocida de todos.

Con las dos operaciones claras, una posible implementación que resuelve nuestro problema podría ser la propuesta en Código 11.4.

Código 11.4. Posible solución a Ejercicio 11.3.

```
#define TAM 3
#include <stdio.h>

int main(void)
{
      short a[TAM][TAM], b[TAM][TAM];
      short s[TAM][TAM], p[TAM][TAM];
      short i, j, k;
```

Código 11.4. (Cont.)

```
// Entrada matriz a.
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
            {
                  printf("a[%hd][%hd] = ", i, j);
                  scanf(" %hd",&a[i][j]);
            }

// Entrada matriz b.
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
            {
                  printf("b[%hd][%hd] = ", i, j);
                  scanf(“ %hd",&b[i][j]);
            }

// Cálculo Suma.
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
                  s[i][j] = a[i][j] + b[i][j];

// Cálculo Producto.
// p[i][j]=a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j]
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
            {
                  p[i][j] = 0;
                  for(k = 0 ; k < TAM ; k++)
                        p[i][j] += a[i][k] * b[k][j];
            }

// Mostrar resultados.

// SUMA
      for(i = 0 ; i < TAM ; i++)
      {
            printf("\n\n");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",a[i][j]);
            printf("\t");
```

Código 11.4. (Cont.)

```
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",b[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",s[i][j]);
            printf("\t");
      }
// PRODUCTO
      printf("\n\n\n");
      for(i = 0 ; i < TAM ; i++)
      {
            printf("\n\n");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",a[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",b[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",p[i][j]);
            printf("\t");
      }
      return 0;
}
```

Como comprueba en los ejemplos de código que se van mostrando, en el manejo de matrices y vectores es frecuente utilizar siempre, como estructura de control de iteración, la opción **for**. Con ella, las mismas variables de control sirven para los índices que recorre la matriz.

11.4. ***Escriba un programa que solicite al usuario un conjunto de valores (tantos como quiera el usuario) y que al final, ordene esos valores de menor a mayor. El usuario termina su entrada de datos cuando introduzca el cero.***

Se propone en Código 11.5. una posible solución.

Introducción de datos: va solicitando uno a uno todos los datos, mediante una estructura de control **do-while**. La entrada de datos termina o cuando se introduce un 0, o cuando se han introducido tantos valores como enteros se han creado en el vector.

Código 11.5. Posible solución a Ejercicio 11.4.

```
#include <stdio.h>
int main(void)
{
      short datos[1000];
      short i, j, nn;
// Introducción de datos.
      i = 0;
      do
      {
            printf("Entada de nuevo dato ... ");
            scanf(“ %hi",&datos[i]);
            i++;
      }while(datos[i - 1] != 0 && i < 1000);
      nn = i - 1; // Total de datos válidos introducidos.
// Ordenar datos
      for(i = 0 ; i <= nn ; i++)
            for(j = i + 1 ; j <= nn ; j++)
                  if(datos[i] > datos[j])
                  {
                        datos[i] ^= datos[j];
                        datos[j] ^= datos[i];
                        datos[i] ^= datos[j];
                  }
// Mostrar datos ordenados por pantalla
      printf("\n\n");
      for(i = 0 ; i <= nn ; i++)
            printf("%li < ", datos[i]);
      printf("\b\b  ");
      return 0;
}
```

Se habrán introducido tantos datos como indique el valor de la variable `i`, donde hay que tener en cuenta que ha sufrido un incremento también cuando se ha introducido el cero, y ese último valor no nos interesa. Por eso ponemos la variable nn al valor `i – 1`.

Ordenar datos: Tiene una forma parecida a la que se presentó para la ordenación de cuatro enteros (en el tema de las estructuras de control condicionales: cfr. Ejercicios 9.2. y 9.3.), pero ahora para una cantidad desconocida para el programador (recogida en la variable nn). Por eso se deben recorrer todos los valores mediante una estructura de iteración, y no como en el ejemplo de los cuatro valores que además no estaban almacenados en vectores, y por lo tanto no se podía recorrer los distintos valores mediante índices. Los datos quedan almacenados en el propio vector, de menor a mayor.

Mostrar datos: Se va recorriendo el vector desde el principio hasta el valor nn: esos son los elementos del vector que almacenan datos introducidos por el usuario.

11.5. *Escriba un programa que solicite al usuario un entero positivo e indique si ese número introducido es primo o compuesto. Además, si el número es compuesto, deberá guardar todos sus divisores en un array y mostrarlos por pantalla.*

Se propone en Código 11.6. una posible solución.

Este programa es semejante a uno presentado en el capítulo 9 sobre las estructuras de control. Allí se comenta el diseño de este código. Ahora añadimos que, cada vez que se encuentra un divisor, se almacena en una posición del vector `D` y se incrementa el índice del vector (variable `i`).

Código 11.6. Posible solución a Ejercicio 11.5.

```
#include <stdio.h>
#define TAM 1000

int main(void)
{
	unsigned long int numero, mitad;
	unsigned long int i, div;
	unsigned long int D[TAM];

	for(i = 0 ; i < TAM ; i++) D[i] = 0;
	D[0] = 1;   // 1 es divisor de cualquier entero.

	printf("Numero que vamos a testear ... ");
	scanf(“ %lu", &numero);

	mitad = numero / 2;
	for(i = 1 , div = 2 ; div <= mitad ; div++)
	{
		if(numero % div == 0)
		{
			D[i] = div;
			i++;
			if(i == TAM)
			{
				printf("Vector mal dimensionado.");
				return -1;;
			}
		}
	}

	if(i < TAM) D[i] = numero;
	if(i == 1) printf("\n%lu es PRIMO.\n",numero);
	else
	{
		printf("\n%lu es COMPUESTO. ", numero);
		printf("Sus divisores son:\n\n");
		for(i = 0 ; i < TAM && D[i] != 0; i++)
			printf("\n%lu", D[i]);
	}
	return 0;
}
```

Se inicia el contador al valor 1 porque a la posición 0 del vector ya se le ha asignado el valor 1. La variable `i` hace de centinela y de chivato. Si después de buscar todos los divisores la variable `i` está al valor 1, entonces es señal de que no se ha encontrado ningún divisor distinto del 1 y del mismo número, y por tanto ese número es primo.

Para la dimensión del vector se utiliza una constante definida con la directiva de procesador **`define`**. Si se desea cambiar ese valor, no será necesario revisar todo el código en busca de las referencias a los límites de la matriz, sino que todo el código está ya escrito sobre ese valor prefijado. Basta cambiar el valor definido en la directiva para que se modifiquen todas las referencias al tamaño del vector.

11.6. Escriba un programa que defina un array de short de 32 elementos, y que almacene en cada uno de ellos los sucesivos dígitos binarios de un entero largo introducido por pantalla. Luego, una vez obtenidos todos los dígitos, el programa mostrará esos dígitos.

Ya se ha explicado anteriormente, en el Capítulo 7, cómo obtener el binario de un entero dado. Allí ha quedado resuelto un código en C que imprimía los dígitos binarios de un entero por pantalla. Ahora esos dígitos se han de guardar en un array de 32 elementos.

Se propone en Código 11.7. una posible solución.

Este código permite introducir tantos enteros como quiera el usuario. Cuando el usuario introduzca el valor cero entonces se termina la ejecución del programa. Ya quedó explicado el funcionamiento de este algoritmo en un tema anterior. Ahora simplemente hemos introducido la posibilidad de que se almacenen los dígitos binarios en un array.

Una posible salida por pantalla de este programa sería la siguiente:

```
Valor N ... 12
Codificación binaria... 00000000000000000000000000001100
Valor N ... -12
Codificación binaria... 11111111111111111111111111110100
Valor N ... 0
```

Código 11.7. Posible solución a Ejercicio 11.6.

```
#include <stdio.h>

int main(void)
{
      signed long N;
      unsigned short bits[32], i;
      unsigned long Test;

      do
      {
            printf("\nValor N ... ");
            scanf(" %li", &N);

            if(N == 0) break;
            for(i = 0 , Test = 0x80000000 ;
                Test ;
                Test >>= 1 , i++)
            {
                  bits[i] = Test & N ? 1 : 0;
            }

            printf("\nCodificación binaria ... ");
            for(i = 0 ; i < sizeof(long) * 8 ; i++)
            {
                  printf("%hu", bits[i]);
            }
      }while(1);

      return 0;
}
```

11.7. ***En la serie triangular cada elemento es igual a la suma de todos los enteros menores o iguales que su cardinal. V. gr., su quinto elemento es igual a 5 + 4 + 3 + 2 + 1 = 15.***

Escriba un programa que guarde en un array los 10 primeros elementos de la serie y los muestre luego por pantalla.

El proceso de cálculo de cada nuevo elemento, a partir del elemento anterior calculado, es sencillo: el elemento en la posición `i` de la serie triangular es igual al elemento en la posición `(i – 1)` más el valor `i`.

Se propone en Código 11.8. una posible solución.

Código 11.8. Posible solución a Ejercicio 11.7.

```
#include <stdio.h>
int main(void)
{
	long tr[10], i;

	for(i = 1 , tr[0] = 1 ; i < 10 ; i++)
		tr[i] = tr[i - 1] + i + 1;

	for(i = 0 ; i < 10 ; i++)
		printf("%ld\n", tr[i]);

	return 0;
}
```

11.8. ***Escriba un programa que cree una matriz triangular superior de dimensión 4, con todos los elementos distintos de cero iguales a 1.***

La matriz se dice triangular si es cuadrada y todos los valores por debajo de su diagonal principal son iguales a cero. Aquí hay que asignar a los restantes elementos de la matriz (diagonal principal y valores por encima de ella) el valor 1. Se propone en Código 11.9. una posible solución. Asigna a cada elemento de la matriz el valor 1 ó el valor 0 según que el índice `i` sea menor o igual que `j`, o sea mayor. También puede hacerse por asignación en la declaración de la matriz.

Código 11.9. Posible solución a Ejercicio 11.8.

```
long m[4][4] = {{1, 1, 1, 1},{0, 1, 1, 1},
                {0, 0, 1, 1},{0, 0, 0, 0, 1}};
```

```
#include <stdio.h>
int main(void)
{
      long m[4][4], i, j;

      for(i = 0 ; i < 4 ; i++)
            for(j = 0 ; j < 4 ; j++)
                  m[i][j] = i <= j ? 1 : 0;
      return 0;
}
```

11.9. *Escriba un programa que determine si en un array de 10 enteros los valores están bien ordenados: el menor en la posición cero y el mayor en la posición 9.*

Código 11.10 ofrece una posible solución. La condición de permanencia en la estructura **for** es que no se haya terminado de recorrer el array, y también que no se haya encontrado un valor fuera de sitio: cada valor es menor o igual que el siguiente.

Código 11.10. Posible solución a Ejercicio 11.9.

```
#include <stdio.h>
int main(void)
{
      long a[10];
      short i;
// Aquí va la parte de introducción de los valores.
// Evidentemente habría que incluir esa parte del código.
// Determinar si los valores están ordenados...
      for(i = 0 ; i < 9 && a[i] <= a[i + 1] ; i++);
      printf("%s ORDENADOS." , i == 9 ? "SI" : "NO");
      return 0;
}
```

11.10. ***Escriba un programa que sume todos los elementos de una matriz cuadrada y muestre ese valor por pantalla.***

Código 11.11. Posible solución a Ejercicio 11.10.

```
#include <stdio.h>
#define TAM 5
int main(void)
{
      long matriz[TAM][TAM] , suma;
      short i, j;
// Aquí va la parte de introducción de los valores.
// Evidentemente debe incluir esa parte del código.
// Calculo de la suma...
      for(suma = 0, i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
                  suma += matriz[i][j];
      printf("La suma es ... %ld", suma);
      return 0;
}
```

11.11. ***Escriba un programa que calcule la suma de los valores situados en el perímetro de una matriz de dimensión DIM.***

Código 11.12 ofrece una posible solución. Simplemente hemos de sumar la primera (`j = 0`) y la última (`j = DIM – 1`) columna. Y luego sumar la primera (`i = 0`) y la última (`j = DIM – 1`) fila. Y hay que tener la precaución de no sumar dos veces los valores de "las cuatro esquinas" de la matriz.

Código 11.12. Posible solución a Ejercicio 11.11.

```
#include <stdio.h>
#define DIM 25 // Este valor puede cambiar.
int main(void)
{
      long m[DIM][DIM] , Suma;
      short i, j;
// Aquí va la parte de introducción de los valores.
// Evidentemente debe incluir esa parte del código.
      for(Suma = 0 , i = 0 ; i < DIM ; i++)
            Suma += m[i][0] + m[i][DIM - 1];
      for(i = 1 ; i < DIM - 1 ; i++)
            Suma += m[0][i] + m[DIM - 1][i];
      printf("\n\nEl valor de la suma es ... %ld", Suma);
      return 0;
}
```

11.12. ***Podemos definir un polinomio mediante un array tipo*** `double` ***de tamaño igual a la dimensión del polinomio más 1, donde el término independiente es el elemento de índice 0; el término de la x es el elemento de índice 1; y, en general, el término de potencia j es el elemento de índice j.***

Diseñe un programa que solicite del usuario un valor de x y ofrezca como salida por pantalla el valor del polinomio para esa entrada.

Desde luego, vale la pena utilizar la función de cálculo de potencias, definida en `math.h`. En Código 11.13. se recoge una posible solución.

Código 11.13. Posible solución a Ejercicio 11.12.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
	double x, pol;
	double a[5] = {3, -1, 2, +5, -1};
	// Polinomio p(x) = 3 – x + 2*x^2 + 5*x^3 – x^4.
	short i;

	printf("Valor de  x ... ");
	scanf(" %lf", &x);

	for(pol = 0 , i = 0 ; i < 5 ; i++)
		pol += pow(x, i) * a[i];

	printf("El valor del poliomio es %lf", pol);

	return 0;
}
```

11.13. ***Escriba un programa que guarde en un array de 10 elementos los factoriales de los enteros comprendidos entre 0 y 9.***

Muy parecido al programa de crear un array que almacene en sus posiciones los valores de la serie triangular. Ahora en lugar de realizar la

suma, lo que debemos realizar es el producto. Todo lo demás es igual. Es muy sencillo calcular el factorial de un entero `i` si se conoce el factorial del entero `i – 1`. El primer valor, el del factorial de cero, se asigna por definición a 1. En Código 11.14. se propone una solución.

Código 11.14. Posible solución a Ejercicio 11.13.

```
#include <stdio.h>
int main(void)
{
      long fac[10];
      short i;

      for(fac[0] = 1 , i = 1 ; i < 10 ; i++)
            fac[i] = i * fac[i - 1];

      for(i = 0 ; i < 10 ; i++)
            printf("Factorial de %hd: %10ld\n", i, fac[i]);

      return 0;
}
```

11.14. ***Escriba un programa que defina una matriz tipo short de dimensión 8 X 8, y asigne como valores iniciales unos y ceros de tal manera que no haya ningún dígito igual a los adyacentes en su fila y en su columna:***

1	**0**	**1**	**0**	**1**	**0**	**1**	**0**
0	**1**	**0**	**1**	**0**	**1**	**0**	**1**
1	**0**	**1**	**0**	**1**	**0**	**1**	**0**

Y muestre, después, por pantalla, la matriz definida.

Hay muchas formas de resolver este sencillo problema. En Código 11.15. proponemos dos. Una es jugar con el valor de los índices para

decidir si el valor deberá ser 0 ó 1: cuando la suma (i + j) es par, entonces el valor de esta posición es 1; cuando la suma es impar, entonces el valor de esa posición es 0.

Código 11.15. Posible solución a Ejercicio 11.14.

```
short a[8][8] = {{1, 0, 1, 0, 1, 0, 1, 0},
                 {0, 1, 0, 1, 0, 1, 0, 1},
                 {1, 0, 1, 0, 1, 0, 1, 0},
                 {0, 1, 0, 1, 0, 1, 0, 1},
                 {1, 0, 1, 0, 1, 0, 1, 0},
                 {0, 1, 0, 1, 0, 1, 0, 1},
                 {1, 0, 1, 0, 1, 0, 1, 0},
                 {0, 1, 0, 1, 0, 1, 0, 1}};
```

```
#include <stdio.h>
int main(void)
{
      short a[8][8], i, j;

      for(i = 0 ; i < 8 ; i++)
            for(j = 0 ; j < 8 ; j++)
                  a[i][j] = (i + j) % 2 ? 0 : 1;
// Impresión por pantalla de la matriz creada...

      for(i = 0 ; i < 8 ; i++)
      {
            printf("\n");
            for(j = 0 ; j < 8 ; j++)
                  printf("%3hd", a[i][j]);
      }
      return 0;
}
```

11.15. ***Escriba un programa que declare un array de 1000 elementos y guarde en él los mil primeros enteros primos.***

Tenemos ahora en el array los primos encontrados hasta el momento. Así, ahora simplemente probamos a dividir con los primos menores o iguales que la raíz cuadrada. Puede ver una posible solución en Código 11.16.

Código 11.16. Posible solución a Ejercicio 11.15.

```
#include <stdio.h>
#include <math.h>
#define RANGO 1000

int main(void)
{
      long pr[RANGO], p;
      short i, j;

      for(pr[0] = 1 , pr[1] = 2 ,
          i = 2, p = 3 ; i < RANGO ;
          p += 2)
      {
            for(j = 1 ;
               (j < i) && (pr[j] <= sqrt(p)) && (p % pr[j]);
               j++);
            if(j == i || pr[j] > sqrt(p))
                  pr[i++] = p;
      }
      return 0;
}
```

11.16. ***Escriba un programa que declare un array de tipo*** `double` ***de 25 elementos. Asigne, a cada una de esas veinticinco posiciones del array, el valor del inverso del factorial del índice de la posición: al primer elemento el valor del inverso del factorial de 0; al segundo, el valor del inverso del factorial de 1; al tercero, el valor del inverso del factorial de 2; etc.***

Muy parecido a algunos anteriores. No requiere especial explicación. Tenga en cuenta que para calcular el factorial de un entero mayor que 13 hay que utilizar ya variables de tipo coma flotante, porque los valores de esos factoriales salen fuera del rango de valores de una variable **long**.

Para calcular el inverso del factorial usamos la definición:

$$\frac{1}{n!} = \frac{1}{(n-1)!} \times \frac{1}{n}$$

El código (muy sencillo) podrá tener una apariencia como la sugerida en Código 11.17.

Código 11.17. Posible solución a Ejercicio 11.16.

```
#include <stdio.h>
int main(void)
{
      double f[25];
      short i;

      for(i = 1, f[0] = 1 ; i < 25 ; i++)
            f[i] = f[i - 1] * 1.0 / i;

      return 0;
}
```

11.17. ***Declarar cuatro arrays de la misma dimensión. Ecriba un programa que asigne al primero los valores por entrada de teclado. Que el segundo sea copia del primero. Que el tercero sea copia en orden inverso del primero. Que el cuarto sea la suma del segundo y el tercero. Y que muestre los 4 arrays en 4 columnas.***

Código 11.18. ofrece una posible solución al ejercicio planteado.

Código 11.18. Posible solución a Ejercicio 11.17.

```
#include <stdio.h>
#define TAM 10

int main(void)
{
      short a[TAM], b[TAM], c[TAM], d[TAM], i, j;

// Introducción de valores en la matriz a...
      for(i = 0 ; i < TAM ; i++)
      {
            printf("a[%2hd] --> ", i);
            scanf(“ %hd",&a[i]);
      }
// La matriz b es copia de la matriz a...
      for(i = 0 ; i < TAM ; i++)   b[i] = a[i];

// La matriz c es copia (en orden inverso) de la matriz a...
      for(i = 0, j = TAM – 1; i < TAM; ) c[i++] = a[j--];

// La matriz s es la suma de las matrices b y c...
      for(i = 0 ; i < TAM ; i++)   d[i] = b[i] + c[i];

// Muestra de las cuatro matrices...
      printf("      a[i]      b[i]");
      printf("      c[i]      d[i]\n");
      for(i = 0 ; i < 4 ; i++)
            printf(" _________");
      printf("\n");
      for(i = 0 ; i < TAM ; i++)
      {
            printf("%10hd %10hd "  , a[i], b[i]);
            printf("%10hd %10hd\n" , c[i], d[i]);
      }
      return 0;
}
```

11.18. ***Cree un programa que asigne a un array los 30 primeros valores de la serie de fibonacci.***

Código 11.19. Posible solución a Ejercicio 11.18.

```
#include <stdio.h>
int main(void)
{
      long fibo[30];
      short i;

      for(i = 2 , fibo[0] = fibo[1] = 1 ; i < 30 ; i++)
            fibo[i] = fibo[i - 1] + fibo[i - 2];
      return 0;
}
```

11.19. ***Cree dos matrices. Asigne como quiera valores a la primera de ellas (o por entrada de teclado o por programa) Luego asigne a la segunda matriz los valores que la definan como matriz traspuesta de la primera.***

Deberá declarar ambas matrices con las dimensiones correctas para que, efectivamente, la segunda pueda ser la traspuesta de la primera.

Código 11.20. recoge un posible resultado.

Para dejar bien definidas las matrices creamos varias directrices **define**; y hacemos que el número de columnas de B sean igual al de las filas de A, y que el número de filas de B sean igual al de las columnas de A.

Código 11.20. Posible solución a Ejercicio 11.19.

```
#include <stdio.h>

#define _FA 8
#define _CA 5
#define _FB _CA
#define _CB _FA

int main(void)
{
	long a[_FA][_CA];
	long b[_FB][_CB];
	short i, j, k;

	for(i = 0 , k = 1 ; i < _FA ; i++)
		for(j = 0 ; j < _CA ; j++ , k++)
			a[i][j] = k;

	for(i = 0 ; i < _FA ; i++)
		for(j = 0 ; j < _CA ; j++)
			b[j][i] = a[i][j];

	return 0;
}
```

11.20. ***Declare 3 matrices. Asigne valores como quiera a las dos primeras. Y realice el producto de matrices que deberá quedar registrado en la tercera matriz.***

Se ha resuelto antes un ejercicio parecido: multiplicar matrices cuadradas 3 X 3. Ahora es más complicado.

Para lograr que las dimensiones de las matrices sean las correctas, usamos directivas define, de la misma forma que hemos hecho en el ejercicio anterior. Se muestra, en Código 11.21., un posible resultado.

Código 11.21. Posible solución a Ejercicio 11.20.

```
#include <stdio.h>

#define _FA 7
#define _CA 4
#define _FB _CA
#define _CB 9
#define _FC _FA
#define _CC _CB

int main(void)
{
      long a[_FA][_CA];
      long b[_FB][_CB];
      long c[_FC][_CC];
      short i, j, k;

// Asignacion de valores a las matrices A y B...
// Pendiente de implementar como usted quiera.

// P R O D U C T O ...
      for(i = 0 ; i < _FC ; i++)
            for(j = 0 ; j < _CC ; j++)
                  for(c[i][j] = k = 0 ; k < _CA ; k++)
                        c[i][j] += a[i][k] * b[k][j];
      return 0;
}
```

11.21. ***Podemos definir un polinomio mediante un array (cfr. enunciado del Ejercicio 11.12.). Declare un array de varios elementos (los que quiera: 10 por ejemplo) y otro de uno más que el anterior (por ejemplo 11). Asigne por teclado valores a cada uno de los coeficientes del polinomio representado por el primer array. Escriba un programa que realice los cálculos necesarios para que un segundo polinomio sea el integrado del primero, y un tercero sea el derivado del primero.***

En Código 11.22. se propone una posible implementación. El tamaño del polinomio integrado deberá ser de uno más que el polinomio de entrada. Y el valor de su primer elemento será cualquiera: le asignamos, por ejemplo, el valor cero. El tamaño del polinomio derivado será de un elemento menos que el inicial de entrada: en nuestra implementación lo que hacemos es asignar el valor cero al elemento de mayor índice.

Se puede plantear otros ejercicios: el valor de la superficie acotada por una función de polinomio y el eje de las x y entre las dos rectas verticales x = a y x = b. Cálculo de máximos, mínimos, inflexiones, etc.

Código 11.22. Posible solución a Ejercicio 11.21.

```
#include <stdio.h>
#include <math.h>
#define GRADO 10

int main(void)                                    {
      double pol[GRADO], ipol[GRADO + 1], dpol[GRADO];
      short i;

      for(i = 0 ; i < GRADO ; i++)                {
            printf("Coeficiente %hd ... ", i);
            scanf(“ %lf", pol + i);
      }

// INTEGRAL:
      for( ipol[0] = 0 , i = 0 ; i < GRADO ; i++)
            ipol[i + 1] = pol[i] / (i + 1);

// DERIVADA:
      for(i = 0 ; i < GRADO - 1; i++)
            dpol[i] = pol[i + 1] * (i + 1);
      dpol[GRADO - 1] = 0;

      return 0;
}
```

CAPÍTULO 12

CARACTERES Y CADENAS DE CARACTERES.

Ya hemos visto que un carácter se codifica en C mediante el tipo de dato **char**. Es una posición de memoria (la más pequeña que se puede reservar en C: un byte) que codifica cada carácter según un código arbitrario. Uno muy extendido en el mundo del PC y en programación con lenguaje C es el código ASCII.

Hasta el momento, cuando hemos hablado de los operadores de una variable **char**, hemos hecho referencia al uso de esa variable (que no se ha recomendado) como entero de pequeño rango o dominio. Pero donde realmente tiene uso este tipo de dato es en el manejo de caracteres y en el de vectores o arrays declarados de este tipo.

A un array de tipo **char** se le suele llamar **cadena de caracteres**.

En este capítulo vamos a ver las operaciones básicas que se pueden realizar con caracteres y con cadenas. No hablamos de operadores,

porque estos ya se han visto antes, en un capítulo previo, y se reducen a los aritméticos, lógicos, relacionales y a nivel de bit. Hablamos de operaciones habituales cuando se manejan caracteres y sus cadenas: operaciones que están definidas como funciones en algunas bibliotecas del ANSI C y que presentaremos en este capítulo.

Operaciones con caracteres.

La biblioteca `ctype.h` contiene abundantes funciones para la manipulación de caracteres. La biblioteca `ctype.h` define funciones de manejo de caracteres de forma individual, no concatenados formando cadenas.

Lo más indicado será ir viendo cada una de esas funciones y explicar qué operación realiza sobre el carácter y qué valores devuelve.

- `int isalnum(int c);`

  Recibe el código ASCII de una carácter y devuelve el valor 1 si el carácter que corresponde a ese código ASCII es una letra o un dígito numérico; en caso contrario devuelve un 0.

  Ejemplo de uso:

```
if(isalnum('@')) printf("Alfanumérico"):
else printf("No alfanumérico");
```

  Así podemos saber si el carácter '@' es considerado alfabético o numérico. La respuesta será que no lo es. Evidentemente, para hacer uso de esta función y de todas las que se van a ver en este epígrafe, hay que indicar al compilador el archivo donde se encuentran declaradas estas funciones: `#include <ctype.h>`.

- `int isalpha(int c);`

  Recibe el código ASCII de una carácter y devuelve el valor 1 si el carácter que corresponde a ese código ASCII es una letra; en caso contrario devuelve un 0.

Ejemplo de uso:

```
if(isalnum('2')) printf("Alfabético"):
else printf("No alfabético");
```

La respuesta será que '2' no es alfabético.

- **int** iscntrl(**int** c);

  Recibe el código ASCII de una carácter y devuelve el valor 1 si el carácter que corresponde a ese código ASCII es el carácter borrado o un carácter de control (ASCII entre 0 y 1F, y el 7F, en hexadecimal); en caso contrario devuelve un 0.

  Ejemplo de uso:

  ```
  if(iscntrl('\n')) printf("Carácter de control"):
  else printf("No carácter de control");
  ```

  La respuesta será que el salto de línea sí es carácter de control.

  Resumidamente ya, presentamos el resto de funciones de esta biblioteca. Todas ellas reciben como parámetro el ASCII de un carácter.

- **int** isdigit(**int** c); Devuelve el valor 1 si el carácter es un dígito; en caso contrario devuelve un 0.
- **int** isgraph(**int** c); Devuelve el valor 1 si el carácter es un carácter imprimible; en caso contrario devuelve un 0.
- **int** isascii(**int** c); Devuelve el valor 1 si el código ASCII del carácter es menor de 128; en caso contrario devuelve un 0.
- **int** islower(**int** c); Devuelve el valor 1 si el carácter es una letra minúscula; en caso contrario devuelve un 0.
- **int** ispunct(**int** c); Devuelve el valor 1 si el carácter es signo de puntuación; en caso contrario devuelve un 0.
- **int** isspace(**int** c); Devuelve el valor 1 si el carácter es el espacio en blanco, tabulador vertical u horizontal, retorno de carro, salto de línea, u otro carácter de significado espacial en un texto; en caso contrario devuelve un 0.

Código 12.1.

```
#include <stdio.h>
#include <ctype.h>
int main(void)                                  {
      unsigned short int a = 0, e = 0, i = 0, o = 0, u = 0;
      char ch;
      do                                        {
            ch = getchar();
            if(isalpha(ch))                     {
                  if(ch == 'a') a++;
                  else if(ch == 'e') e++;
                  else if(ch == 'i') i++;
                  else if(ch == 'o') o++;
                  else if(ch == 'u') u++;
            }
            else
                  printf("\b ");
      }while(ch != 10);

      printf("\nVocal a ... %hu" , a);
      printf("\nVocal e ... %hu" , e);
      printf("\nVocal i ... %hu" , i);
      printf("\nVocal o ... %hu" , o);
      printf("\nVocal u ... %hu" , u);

      return 0;
}
```

- **int** isupper(**int** c); Devuelve el valor 1 si el carácter es una letra mayúscula; en caso contrario devuelve un 0.
- **int** isxdigit(**int** c); Devuelve el valor 1 si el carácter es un dígito hexadecimal (del '0' al '9', de la 'a' a la 'f' ó de la 'A' a la 'F'); en caso contrario devuelve un 0.
- **int** tolower(**int** ch); Si el carácter recibido es una letra mayúscula, devuelve el ASCII de su correspondiente minúscula; en caso contrario devuelve el mismo código ASCII que recibió como entrada.

- **int** toupper(**int** ch); Si el carácter recibido es una letra minúscula, devuelve el ASCII de su correspondiente mayúscula; en caso contrario devuelve el mismo código ASCII que recibió como entrada.

  Con estas funciones definidas se pueden trabajar muy bien muchas operaciones que se pueden realizar con caracteres. Por ejemplo, veamos un programa (Código 12.1.) que solicita caracteres por consola, hasta que recibe el carácter salto de línea, y que únicamente muestra por pantalla los caracteres introducidos que sean letras. Al final indicará cuántas veces han sido pulsadas cada una de las cinco vocales.

  Si el carácter introducido es alfabético, entonces simplemente verifica si es una vocal y aumenta el contador particular para cada vocal. Si no lo es, entonces imprime en pantalla un retorno de carro y un carácter blanco, de forma que borra el carácter que había sido introducido mediante la función `getchar` y que no deseamos que salga en pantalla.

  El carácter intro es el ASCII 13. Así lo hemos señalado en la condición que regula la estructura **do-while**.

Entrada de caracteres.

Hemos visto dos funciones que sirven bien para la introducción de caracteres.

La función `scanf` espera la entrada de un carácter más el carácter intro. No es muy cómoda cuando se espera únicamente un carácter.

La función `getchar` también está definida en la biblioteca `stdio.h`. De todas formas, su uso no es siempre el esperado, por problemas del buffer de teclado. Un buffer es como una cola o almacén que crea y gestiona el sistema operativo. Con frecuencia ocurre que la función `getchar` debería esperar la entrada por teclado de un carácter, pero no lo hace porque ya hay caracteres a la espera en el buffer gestionado por el sistema operativo.

Si se está trabajando con un editor en el sistema operativo Windows, se puede hacer uso de la biblioteca `conio.h` y de algunas de sus funciones de entrada por consola. Esta biblioteca no es estándar en ANSI C, pero si vamos a trabajar en ese sistema operativo sí resulta válida.

En esa biblioteca vienen definidas dos funciones útiles para la entrada por teclado, y que no dan los problemas que da, en Windows, la función `getchar`.

Esas dos funciones son:

`int` `getche(`**`void`**`);` espera del usuario un pulso de teclado. Devuelve el código ASCII del carácter pulsado y muestra por pantalla ese carácter. Al ser invocada esta función, no recibe valor o parámetro alguno: por eso se define como de tipo **`void`**.

`int` `getch(`**`void`**`);` espera del usuario un pulso de teclado. Devuelve el código ASCII del carácter pulsado. Al ser invocada esta función, no recibe valor o parámetro alguno: por eso se define como de tipo **`void`**. Esta función no tiene eco en pantalla, y no se ve el carácter pulsado.

Cadena de caracteres.

Una cadena de caracteres es una formación de caracteres. Es un vector tipo **`char`**, cuyo último elemento es el carácter nulo (carácter '\0' cuyo ASCII es 0). Toda cadena de caracteres termina con ese carácter. Este carácter indica donde termia la cadena.

La declaración de una cadena de caracteres se realiza de forma similar a la de un vector de cualquier otro tipo:

```
char mi_cadena[dimensión];
```

donde `dimensión` es un literal que indica el número de bytes que se deben reservar para la cadena (recuerde que una variable tipo **`char`** ocupa un byte).

La asignación de valores, cuando se crea una cadena, puede ser del mismo modo que la asignación de vectores:

```
char mi_cadena[4] = {'h', 'o', 'l', 'a'};
```

Y así hemos asignado a la variable `mi_cadena` el valor de la cadena `"hola"`.

Y así, con esta operación de asignación, acabamos de cometer un error importante. Repasemos… ¿qué es una cadena?: es un vector tipo **char**, cuyo último elemento es el carácter nulo. Pero si el último elemento es el carácter nulo… ¿no nos hemos equivocado en algo?: Sí.

La correcta declaración de esta cadena sería:

```
char mi_cadena[5] = {'h', 'o', 'l', 'a', '\0'};
```

Faltaba el carácter `'\0'` con el que debe terminar toda cadena. De todas formas, la asignación de valor a una cadena suele hacerse mediante **comillas dobles**, de la siguiente manera:

```
char mi_cadena[5] = "hola";
```

Y ya en la cadena `"hola"` se recoge el quinto carácter: el carácter nulo. Ya se encarga el compilador de introducirlo al final de la cadena.

Y es que hay que distinguir, por ejemplo, entre el carácter `'a'` y la cadena `"a"`. En el primer caso tratamos del valor ASCII 97, que codifica la letra `'a'` minúscula; en el segundo caso tratamos de una cadena de dos caracteres: el carácter `'a'` seguido del carácter nulo (cuyo valor ASCII es 0).

También podríamos haber hecho lo siguiente:

```
char mi_cadena[100] = "hola";
```

donde todos los bytes posteriores al carácter nulo tendrán valores aleatorios, no asignados. Pero eso no importa, porque la cadena sólo se extiende hasta el carácter nulo: no nos interesa lo que pueda haber más allá de ese carácter.

Y al igual que con los arrays, podemos inicializar la cadena, sin necesidad de dimensionarla:

```
char mi_cadena[] = "hola";
```

Y entonces ya se encarga el compilador de reservar cinco bytes para la variable `mi_cadena`.

Una forma de vaciar una cadena y asignarle el valor de cadena vacía es el siguiente:

```
mi_cadena[0] = 0; // Valor ASCII del carácter '\0'.
```

Definimos **longitud de la cadena** como el número de caracteres previos al carácter nulo. El carácter nulo no cuenta como parte para el cálculo de la longitud de la cadena. La cadena `"hola"` necesita cinco variables **char** para ser almacenada, pero su longitud se dice que es cuatro. Asignando al elemento de índice 0 el valor nulo, tenemos una cadena de longitud cero.

Cada elemento de la cadena se reconoce a través del índice entre corchetes. Cuando se quiere hacer referencia a toda la cadena en su conjunto, se emplea el nombre de la cadena sin ningún corchete ni índice.

Dar valor a una cadena de caracteres.

Para recibir cadenas por teclado disponemos, entre otras, de la función `scanf` ya presentada en capítulos anteriores. Basta indicar, entre comillas dobles, y después del carácter `'%'` la letra `'s'`. Ya lo vimos en el Capítulo 8.

Pero esta función toma la cadena introducida por teclado, y la corta a partir de la primera entrada de un carácter en blanco. Puede hacer un pequeño programa (Código 12.2.) para comprobar a qué nos estamos refiriendo.

Código 12.2.

```
#include <stdio.h>
int main(void)
{
	char cadena[100];

	printf("Valor de la cadena ... ");
	scanf(" %s", cadena);

	printf("Cadena introducida ... %s.\n", cadena);

	return 0;
}
```

Si introduce una cadena con un carácter en blanco (por ejemplo, si introduce "ABCDE ABCDE ABCDE") obtendrá una salida truncada desde el momento en que en la entrada encontró un primer espacio en blanco. La salida por pantalla de este programa es:

```
Cadena introducida ... ABCDE.
```

El motivo es que la función scanf toma los datos de entrada del buffer (porción temporal de memoria: ya se explicó en el Capítulo 8) del archivo estándar de entrada `stdin`. Cuando se pulsa la techa Enter (INTRO, Return, o como quiera llamarse) la función `scanf` verifica que la información recibida es correcta y, si lo es, almacena esa información recibida en el espacio de memoria indicado (`cadena`, en nuestro ejemplo). Y como `scanf` toma las palabras como cadenas de texto, y toma los espacios en blanco como separador de entradas para distintas variables, se queda con la primera entrada (hasta llegar a un espacio en blanco) y deja el resto en el buffer.

Podemos averiguar lo que hay en el buffer. Eso nos servirá para comprender mejor el comportamiento de la función `scanf` y comprender también su relación con el buffer.

Código 12.3.

```
#include <stdio.h>

int main(void)
{
	char a[100], c;
	printf("Introducir cadena de texto ... ");
	scanf(" %s", a);
	printf("\nCadena recibida por scanf ... %s", a);

	printf("\nQueda en el buffer .......... ");
	while((c = getchar()) != '\n')
		printf("%c", c);

	return 0;
}
```

Introduzca el programa Código 12.3. en su editor de C. Ejecute el programa, e introduzca por teclado, por ejemplo, la entrada "Programar es sencillo" (introduzca el texto sin comillas, claro); y pulse la tecla intro.

Ésta será la salida que, por pantalla, obtendrá:

```
Cadena recibida por scanf ... Programar
Queda en el buffer ..........  es sencillo
```

Se puede evitar ese corte, insertando en la cadena de control y entre corchetes algunos caracteres. Si se quiere utilizar la función scanf para tomar desde teclado el valor de una cadena de caracteres, sin tener problema con los caracteres en blanco, bastará poner:

```
scanf(" %[^\n]s, nombreCadena);
```

Entre corchetes ha quedado indicado el único carácter ante el que la función scanf considerará que se ha llegado al final de cadena: aquí ha quedado indicado, como carácter de fin de cadena, el de salto de línea.

Si en Código 12.2. cambiamos la llamada a la función scanf y ponemos:

```
scanf(" %[^\n]s", cadena);
```

E introducimos ahora la misma entrada, tendremos la siguiente salida:

```
Cadena introducida ... ABCDE ABCDE ABCDE.
```

Y en Código 12.3., si modificamos la sentencia scanf, y la dejamos `scanf(" %[^\n]s", a);` la salida por pantalla quedará:

```
Cadena recibida por scanf ... Programar es sencillo
Queda en el buffer ..........
```

Acabamos de introducir una herramienta sintáctica que ofrece la función scanf para filtrar las entradas de texto. Cuando la entrada de la función scanf es una cadena de texto, podemos indicarlo mediante la letra de tipo %s, o mediante %[], indicando entre corchetes qué conjunto de posibles caracteres se esperan como entrada, o cuáles no se esperan.

Veamos algunos ejemplos:

- `scanf(" %[A-Za-z]”, a);`

  La cadena estará formada por los caracteres introducidos mientras éstos sean letras mayúsculas o minúsculas. Si el usuario introduce "ABCDE12345ABCDE", la cadena a valdrá únicamente “ABCDE”: en cuanto haga su aparición el primer carácter no incluido en la lista entre corchetes se terminará la entrada de la cadena. El resto de la entrada quedará en el buffer de stdin.

- `scanf("%[^A-Z]", a);`

  En este caso, la función scanf no terminará su ejecución hasta que no reciba un carácter alfabético en mayúsculas.

  Por ejemplo, podría dar como entrada la siguiente cadena:

  ```
  "buenas tardes.
  este texto de prueba sigue siendo parte de la entrada.
  12345: se incluyen los dígitos.
  FIN"
  ```

La cadena a valdría entonces todos los caracteres introducidos desde la 'b' de "buenas" hasta el punto después de la palabra "dígitos". El siguiente carácter introducido es una letra mayúscula (la primera de la palabra "FIN", y entonces scanf interrumpe la entrada de la cadena. Esos tres últimos caracteres "FIN", quedarán en el buffer del archivo estándar stdin.

También se puede limitar el número de caracteres de la entrada: por ejemplo:

```
scanf(" %5s", a);
```

Si el usuario introduce la cadena "ABCDEabcdeABCDE", el valor de a será únicamente "ABCDE", que son los cinco primeros caracteres introducidos. El limitador de longitud de la cadena es muy útil a la hora de dar entrada a cadenas de texto, porque se evita una violación de memoria en el caso de que la cadena de entrada sea de longitud mayor que la longitud del array tipo **char** que se ha declarado.

Otra función, quizá más cómoda (pero que no goza de tantas posibilidades de control en el valor de la cadena de entrada), es la función gets. La función gets está también definida en la biblioteca stdio.h, y su prototipo es el siguiente:

```
char *gets(char *s);
```

Esta función asigna a la cadena s todos los caracteres introducidos como cadena. La función queda a la espera de que el usuario introduzca la cadena de texto. Hasta que no se pulse la tecla intro, se supone que todo lo que se teclee será parte de la cadena de entrada. Al final de todos ellos, como es natural, coloca el carácter nulo.

Hay que hacer una advertencia grave sobre el uso de estas funciones de entrada de cadenas de texto: puede ocurrir que la cadena introducida por teclado sea de mayor longitud que el número de bytes que se han reservado. Esa incidencia no es vigilada por la función gets. Y si ocurre,

entonces, además de grabar información de la cadena en los bytes reservados para ello, se hará uso de los bytes, inmediatamente consecutivos a los de la cadena, hasta almacenar toda la información tecleada más su carácter nulo final. Esa violación de memoria puede tener —y de hecho habitualmente tiene— consecuencias desastrosas para el programa.

En Código 12.4. se recoge un programa que solicita al usuario su nombre y lo muestra entonces por pantalla. El programa está bien construido. Pero hay que tener en cuenta que el nombre que se introduce puede, fácilmente, superar los 10 caracteres. Por ejemplo, si un usuario responde diciendo "José Antonio", ya ha introducido 13 caracteres: 4 por José, 7 por Antonio, 1 por el carácter en blanco, y otro más por el carácter nulo final. En ese caso, el comportamiento del programa sería imprevisible.

Por todo ello, está recomendado que NO se haga uso de esta función.

Código 12.4.

```
#include <stdio.h>
int main(void)
{
      char nombre[10];
      printf("¿Cómo te llamas?  ");
      gets(nombre);
      printf("Hola, %s.", nombre);
      return 0;
}
```

En el manual de prácticas puede encontrar referencia al uso de otras funciones de stdio.h (función fgets). También puede encontrar ayuda suficiente en las ayudas de los distintos IDE's o en Internet. El prototipo de esta función es:

```
char *fgets(char *cadena, int n, FILE *stream);
```

Donde el primer parámetro es la cadena de texto donde se guardará la entrada que se reciba por el archivo indicado en el tercer parámetro, y que si queremos que sea entrada por teclado deberemos indicar como `stdin`. El segundo parámetro es la longitud máxima de la entrada: si la cadena de entrada tiene, por ejemplo, un tamaño de 100 elementos, y así se indica a la llamada a la función, entonces al array se le asignarán, como máximo, 99 caracteres más el carácter de fin de cadena.

Esta función incluye el carácter intro como uno más en la entrada. Si el usuario introduce la cadena `"hola"` (y al final pulsará la tecla intro…) en la llamada a la función `fgets(entrada , 100 , stdin);`, entonces tendremos que `entrada[0]` vale `'h'`, y `entrada[3]` vale `'a'`. La función habrá asignado a `entrada[4]` el carácter de valor ASCII 10, que es el que corresponde a la tecla intro. Y `entrada[5]` valdrá, por fin, el carácter fin de cadena (valor ASCII 0). Lo que haya más allá de ese elemento ya no corresponde a la entrada recibida.

El estándar C11 ha definido nuevas funciones, sencillas de usar, (como la nueva función `gets_s`), que resuelven el problema de uso de esta función. No haga uso de estas funciones nuevas en C11 si su compilador no tiene incorporados estas novedades.

Operaciones con cadenas de caracteres.

Todo lo visto en el capítulo de vectores es aplicable a las cadenas: de hecho una cadena no es más que un vector de tipo **`char`**. Sin embargo, las cadenas merecen un tratamiento diferente, ya que las operaciones que se pueden realizar con ellas son muy distintas a las que se realizan con vectores numéricos: concatenar cadenas, buscar una subcadena en una cadena dada, ordenar alfabéticamente varias cadenas, etc. Vamos a ver algunos programas de manejo de cadenas, y presentamos algunas funciones definidas para ellas, disponibles en la biblioteca `string.h`.

- ***Copiar el contenido de una cadena en otra cadena***.

  Vea Código 12.5. Mientras no se llega al carácter nulo de `original`, se van copiando uno a uno los valores de las variables de la cadena en `copia`. Al final, cuando ya se ha llegado al nulo en original, habrá que cerrar también la cadena en `copia`, mediante un carácter nulo.

  Las líneas de código entre las líneas `/*01*/` y `/*02*/` son equivalentes a la línea `/*03*/`, donde se hace uso de la función `strcpy` de `string.h` cuy0 prototipo es:

  ```
  char *strcpy(char *dest, const char *src);
  ```

Código 12.5. Copia de una cadena en otra.

```
      #include <stdio.h>
      #define TAM_ 100
      int main(void)
      {
            char original[TAM_];
            char copia[TAM_];
            short int i = 0;

            printf("Cadena original ... ");
            gets(original);   // Uso desaconsejado.
/*01*/      while(original[i])
            {
                  copia[i] = original[i];
                  i++;
            }
/*02*/      copia[i] = 0;

            printf("Original: %s\n",original);
            printf("Copìa: %s\n",copia);
            return 0;
      }
```

```
/*03*/      strcpy(copia, original);      // En string.h
```

que recibe como parámetros las dos cadenas, origen (`src`) y destino (`dest`), y devuelve la dirección de la cadena de destino.

- ***Determinar la longitud de una cadena***.

  Vea Código 12.6. El contador `i` se va incrementando hasta llegar al carácter 0; así, en `i`, tenemos la longitud de la cadena. Esta operación también se puede hacer con la función `strlen` de `string.h` cuyo prototipo es:

  ```
  size_t strlen(const char *s);
  ```

  que recibe como parámetro una cadena de caracteres y devuelve su longitud. El tipo `size_t` es, para nosotros y ahora mismo, el tipo **`int`**.

  Así, las líneas `/*01*/` y `/*02*/` llegan al mismo resultado.

Código 12.6. Longitud de una cadena.

```
      #include <stdio.h>
      int main(void)
      {
            char or[100];
            short int i = 0;
            printf("Cadena original ... ");
            gets(or);
/*01*/      while(or[i]) i++;
            printf("%s tiene longitud %hd\n" , or , i);
            return 0;
}
```

```
/*02*/      i = strlen(original);   // En string.h
```

- ***Concatenar una cadena al final de otra***.

  Vea Código 12.7. El código comprendido entre las líneas marcadas con /**/ y /**/ es equivalente a la línea marcada con /**/, donde se hace

uso de la función de `string.h` que concatena cadenas: `strcat`. Esta función recibe como parámetros las cadenas destino de la concatenación y fuente, y devuelve la dirección de la cadena destino. Su prototipo es:

```
char *strcat(char *dest, const char *src);
```

También existe otra función, parecida a esta última, que concatena no toda la segunda cadena, sino hasta un máximo de caracteres, fijado por un tercer parámetro de la función:

```
char *strncat(char *dest, const char *src, size_t maxlen);
```

Código 12.7. Concatenar una cadena al final de otra.

```
      #include <stdio.h>
      int main(void)
      {
            char cad1[100];
            char cad2[100];
            short int i = 0, j = 0;

            printf("Primer tramo de cadena ... ");
            gets(cad1);
            printf("Tramo a concatenar ... ");
            gets(cad2);

/*01*/      while(cad1[i]) i++; // i: longitud de cad1.
            while(cad2[j])
            {
                  cad1[i++] = cad2[j++];
            }
/*02*/      cad1[i] = 0;

            printf("Texto concatenado: %s\n", cad1);

            return 0;
      }
```

```
/*03*/      strcat(cad1, cad2);
```

- ***Comparar dos cadenas e indicar cuál de ellas es mayor, o si son iguales***.

Código 12.8. Comparación de dos cadenas.

```
      #include <stdio.h>
      int main(void)
      {
            char c1[100] , c2[100];
            short int i = 0 , chivato = 0;
            printf("Primera Cadena ... ");     gets(c1);
            printf("Segunda Cadena ... ");     gets(c2);

/*01*/      while(c1[i] && c2[i] && !chivato)
            {
                  if(c1[i] > c2[i])       chivato = 1;
                  else if(c1[i] < c2[i])  chivato = 2;
                  i++;
/*02*/      }
/*03*/      if(chivato == 1)              printf("c1 > c2");
            else if(chivato == 2)         printf("c2 > c1");
            else if(!c1[i] && c2[i])      printf("c2 > c1");
            else if(c1[i] && !c2[i])      printf("c1 > c2");
/*04*/      else                          printf("c1 = c2");
            return 0;
      }
```

```
/*05*/      int comp = strcmp(c1,c2);
/*06*/      if(comp < 0)           printf("c2 > c1");
            else if(comp > 0)      printf("c1 > c2");
/*07*/      else                   printf("c1 = c2");
```

Vea una solución implementada en Código 12.8. La operación realizada entre las líneas /*01*/ y /*02*/ es la misma que la indicada en la línea /*05*/. La operación realizada entre las líneas /*03*/ y /*04*/ es la misma que la indicada entre las líneas /*06*/ y /*07*/.

La función `strcmp`, de `string.h`, tiene el siguiente prototipo:

```
int strcmp(const char *s1, const char *s2);
```

Es una función que recibe como parámetros las cadenas a comparar y devuelve un valor negativo si `s1 < s2`; un valor positivo si `s1 > s2`; y un cero si ambas cadenas son iguales.

El uso de mayúsculas o minúsculas influye en el resultado; por tanto, la cadena, por ejemplo, `"XYZ"` es anterior, según estos códigos, a la cadena `"abc"`.

También existe una función que compara hasta una cantidad de caracteres señalado, es decir, una porción de la cadena:

```
int strncmp(const char *s1, const char *s2, size_t maxlen);
```

donde `maxlen` es el tercer parámetro, que indica hasta cuántos caracteres se han de comparar.

Otras funciones de cadena.

Vamos a detenernos en la conversión de una cadena de caracteres, todos ellos numéricos, en la cantidad numérica, para poder luego operar con ellos. Las funciones que veremos en este epígrafe se encuentran definidas en otras bibliotecas: en la `stdlib.h` o en la biblioteca `math.h`.

- ***Convertir una cadena de caracteres (todos ellos dígitos o signo decimal) en un*** **`double`*.***

  ```
  double strtod(const char *s, char **endptr);
  ```

  Esta función convierte la cadena s en un valor **`double`**. La cadena s debe ser una secuencia de caracteres que puedan ser interpretados como un valor **`double`**. La forma genérica en que se puede presentar esa cadena de caracteres es la siguiente:

  ```
  [ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
  ```

donde [ws] es un espacio en blanco opcional; [sn] es el signo opcional (+ ó –); [ddd] son dígitos opcionales; [fmt] es el formato exponencial, también opcional, que se indica con las letras 'e' ó 'E'; finalmente, el [.] es el carácter punto decimal, también opcional. Por ejemplo, valores válidos serían + 1231.1981 e-1; ó 502.85E2; ó + 2010.952.

La función abandona el proceso de lectura y conversión en cuanto llega a un carácter que no puede ser interpretable como un número real. En ese caso, se puede hacer uso del segundo parámetro para detectar el error que ha encontrado: aquí se recomienda que para el segundo parámetro de esta función se indique el valor nulo: en esta parte del libro aún no se tiene suficiente formación en C para poder comprender y emplear bien este segundo parámetro.

La función devuelve, si ha conseguido la transformación, el número ya convertido en formato **double**.

Código 12.9. recoge un ejemplo de uso de esta función.

Código 12.9. Uso de la función strtod.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      char entrada[80];
      double valor;

      printf("Número decimal ... ");     gets(entrada);

      valor = strtod(entrada, 0);

      printf("La cadena es %s ", entrada);
      printf("y el número es %lf\n", valor);

      return 0;
}
```

De forma semejante se comporta la función `atof`, de la biblioteca `math.h`. Su prototipo es:

```
double atof(const char *s);
```

Donde el formato de la cadena de entrada es el mismo que hemos visto para la función `strtod`.

Consultando la ayuda del compilador se puede ver cómo se emplean las funciones `strtol` y `strtoul`, de la biblioteca `stdlib.h`: la primera convierte una cadena de caracteres en un entero largo; la segunda es lo mismo pero el entero es siempre largo sin signo. Y también las funciones `atoi` y `atol`, que convierte la cadena de caracteres a **`int`** y a **`long int`** respectivamente.

Ejercicios.

12.1. ***Escribir un programa que solicite del usuario la entrada de una cadena y muestre por pantalla en número de veces que se ha introducido cada una de las cinco vocales.***

Código 12.10. Posible solución al Ejercicio 12.1.

```
#include <stdio.h>
int main(void)
{
	char cadena[100];
	short int a, e, i, o, u, cont;

	printf("Cadena ... \n");
	gets(cadena);

	a = e = i = o = u = 0;
```

Código 12.10. (Cont.).

```
      for(cont = 0 ; cadena[cont] ; cont++)
      {
            if(cadena[cont] == 'a') a++;
            else if(cadena[cont] == 'e') e++;
            else if(cadena[cont] == 'i') i++;
            else if(cadena[cont] == 'o') o++;
            else if(cadena[cont] == 'u') u++;
      }
      printf("Las vocales introducidas han sido ... \n");
      printf("a ... %hd\n",a);
      printf("e ... %hd\n",e);
      printf("i ... %hd\n",i);
      printf("o ... %hd\n",o);
      printf("u ... %hd\n",u);

      return 0;
}
```

12.2. ***Escribir un programa que solicite del usuario la entrada de una cadena y muestre por pantalla esa misma cadena en mayúsculas.***

Código 12.11. Posible solución al Ejercicio 12.2.

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
      char cadena[100];
      short int cont;

      printf("Cadena de texto ... \n");
      gets(cadena);
```

Código 12.11. (Cont.).

```
      for(cont = 0 ; cadena[cont] ; cont++)
            cadena[cont] = toupper(cadena[cont]);

      printf("Cadena introducida...%s\n",cadena);
      return 0;
}
```

12.3. ***Escribir un programa que solicite del usuario la entrada de una cadena y elimine de ella todos los espacios en blanco. Imprima luego por pantalla esa cadena.***

Código 12.12. Posible solución al Ejercicio 12.3.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
      char cadena[100];
      short int i, j;

      printf("Introduzca una cadena de texto ... \n");
      gets(cadena);

      for(i = 0 ; cadena[i] ; )
            if(cadena[i] == 32)
                  for(j = i ; cadena[j] ; j++)
                        cadena[j] = cadena[j + 1];
            else i++;

      printf("Cadena introducida %s\n" , cadena);
      return 0;
}
```

Este problema puede resolverse de dos maneras. La primera (Código 12.12), sería haciendo la copia sin espacios en blanco en la misma cadena de origen: es decir, trabajando con una única cadena de texto, que se modifica eliminando de ella todos los caracteres en blanco. La segunda (Código 12.13.) es creando una segunda cadena y asignándole a ésta los valores de los caracteres de la primera que no sean el carácter blanco.

Si el carácter `i` es el carácter blanco (ASCII 32) entonces no se incrementa el contador sino que se adelantan una posición todos los caracteres hasta el final. Si el carácter no es el blanco, entonces simplemente se incrementa el contador y se sigue rastreando la cadena de texto.

Código 12.13. Posible solución al Ejercicio 12.3.

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

int main(void)
{
      char original[100], copia[100];
      short int i, j;

      printf("Introduzca una cadena de texto ... \n");
      gets(original);

      for(i = 0 , j = 0 ; original[i] ; i++)
            if(original[i] != 32)
                  copia[j++] = original[i];
      copia[j] = 0;

      printf("La cadena introducida ha sido ...\n");
      printf("%s\n",original);
      printf("La cadena transformada es ...\n");
      printf("%s\n",copia);
      return 0;
}
```

Esta segunda forma (Código 12.13.) es, desde luego, más sencilla. Una observación conviene hacer aquí: en la segunda solución, al acabar de hacer la copia, hemos cerrado la cadena `copia` añadiéndole el carácter nulo de fin de cadena. Esa operación también se debe haber hecho en la primera versión de resolución del ejercicio, pero ha quedado implícita en el segundo anidado **`for`**. Intente comprenderlo.

12.4. ***Escriba un programa que solicite al usuario una cadena de texto y genere luego otra a partir de la primera introducida donde únicamente se recojan los caracteres que sean letras: ni dígitos, ni espacios, ni cualquier otro carácter. El programa debe, finalmente, mostrar ambas cadenas por pantalla.***

Código 12.14. Posible solución al Ejercicio 12.4.

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

int main(void)
{
	char cadena1[100], cadena2[100];
	short i, j;

	printf("Primera cadena ... ");
	gets(cadena1);

	for(i = 0 , j = 0 ; i < 100 && cadena1[i] ; i++)
		if(isalpha(cadena1[i]))
			cadena2[j++] = cadena1[i];
	cadena2[j] = '\0';

	printf("Cadena primera: %s\n", cadena1);
	printf("Cadena segunda: %s\n", cadena2);
	return 0;
}
```

12.5. ***Escriba un programa que reciba una cadena de texto y busque los caracteres que sean de tipo dígito (0, 1, …, 9) y los sume.***

Por ejemplo, si la cadena de entrada es "Hoy es 12 de septiembre de 2008", el cálculo que debe realizar es la suma 1 + 2 (correspondientes al texto "12") + 2 + 0 + 0 + 8 (correspondientes al texto "2008").

Código 12.15. Posible solución al Ejercicio 12.5.

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

int main(void)
{
      char cadena[100];
      short i;
      long Suma = 0;

      printf("Introduzca la cadena ... ");     gets(cadena);

      for(i = 0 ; i < 100 && cadena[i] ; i++)
            if(isdigit(cadena[i])) Suma += cadena[i] - '0';
      printf("El valor de la suma es ... %ld", Suma);
      return 0;
}
```

Observación a Código 12.15.: el carácter '0' no tiene el valor numérico 0, ni el carácter '1' el valor numérico 1,... Se calcula fácilmente el valor de cualquier carácter dígito de la siguiente forma: Del carácter dígito '1' llegamos al valor numérico 1 restando '1' – '0'. Del carácter dígito '2' llegamos al valor numérico 2 restando '2' – '0'. Y así sucesivamente: del carácter dígito '9' llegamos al valor numérico 9 restando '0' a '9'.

12.6. ***Escriba la salida que, por pantalla, ofrece el programa recogido en Código 12.16.***

Código 12.16. Código correspondiente al enunciado del Ejercicio 12.6.

```
#include <stdio.h>
#include <string.h>
#define LNG 100

int main(void)
{
	char cad1[LNG] = "3 DE SEPTIEMBRE DE 2011";
	char cad2[LNG];
	short int i, l;

	i = l = strlen(cad1);
	while(i) cad2[i - 1] = cad1[l - i--];
	cad2[l] = '\0';

	printf("cad1: %s\n", cad1);
	printf("cad2: %s\n", cad2);

	return 0;
}
```

Este programa copia la cadena `cad1` en la cadena `cad2`, pero colocando los caracteres en orden inverso. La salida del programa es la siguiente:

```
cad1: 15 DE AGOSTO DE 2012
cad2: 2102 ED OTSOGA ED 51
```

12.7. ***Escriba un programa que reciba del usuario y por teclado una cadena de texto de no más de 100 caracteres y a partir de ella se genere una cadena nueva con todas las letras minúsculas y***

en la que se haya eliminado cualquier carácter que no sea alfanumérico.

Ayuda: En la biblioteca ctype.h dispone de una función isalnum que recibe como parámetro un carácter y devuelve un valor verdadero si ese carácter es alfanumérico; de lo contrario el valor devuelto es cero. Y también dispone de la función tolower que recibe como parámetro un carácter y devuelve el valor del carácter en minúscula si la entrada ha sido efectivamente un carácter alfabético; y devuelve el mismo carácter de la entrada en caso contrario.

Mostramos en Código 12.17., una posible solución. Convendrá hacer uso de la función `isalnum` y la macro `tolower`, de `ctype.h`.

Código 12.17. Posible solución al Ejercicio 12.7.

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

int main(void)
{
      char in[100], cp[100];
      short i, j;

      printf("Cadena de entrada ... ");  gets(in);

      for(i = 0 , j = 0 ; i < 100 && in[i] ; i++)
            if(isalnum(in[i])) cp[j++] = tolower(in[i]);
      cp[j] = '\0';

      printf("Cadena copia ... %s", cp);
      return 0;
}
```

12.8. ***Escriba un programa que solicite del usuario dos cadenas de texto de cinco elementos (la longitud máxima de las cadenas será por tanto de 4 caracteres) formadas por sólo caracteres de dígito ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9'), y que muestre por pantalla la suma de esas dos entradas.***

En la solución que se propone (Código 12.18.), no se hace una verificación previa sobre la correcta entrada realizada por el usuario. Se supone que el usuario ha introducido dos enteros de, como mucho, 4 dígitos. Si no es así, el programa no ofrecerá un resultado válido.

Lo que hacemos es convertir la cadena de dígitos en su valor entero. Se explica el procedimiento con un ejemplo: supongamos que la entrada para la cadena `c1` ha sido 579: es decir, `c1[0] = '5', c1[1] = '7', c[2] = '9'; c1[3]` = carácter nulo de fin de cadena; `c1[4]`: no se le ha asignado ningún valor.

Lo que hace el programa que proponemos como solución es recorrer el array desde el primer elemento y hasta llegar al carácter nulo. Inicializada a cero, toma la variable `n1` y le asigna, para cada valor del índice `i` que recorre el array, el valor que tenía antes multiplicado por 10, al que le suma el nuevo valor del array en la posición `i`.

Cuando `i = 0` y `n1` vale inicialmente 0, se lee en valor `c1[0]` (valor 5). Entonces `n1` pasa a valer `n1 * 10 + 5`, es decir, 5.

Cuando `i = 1` y `n1` vale 5, se lee en valor `c1[1]` (valor 7). Entonces `n1` pasa a valer `n1 * 10 + 7`, es decir, 57.

Cuando `i = 2` y `n1` vale 57, se lee en valor `c1[2]` (valor 9). Entonces `n1` pasa a valer `n1 * 10 + 9`, es decir, 579.

Ahora leerá el siguiente valor que será el de fin de cadena, y terminará el proceso de encontrar el valor numérico del entero introducido como cadena de caracteres.

Como ya ha quedado explicado antes (cfr. Ejercicio 12.5.) para obtener el valor numérico de un carácter dígito basta restar a su valor el ASCII del carácter '0'.

Código 12.18. Posible solución al Ejercicio 12.8.

```
#include <stdio.h>
int main(void)
{
      char c1[5], c2[5];
      short n1, n2, i;

      printf("Primera cadena ... ");     gets(c1);
      printf("Segunda cadena ... ");     gets(c2);

      for(i = 0 , n1 = 0 ; c1[i] && i < 5 ; i++)
            n1 = n1 * 10 + (c1[i] - '0');

      for(i = 0 , n2 = 0 ; c2[i] && i < 5 ; i++)
            n2 = n2 * 10 + (c2[i] - '0');

      printf("La suma vale ... %hd", n1 + n2);

      return 0;
}
```

12.9. ***Declare dos cadenas de texto y asígneles valor mediante entrada de teclado. Mediante una función de string.h determine si la primera cadena es menor, igual o mayor que la segunda cadena.***

El concepto de menor y mayor no tiene aquí relación con la longitud de la cadena, sino con el contenido y, en concreto, con el orden alfabético. Diremos, por ejemplo, que la cadena "C" es mayor que la cadena

"AAAAA", y que la cadena "ABCDE" es menor que la cadena "ABCDEF". En Código 12.19. se ofrece una posible solución.

Código 12.19. Posible solución al Ejercicio 12.9.

```
#include <stdio.h>
#include <string.h>
#define LLL 100

int main(void)
{
      char a[LLL], b[LLL];

// ENTRADA DE LA CADENA a ...

      printf("Cadena a ... ");      gets(a);
      printf("Cadena b ... ");      gets(b);

// DECIR CUAL DE LAS DOS CADENAS ES MAYOR
      if(strcmp(a , b) < 0)
            printf("%s menor que %s.\n", a, b);
      else if(strcmp(a, b) > 0)
            printf("%s mayor que %s.\n", a, b);
      else
            printf("Cadenas iguales.\n");

      return 0;
}
```

CAPÍTULO 13

ÁMBITO Y VIDA DE LAS VARIABLES.

Este breve capítulo pretende completar algunos conceptos presentados en el capítulo 7 sobre los tipos de datos y variables. Ahora que hemos avanzado en el conocimiento del lenguaje C y que ya sabemos desarrollar nuestros pequeños programas, estos conceptos pueden ser comprendidos con claridad. También presentamos una breve descripción de cómo se gestiona las variable en la memoria del ordenador.

Ámbito y Vida.

Entendemos por **ámbito de una variable** la porción de código, dentro de un programa, en el que esta variable tiene significado. Hasta el momento todas nuestras variables han tenido como ámbito todo el programa, y quizá ahora no es sencillo hacerse una idea intuitiva de este concepto; pero realmente, no todas las variables están "en activo" a lo largo de todo el programa.

Además del ámbito, existe otro concepto, que podríamos llamar **extensión o tiempo de vida**, que define el intervalo de tiempo en el que el espacio de memoria reservado por una variable sigue en reserva; cuando la variable "muere", ese espacio de memoria vuelve a estar disponible para otros usos que el ordenador requiera. También este concepto quedará más aclarado a medida que avancemos en este breve capítulo.

El almacenamiento de las variables y la memoria.

Para comprender las diferentes formas en que se puede crear una variable, es conveniente describir previamente el modo en que se dispone la memoria de datos en el ordenador.

Hay diferentes espacios donde se puede ubicar una variable declarada en un programa:

1. **Registros**. El registro es el elemento más rápido de almacenamiento y acceso a la memoria. La memoria de registro está ubicada directamente dentro del procesador. Sería muy bueno que toda la memoria fuera de estas características, pero de hecho el número de registros en el procesador está muy limitado. El compilador decide qué variables coloca en estas posiciones privilegiadas. El programador no tiene baza en esa decisión. El lenguaje C permite sugerir, mediante algunas palabras clave, la conveniencia o inconveniencia de que una determinada variable se cree en este espacio privilegiado de memoria.
2. **La Pila**. La memoria de pila reside en la memoria RAM (*Random Access Memory*: memoria de acceso aleatorio) De la memoria RAM es de lo que se habla cuando se anuncian "los megas" o "gigas" que tiene la memoria de un ordenador.

   El procesador tiene acceso y control directo a la pila gracias al "puntero de pila", que se desplaza hacia abajo cada vez que hay que

reservar más memoria para una nueva variable, y vuelve a recuperar su posición hacia arriba para liberar esa memoria. El acceso a la memoria RAM es muy rápido, sólo superado por el acceso a registros. El compilador debe conocer, mientras está creando el programa, el tamaño exacto y la vida de todas y cada una de las variables implicadas en el proceso que se va a ejecutar y que deben ser almacenados en la pila: el compilador debe generar el código necesario para mover el puntero de la pila hacia abajo y hacia arriba. Esto limita el uso de esta buena memoria tan rápida. Hasta el capítulo 10, cuando hablemos de la asignación dinámica de la memoria, todas las variables que empleamos pueden existir en la pila, e incluso algunas de ellas en las posiciones de registro.

3. **El montículo**. Es un espacio de memoria, ubicada también en la memoria RAM, donde se crean las variables de asignación dinámica. Su ventaja es que el compilador no necesita, al generar el programa, conocer cuánto espacio de almacenamiento necesita asignar al montículo para la correcta ejecución del código compilado. Esta propiedad ofrece una gran flexibilidad al código de nuestros programas. A cambio hay que pagar un precio con la velocidad: lleva más tiempo asignar espacio en el montículo que tiempo lleva hacerlo en la pila.
4. **Almacenamiento estático**. El almacenamiento estático contiene datos que están disponibles durante todo el tiempo que se ejecuta el programa. Más adelante, en este capítulo, veremos cómo se crean y qué características tienen las variables estáticas.
5. **Almacenamiento constante**. Cuando se define un valor constante, éste se ubica habitualmente en los espacios de memoria reservados para el código del programa: lugar seguro, donde no se ha de poder cambiar el valor de esa constante.

Variables Locales y Variables Globales.

Una variable puede definirse fuera de la función principal o de cualquier función. Esas variables se llaman **globales**, y son válidas en todo el código que se escriba en ese programa. Su espacio de memoria queda reservado mientras el programa esté en ejecución. **Diremos que son variables globales, que su ámbito es todo el programa y que su vida perdura mientras el programa esté en ejecución**.

Código 13.1.

```
long int Fact;
#include <stdio.h>
int main(void)
{
      short int n;

      printf("Introduce el valor de n ... ");
      scanf(“ %hd",&n);
      printf("El factorial de %hd es ... ",n);

      Fact = 1;
      while(n) Fact *=n--;

      printf("%ld",Fact);
      return 0;
}
```

Veamos como ejemplo el programa implementado en Código 13.1. La variable n es local: su ámbito es sólo el de la función principal `main`. La variable `Fact` es global: su ámbito se extiende a todo el programa.

Advertencia: salvo para la declaración de variables globales (y declaración de funciones, que veremos más adelante), el lenguaje C no admite ninguna otra sentencia fuera de una función. Ahora mismo este concepto nos queda fuera de intuición porque no hemos visto aún la

posibilidad de crear y definir en un programa otras funciones, aparte de la función principal. Pero esa posibilidad existe, y en ese caso, si una variable es definida fuera de cualquier función, entonces esa variable es accesible desde todas las funciones del programa.

No se requiere ninguna palabra clave del lenguaje C para indicar al compilador que esa variable concreta es global. Es suficiente con ubicar la declaración fuera del ámbito de cualquier bloque de código.

Se recomienda, en la medida de lo posible, **no hacer uso de variables globales**. Cuando una variable es manipulable desde cualquier ámbito de un programa es fácil sufrir efectos imprevistos.

Una variable será **local** cuando se crea en un bloque del programa, que puede ser una función, o un bloque interno de una función. Por ejemplo, vea las líneas recogidas en Código 13.2. La variable y tiene su ámbito (y su vida) únicamente dentro del bloque donde ha sido creada. Fuera de ese ámbito, no se puede acceder a esa variable.

Código 13.2.

```
long x = 12;
// Sólo x está disponible.
{   // inicio de un bloque: crea un ámbito
      long y = 25;
// Tanto x como y están disponibles.
}
// La variable y está fuera de ámbito. Ha terminado su vida.
```

El ámbito de una variable local es el del bloque en el que está definida. En C, puede declararse una variable local, con el mismo nombre de una variable más global: en ese ámbito no se tendrá acceso a la variable global. Vea las líneas recogidas en Código 13.3.: dentro del ámbito creado por las llaves sólo puede accederse a la x creada dentro de él.

Código 13.3.

```
long x = 12;
// Sólo x está disponible.
{
      long x = 25;
// En este bloque la única variable x accesible vale 25.
}
// La única variable x en este ámbito vale 12.
```

También pueden definirse variables locales del mismo nombre en ámbitos diferentes y disjuntos. Al no coincidir en ámbito en ninguna sentencia, no puede haber equívoco y cada variable, del mismo nombre, existe sólo en su propio ámbito. Vea las líneas recogidas en Código 13.4.

Código 13.4.

```
long x = 12;
// Sólo x está disponible.
{
      long y = 25;
// Tanto x como y están disponibles.
}// La variable y está fuera de ámbito. Terminó su vida.
{
      long y = 40;
// Tanto x como y están disponibles.
}
// La variable y está fuera de ámbito. Terminó su vida.
```

Veamos (Código 13.5.) un ejemplo sencillo de uso de diferentes variable locales. En este código, que como vimos permite buscar los números perfectos entre los primeros 10000 enteros, declara dos variables (`j` y `suma`) en el bloque de la estructura del primer **`for`**. Al terminar la ejecución del **`for`** gobernado por la variable `i`, esas dos variables dejan

de existir. Si a su vez, la estructura **for** más externa estuviera integrada dentro de otra estructura de iteración, cada vez que se volviera a ejecutar ese **for** se volverían a crear esas dos variables, que tendrían el mismo nombre, pero no necesariamente las mismas direcciones.

Código 13.5.

```
unsigned short i;
for(i = 2 ; i < 10000 ; i++)
{
      unsigned short suma = 1;
      unsigned short j;

      for(j = 2 ; j <= i / 2 ; j++)
            if(i % j == 0) suma += j;

      if(suma == i) printf("%hu" , i);
}
```

Hay una palabra en C, que rara vez se usa, para indicar que la variable es local. Es la palabra reservada **auto**. El compilador descubre siempre el ámbito de una variable gracias al lugar donde ésta se declara.

Un ejemplo muy simple (Código 13.6.) puede ayudar a presentar estas ideas de forma más clara. En este código, las variables b y c han sido declaradas globales, e inicializadas a cero. Luego, dentro de la función principal, se ha declarado, local dentro del **for**, la variable c. Y dentro del **for** se han variado los valores de las variables b y c.

Una aclaración previa: cuando al inicio de en un bloque iterado se declara una variable, esa sentencia de declaración sólo se ejecuta en la primera iteración. No se vuelve a ejecutar en cada iteración la sentencia de declaración. Y si en esa declaración se asigna a dicha variable un valor inicial, tampoco se ejecuta la asignación en cada nueva iteración.

Código 13.6.

```
#include <stdio.h>

long b = 0, c = 0;
int main(void)
{
	for(b = 0 ; b < 10 ; b++)
	{
		long c = 0;
		c++;
	}
	printf("El valor de b es %ld y el de c es %ld", b, c);
	return 0;
}
```

¿Cuál es la salida que ofrecerá por pantalla este código? Por lo que respecta a la variable b no hay ninguna duda: se ha incrementado diez veces, y su valor, después de ejecutar la estructura `for`, será 10. Pero, ¿y c? Esta variable ha sufrido también una variación y ha llegado al valor 10. Pero… ¿cuál de los dos variables c ha cambiado?: la de ámbito más local. Y como la sentencia que ejecuta la función `printf` ya está fuera de la estructura **`for`**, y para entonces la variable local c ya ha muerto, la variable c que muestra la función `printf` no es otra que la global: la única viva en este momento. La salida que mostrará el programa es la siguiente:

```
El valor de b es 10 y el de c es 0.
```

Una advertencia importante: ya se ha visto que se pueden declarar, en ámbitos más reducidos, variables con el mismo nombre que otras que ya existen en ámbitos más globales. Lo que no se puede hacer es declarar, en un mismo ámbito, dos variables con el mismo nombre. Ante esa circunstancia, el compilador dará error y no compilará.

Variables estáticas y dinámicas.

Con respecto a la extensión o tiempo de vida, las variables pueden ser estáticas o dinámicas. Será **estática** aquella variable que una vez definida, persiste hasta el final de la ejecución del programa. Y será **dinámica** aquella variable que puede ser creada y destruida durante la ejecución del programa.

No se requiere ninguna palabra clave para indicar al compilador que una variable creada es dinámica: toda variable lo es por defecto. Sí es en cambio necesario indicar al compilador, mediante la palabra clave **`static`**, cuándo queremos que una variable sea creada estática. Esa variable puede ser local, y en tal caso su ámbito será local, y sólo podrá ser usada cuando se estén ejecutando sentencias de su ámbito; pero su extensión será la misma que la del programa, y siempre que se vuelvan a las sentencias de su ámbito allí estará la variable, ya creada, lista para ser usada. Cuando terminen de ejecutarse las sentencias de su ámbito esas posiciones de memoria no serán accesibles, porque estaremos fuera de ámbito, pero tampoco podrá hacerse uso de esa memoria para otras variables, porque la variable estática seguirá "viva" y esa posición de memoria sigue almacenando el valor que quedó de la última vez que se ejecutaron las sentencias de su ámbito.

Cuando se crea una variable local dentro de una bloque, o dentro de una función, el compilador reserva espacio para esa variable cada vez que se llama a la función: mueve en cada ocasión hacia abajo el puntero de pila tanto como sea preciso para volver a crear esa variable. Si existe un valor inicial para la variable, la inicialización se realiza cada vez que se pasa por ese punto de la secuencia.

Si se quiere que el valor permanezca durante la ejecución del programa entero, y no sólo cada vez que se entra de nuevo en el ámbito de esa variable, entonces tenemos dos posibilidades: La primera consiste en crear esa variable como global, extendiendo su ámbito al ámbito de todo el programa (en este caso la variable no queda bajo control del bloque

donde queríamos ubicarla, o bajo control único de la función que la necesita, sino que es accesible (se puede leer y se puede variar su valor) desde cualquier sentencia del programa); La segunda consiste en crear una variable **static** dentro del bloque o función. El almacenamiento de esa variable no se lleva a cabo en la pila sino en el área de datos estáticos del programa. La variable sólo se inicializa una vez —la primera vez que se llama a la función—, y retiene su valor entre diferentes invocaciones.

Código 13.7.

```
#include <stdio.h>

int main(void)
{
      long i, j;
      for(i = 0 ; i < 3 ; i++)
            for(j = 0 ; j < 4 ; j++)
            {
                  static long a = 0;
                  long b = 0;

                  a += 5;
                  b += 5;

                  printf("a = %3ld.   b = %3ld.\n", a , b);
            }

      return 0;
}
```

Veamos el ejemplo recogido en Código 13.7., donde tenemos dos variables locales que sufren las mismas operaciones: una estática (la variable que se ha llamado a) y la otra no (la que se ha llamado b). El programa ofrece la siguiente salida por pantalla:

```
a =   5.   b =   5.
a =  10.   b =   5.
a =  15.   b =   5.
a =  20.   b =   5.
a =  25.   b =   5.
a =  30.   b =   5.
a =  35.   b =   5.
a =  40.   b =   5.
a =  45.   b =   5.
a =  50.   b =   5.
a =  55.   b =   5.
a =  60.   b =   5.
```

Comprobamos, pues, que la variable a mantiene su valor para cada nueva iteración, mientras que la variable b se inicializa a cero en cada iteración.

Variables en registro.

Cuando se declara una variable, se reserva un espacio de memoria para almacenar sus sucesivos valores. Cuál sea ese espacio de memoria es cuestión que no podemos gobernar del todo. Especialmente, como ya se ha dicho, no podemos decidir cuáles son las variables que deben ubicarse en los espacios de registro.

El compilador, al traducir el código, puede detectar algunas variables empleadas repetidamente, y decidir darles esa ubicación preferente. En ese caso, no es necesario traerla y llevarla de la ALU a la memoria y de la memoria a la ALU cada vez que hay que operar con ella.

El programador puede tomar parte en esa decisión, e indicar al compilador que convendría ubicar en los registros de la ALU alguna o algunas variables. Eso se indica mediante la palabra clave **register**.

Si al declarar una variable, se precede a toda la declaración la palabra **register**, entonces esa variable queda creada en un registro de la ALU. Una variable candidata a ser declarada **register** es, por ejemplo, las que actúan de contadoras en estructuras **for**.

También puede ocurrir que no se desee que una variable sea almacenada en un registro de la ALU. Y quizá se desea indicar al compilador que, sea cual sea su opinión, una determinada variable no debe ser almacenada allí sino en la memoria, como una variable cualquiera normal. Para evitar que el compilador decida otra cosa se le indica con la palabra **volatile**.

El compilador tomas las indicaciones de **register** a título orientativo. Si, por ejemplo, se ha asignado el carácter de **register** a más variables que permite la capacidad de la ALU, entonces el compilador resuelve el conflicto según su criterio, sin abortar el proceso de compilación.

Variables ***extern***.

Aunque estamos todavía lejos de necesitar este tipo de declaración, presentamos ahora esta palabra clave de C, que hace referencia al ámbito de las variables.

El lenguaje C permite trocear un problema en diferentes módulos que, unidos luego, formarán la aplicación. Esos módulos muchas veces serán programas independientes que se compilan por separado y finalmente se "linkan" o se juntan. Debe existir la forma de indicar, a cada uno de esos programas desarrollados por separado, la existencia de variables globales comunes para todos ellos. Variables cuyo ámbito trasciende el ámbito del programa donde se declaran, porque abarcan todos los programas que luego, "linkados", darán lugar a la aplicación final.

Se podrían declarar todas las variables en todos los archivos. C en la compilación de cada programa por separado no daría error, y asignaría tanta memoria como veces estuvieran declaradas. Pero en el enlazado daría error de duplicidad.

Para evitar ese problema, las variable globales que deben permanecer en todos o varios de los módulos de un programa se declaran como **extern** en todos esos módulos excepto en uno, donde se declara como

variable global sin la palabra **extern**. Al compilar entonces esos módulos, no se creará la variable donde esté puesta la palabra **extern**, y permitirá la compilación al considerar que, en alguno de los módulos de linkado, esa variable sí se crea. Evidentemente, si la palabra **extern** se coloca en todos los módulos, entonces en ninguno se crea la variable y se producirá error en el linkado.

El identificador de una variable declarada como **extern** es conveniente que no tenga más de seis caracteres, pues en los procesos de linkado de módulos sólo los seis primeros caracteres serán significativos.

En resumen…

Ámbito: El ámbito es el lugar del código donde las sentencias pueden hacer uso de una variable.

Una variable **local** queda declarada en el interior de un bloque. Puede indicarse ese carácter de local al compilador mediante la palabra auto. De todas formas, la ubicación de la declaración ofrece suficientes pistas al compilador para saber de la localidad de cada variable. Su ámbito queda localizado únicamente a las instrucciones que quedan dentro del bloque donde ha sido creada la variable.

Una variable es **global** cuando queda declarada fuera de cualquier bloque del programa. Su ámbito es todo el programa: cualquier sentencia de cualquier función del programa puede hacer uso de esa variable global.

Extensión: La extensión es el tiempo en que una variable está viva, es decir, en que esa variable sigue existiendo en la memoria.

Una variable global debe existir mientras el programa esté en marcha, puesto que cualquier sentencia del programa puede hacer uso de ella.

Una variable local sólo existe en el intervalo de tiempo transcurrido desde la ejecución de la primera sentencia del bloque donde se ha

creado esa variable y hasta que se sale de ese bloque. Es, tras la ejecución de la última sentencia del bloque, el momento en que esa variable desaparece. Si el bloque vuelve a ejecutarse entonces vuelve a crearse una variable con su mismo nombre, que se ubicará donde antes, o en otra dirección de memoria diferente: es, en todo caso, una variable diferente a la anterior.

Se puede forzar a que una variable local exista durante toda la ejecución del programa. Eso puede hacerse mediante la palabra reservada de C **`static`**. En ese caso, al terminar la ejecución de la última instrucción del bloque donde está creada, la variable no desaparece. De todas formas, mientras no se vuelva a las sentencias de ese bloque, esa variable no podrá ser reutilizada, porque fuera de ese bloque, aún estando viva, está fuera de su ámbito.

Ubicación: Podemos indicar al compilador si queremos que una variable sea creada en los registros de la ALU, utilizando la palabra reservada **`register`**. Podemos indicarla también al compilador que una variable no se cree en esos registros, mediante la palabra reservada **`volatile`**. Fuera de esas indicaciones que da el programador, el compilador puede decidir qué variables se crean en la ALU y cuáles en la memoria principal.

No se ha dicho nada en este capítulo sobre la creación de espacios de memoria con valores constantes. Ya se presentó la forma de hacerlo en el capítulo 2 sobre tipos de datos y variables en C. Una variable declarada como **`const`** quedará almacenada en el espacio de memoria de las instrucciones. No se puede modificar (mediante el operador asignación) el valor de una variable definida como **`const`**. Por eso, al crear una variable de esta forma hay que asignarle valor en su declaración.

Ejercicios.

13.1. ***Haga un programa que calcule el máximo común divisor de dos enteros introducidos por consola. El proceso de cálculo se repetirá hasta que el usuario introduzca un par de ceros.***

El programa mostrará por pantalla, cada vez que ejecute el código del bucle, un valor contador que se incrementa y que indica cuántas veces se está ejecutando a lo largo de toda la aplicación. Esa variable contador será declarada como `static`***.***

Código 13.8. Posible solución para el Ejercicio 13.1.

```
#include <stdio.h>
int main(void)
{
	unsigned short a, b, mcd;
	do
	{
		printf("Valor de a ... ");	scanf(" %hu",&a);
		printf("Valor de b ... ");	scanf(" %hu",&b);

		if(a == 0 && b == 0) break;
		while(b)
		{
			static unsigned short cont = 0;
			mcd = b;
			b = a % b;
			a = mcd;
			printf("\ncont = %hu", ++cont);
		}

		printf("\n\nEl mcd es %hu.", mcd);
	}while(1);

	return 0;
}
```

Código 13.8. recoge una posible solución al problema propuesto.

Cada vez que se ejecuta el bloque de la estructura **do–while** se incrementa en uno la variable `cont`. Esta variable se inicializa a cero únicamente la primera vez que se ejecuta la sentencia **while** de cálculo del máximo común divisor.

Observación: quizá podría ser interesante, que al terminar de ejecutar todos los cálculos que desee el usuario, entonces se mostrara por pantalla el número de veces que se ha entrado en el bucle. Pero eso no es posible tal y como está el código, puesto que fuera del ámbito de la estructura **while** que controla el cálculo del máximo común divisor, la variable `cont`, sigue viva, pero estamos fuera de ámbito y el compilador no reconoce ese identificador como variable existente.

CAPÍTULO 14

FUNCIONES.

Hemos llegado a las funciones.

Al principio del tema de estructuras de control señalábamos que había dos maneras de romper el flujo secuencial de sentencias. Y hablábamos de dos tipos de instrucciones que rompen el flujo: las instrucciones condicionales y de las incondicionales. Las primeras nos dieron pie a hablar largamente de las estructuras de control: condicionales y de iteración. Ahora toca hablar de las instrucciones incondicionales que realizan la transferencia a una nueva dirección del programa sin evaluar condición alguna. Eso lo hacen las llamadas a las funciones.

De hecho, ya hemos visto muchas funciones. Y las hemos utilizado. Cuando hemos querido mostrar por pantalla un mensaje hemos acudido a la función `printf` de la biblioteca `stdio.h`. Cuando hemos querido saber la longitud de una cadena hemos utilizado la función `strlen`, de la biblioteca `string.h`. Y cuando hemos querido hallar la función seno de un valor concreto, hemos acudido a la función `sin`, de `math.h`.

Y ya hemos visto que, sin saber cómo, hemos echado mano de una función estándar programada por ANSI C que nos ha resuelto nuestro problema. ¿Cómo se ha logrado que se vea en pantalla un texto, o el valor de una variable? ¿Qué desarrollo de Taylor se ha aplicado para llegar a calcular el seno de un ángulo dado? No lo sabemos. ¿Dónde está el código que resuelve nuestro problema? Tampoco lo sabemos. Pero cada vez que hemos invocado a una de esas funciones, lo que ha ocurrido es que el contador de programa ha abandonado nuestra secuencia de sentencias y se ha puesto con otras sentencias, que son las que codifican las funciones que hemos invocado.

De forma incondicional, cada vez que se invoca una función, se transfiere el control de ejecución a otra dirección de memoria, donde se encuentran codificadas otras sentencias, que resuelven el problema para el que se ha definido, editado y compilado esa función.

Son transferencias de control con retorno. Porque cuando termina la ejecución de la última de las sentencias de la función, entonces regresa el control a la sentencia inmediatamente posterior a aquella que invocó esa función.

Quizá ahora, cuando vamos a abordar la teoría de creación, diseño e implementación de funciones, será buen momento para releer lo que decíamos en capítulos 5 y 6 al tratar de la modularidad. Y recordar también las tres propiedades que debían tener los distintos módulos: independencia funcional, comprensibilidad, adaptabilidad. No lo vamos a repetir ahora: allí se trató.

Definiciones.

Abstracción – Modularidad – Programación estructurada.

Esas eran las tres notas básicas que presentamos al presentar el lenguaje de programación C. De la programación estructurada ya hemos hablado, y lo seguiremos haciendo en este capítulo. La abstracción es el

paso previo de toda programación: conocer el sistema e identificar los más significativos elementos que dan con la esencia del problema a resolver. Y la modularidad es la capacidad de dividir el sistema estudiado en partes diferenciadas.

Eso que hemos llamado módulo, en un lenguaje de programación se puede llamar procedimiento o se puede llamar función. Las funciones y los procedimientos permiten crear programas complejos, mediante un reparto de tareas que permite construir el programa de forma estructurada y modular.

Desde un punto de vista académico, se entiende por **procedimiento** el conjunto de sentencias a las que se asocia un identificador (un nombre), y que realiza una tarea que se conoce por los cambios que ejerce sobre el conjunto de variables. Y entendemos por **función** el conjunto de sentencias a las que se asocia un identificador (un nombre) y que genera un valor nuevo, calculado a partir de los argumentos que recibe.

Los elementos que componen un procedimiento o función son, pues:

1. Un **identificador**, que es el nombre que sirve para invocar a esa función o a ese procedimiento.
2. Una lista de **parámetros**, que es el conjunto de variables que se facilitan al procedimiento o función para que realice su tarea modularizada. Al hacer la abstracción del sistema, y modularlo en partes más accesibles, hay que especificar los parámetros formales que permiten la comunicación y definen el dominio (tipo de dato) de los datos de entrada. Esa lista de parámetros define el modo en que podrán comunicarse el programa que utiliza a la función y la función usada.
3. Un **cuerpo** o conjunto de sentencias. Las necesarias para poder realizar la tarea para la que ha sido definida la función o el procedimiento.
4. Un **entorno**. Entendemos por entorno el conjunto de variables

globales, y externas por tanto al procedimiento o función, que pueden ser usadas y modificadas dentro del ámbito de la función. Esas variables, por ser globales y por tanto definidas en un ámbito más amplio al ámbito local de la función, no necesitan ser explicitadas en la lista de parámetros de la función.

Es una práctica desaconsejable trabajar con el entorno de la función desde el ámbito local de la función. Hacerlo lleva consigo que esa función deja de ser independiente de ese entorno y, por tanto, deja de ser exportable. Perderíamos entonces el valor de la independencia funcional, que es una de las propiedades de la programación por módulos.

Podemos pues concluir que el uso de variables globales dentro del cuerpo de un procedimiento o función es altamente desaconsejable.

En el lenguaje C no se habla habitualmente de procedimientos, sino sólo de funciones. Pero de hecho existen de las dos cosas. Procedimientos serían, por ejemplo, la función `printf` no se invoca para calcular valores nuevos, sino para realizar una tarea sobre las variables. Más claro se ve con la función `scanf` que, efectivamente, realiza una tarea que se conoce por los cambios que ejerce sobre una variable concreta. Y funciones serían, por ejemplo, la función `strlen`, que a partir de una cadena de caracteres que recibe como parámetro de entrada calcula un valor, que es la longitud de esa cadena; o la función sin, que a partir de un ángulo que recibe como valor de entrada, calcula el seno de ese ángulo como valor de salida.

En definitiva, una **función** es una porción de código, identificada con un nombre concreto (su identificador), que realiza una tarea concreta, que puede ser entendida de forma independiente al resto del programa, y que tiene muy bien determinado cómo se hace uso de ella, con qué parámetros se la invoca y bajo qué condiciones puede ser usada, cuál es la tarea que lleva a cabo, y cuál es el valor que calcula y devuelve.

Tanto los procedimientos como las funciones pueden ser vistos como cajas negras: un código del que desconocemos sus sentencias, al que se le puede suministrar unos datos de entrada y obtener modificaciones para esos valores de entrada y/o el cálculo de un nuevo valor, deducido a partir de los valores que ha recibido como entrada.

Con eso se consigue programas más cortos; que el código pueda ser usado más de una vez; mayor facilidad para gestionar un correcto orden de ejecución de sentencias; que las variables tengan mayor carácter local, y no puedan ser manipuladas fuera del ámbito para el que han sido creadas.

Funciones en C.

Una **función**, **en C**, es un segmento independiente de código fuente, diseñado para realizar una tarea específica. Esta tarea, o bien es el cálculo de un resultado que se recibe en el ámbito donde se invocó la función llamada, o bien realiza alguna operación de salida de información por pantalla, o en algún archivo de salida, o en la red.

Las funciones son los elementos principales de un programa en C. Cada una de las funciones de un programa constituye una unidad, capaz de realizar una tarea determinada. Quizá se podría decir que un programa es simplemente un conjunto de definiciones de distintas funciones, empleadas luego de forma estructurada.

La primera función que aparece en todo programa C es la **función principal**, o función `main`. Todo programa ejecutable tiene una, y sólo una, función `main`. Un programa sin función principal no genera un ejecutable. Y si todas las funciones se crean para poder ser utilizadas, la función principal es la única que no puede ser usada por nadie: nadie puede invocar a la función principal de un programa. Tampoco puede llamarse a sí misma (este concepto de "*autollamada*", denominado recurrencia, lo trataremos más adelante en el siguiente capítulo).

Además de la función principal, en un programa se pueden encontrar otras funciones: o funciones creadas y diseñadas por el programador para esa aplicación, o funciones ya creadas e implementadas y compiladas en librerías: de creación propia o adquirida o pertenecientes al estándar de ANSI C.

Las funciones estándar de ANSI C se encuentran clasificadas en distintas librerías de acuerdo con las tareas que desarrollan. Al montar un programa en C, se buscan en las librerías las funciones que se van a necesitar, que se incluyen en el programa y se hacen así parte del mismo.

También se pueden crear las propias funciones en C. Así, una vez creadas y definidas, ya pueden ser invocadas tantas veces como se quiera. Y así, podemos ir creando nuestras propias bibliotecas de funciones.

Siempre que hemos hablado de funciones hemos utilizado dos verbos, uno después del otro: creación y definición de la función. Y es que en una función hay que distinguir entre su declaración o prototipo (creación de la función), su definición (el cuerpo de código que recoge las sentencias que debe ejecutar la función para lograr llevar a cabo su tarea) y, finalmente, su invocación o llamada: una función creada y definida sólo se ejecuta si otra función la invoca o llama. Y en definitiva, como la única función que se ejecuta sin ser invocada (y también la única función que no permite ser invocada) es la función `main`, cualquier función será ejecutada únicamente si es invocada por la función `main` o por alguna función que ha sido invocada por la función `main` o tiene en su origen, en una cadena de invocación, una llamada desde la función `main`.

Declaración de la función.

La declaración de una función se realiza a través de su **prototipo**. Un prototipo tiene la forma:

```
tipo_funcion nombre_funcion
            ([tipo1 [var1][,... tipoN [varN]]);
```

Donde `tipo_funcion` declara de qué tipo es el valor que devolverá la función. Una función puede devolver valores de cualquier tipo de dato válido en C, tanto primitivo como diseñado por el programador (se verá la forma de crear tipos de datos en unos temas más adelante). Si no devuelve ningún valor, entonces se indica que es de tipo **`void`**.

Donde `tipo1,..., tipoN` declara de qué tipo es cada uno de los valores que la función recibirá como parámetros al ser invocada. En la declaración del prototipo es opcional indicar el nombre que tomarán las variables que recibirán esos valores y que se comportarán como variables locales de la función. Sea como sea, ese nombre sí deberá quedar recogido en la definición de la función. Pero eso es adelantar acontecimientos.

Al final de la declaración viene el punto y coma. Y es que la declaración de una función es una sentencia en C. Una sentencia que se consigna fuera de cualquier función. La declaración de una función tiene carácter global dentro de programa donde se declara. No se puede declarar, ni definir, una función dentro de otra función: eso siempre dará error de compilación.

Toda función que quiera ser definida e invocada debe haber sido previamente declarada. El prototipo de la función presenta el modo en que esa función debe ser empleada. Es como la definición de su *interface*, de su forma de comunicación: qué valores, de qué tipo y en qué orden debe recibir la función como argumentos al ser invocada. El prototipo permite localizar cualquier conversión ilegal de tipos entre los argumentos utilizados en la llamada de la función y los tipos definidos

en los parámetros, entre los paréntesis del prototipo. Además, controla que el número de argumentos usados en una llamada a una función coincida con el número de parámetros de la definición.

Existe una excepción a esa regla: cuando una función es de tipo **int**, puede omitirse su declaración. Si en una expresión, en una sentencia dentro del cuerpo de una función, aparece un nombre o identificador que no ha sido declarado previamente, y ese nombre va seguido de un paréntesis de apertura, el compilador supone que ese identificador corresponde al nombre de una función de tipo **int**. Es recomendable no hacer uso de esa excepción.

Todas las declaraciones de función deben preceder a la definición del cuerpo de la función `main`.

Definición de la función.

Ya tenemos la función declarada. Con el prototipo ha quedado definido el modo en que podemos utilizarla: cómo nos comunicamos nosotros con ella y qué resultado nos ofrece.

Ahora queda la tarea de definirla.

Hay que escribir el código, las sentencias, que van a realizar la tarea para la que ha sido creada la función.

La forma habitual que tendrá la definición de una función la conocemos ya, pues hemos visto ya muchas: cada vez que hacíamos un programa, y escribíamos la función principal, estábamos definiendo esa función `main`. Esa forma es:

```
tipo_funcion nombre_funcion([tipo1 var1][,... tipoN varN])
{
      [declaración de variables locales]
      [cuerpo de la función: grupo de sentencias]
      [return(parámetro);]
}
```

Donde el `tipo_funcion` debe coincidir con el de la declaración, lo mismo que `nombre_funcion` y lo mismo que la lista de parámetros. Ahora, en la definición, los parámetros de la función siguen recogiendo el tipo de dato y el nombre de la variable: pero ahora ese nombre NO es opcional. Debe ponerse, porque esos nombres serán los identificadores de las variables que recojan los valores que se le pasan a la función cuando se la llama o invoca. A esas variables se las llama parámetros formales: son variables locales a la función: se crean cuando la función es invocada y se destruyen cuando se termina la ejecución de la función.

La lista de parámetros puede ser una lista vacía porque no se le quiera pasar ningún valor a la función: eso es frecuente. En ese caso, tanto en el prototipo como en la definición, entre los paréntesis que siguen al nombre de la función se coloca la palabra clave **`void`**.

```
tipo_funcion nombre_funcion(void); // declaración prototipo
```

Si la función no devuelve valor alguno, entonces se indica como de tipo **`void`**, al igual que ya se hizo en la definición del prototipo. Una función declarada como de tipo **`void`** no puede ser usada como operando en una expresión de C, porque esa función no tiene valor alguno. Una función de tipo **`void`** puede mostrar datos por pantalla, escribir o leer ficheros, etc.

En el bloque de la función podemos encontrar, en primer lugar, la declaración de las variables locales; y luego, el cuerpo de la función, donde están las sentencias que llevarán a cabo la tarea para la que ha sido creada y definida la función.

El bloque de la función viene recogido entre llaves. Aunque la función tenga una sola sentencia, es obligatorio recoger esa sentencia única entre las llaves de apertura y de cerrado.

Las variables creadas en el cuerpo de la función serán locales a ella. Se pueden usar identificadores idénticos para nombrar distintas variables de diferentes funciones, porque cada variable de cada función pertenece

a un ámbito completamente disjunto al ámbito de otra función, y no hay posibilidad alguna de confusión.

Todas las funciones en C están en el mismo nivel de ámbito, es decir, no se puede declarar ninguna función dentro de otra función, y no se puede definir una función como bloque interno en el cuerpo de otra función.

Hay una sentencia siempre especial en el cuerpo de cualquier función: la sentencia **`return`**, que ya hemos utilizado innumerables veces en la función `main`. Ésta interrumpe la ejecución de las sentencias de la función y devuelve el control del programa a la función que invoco a esa otra que ha ejecutado la sentencia **`return`**. Esta sentencia permite que a la función se le pueda asignar un valor concreto (excepto en el caso de que la función sea de tipo **`void`**) del mismo tipo de dato que el tipo de la función. Una sentencia **`return`** debe ir seguida de una expresión que pueda ser evaluada como un valor del tipo de la función. Por eso, en el ámbito de esta función llamante, invocar a una función equivale a poner un valor concreto del tipo de la función: el valor que se haya recibido con la sentencia **`return`**. Desde luego, si la función es de tipo **`void`** entonces esa función no puede ser empleada en ninguna expresión, porque, de hecho, esa función no devuelve valor de ningún tipo.

Por último, y esto es importante, una función puede tener tantas sentencias **`return`** como sean necesarias. Evidentemente, en ese caso, éstas deben estar condicionadas: sería absurdo tener cualquier código más allá de una sentencia **`return`** no condicionada por nada.

Llamada a la función.

La llamada a una función es una sentencia habitual en C. Ya la hemos usado con frecuencia, invocando hasta el momento únicamente funciones de biblioteca. Pero la forma de invocar es la misma para cualquier función.

```
nombre_funcion([argumento1][, ..., argumentoN]);
```

La sentencia de llamada está formada por el nombre de la función y sus argumentos (los valores que se le pasan) que deben ir recogidos en el mismo orden que la secuencia de parámetros del prototipo y entre paréntesis. Si la función no recibe parámetros (porque así esté definida), entonces se coloca después de su nombre los paréntesis de apertura y cerrado sin ninguna información entre ellos. Si no se colocan los paréntesis, se produce un error de compilación.

El paso de parámetros en la llamada exige una asignación para cada parámetro. El valor del primer argumento introducido en la llamada a la función queda asignado en la variable del primer parámetro formal de la función; el segundo valor de argumento queda asignado en el segundo parámetro formal de la función; y así sucesivamente. Hay que asegurar que el tipo de dato de los parámetros formales es compatible en cada caso con el tipo de dato usado en lista de argumentos en la llamada de la función. El compilador de C no dará error si se fuerzan cambios de tipo de dato incompatibles, pero el resultado será inesperado totalmente.

La lista de argumentos estará formada por nombres de variables que recogen los valores que se desean pasar, o por literales. No es necesario (ni es lo habitual) que los identificadores de los argumentos que se pasan a la función cuando es llamada coincidan con los identificadores de los parámetros formales.

Las llamadas a las funciones, dentro de cualquier función, pueden realizarse en el orden que sea necesario, y tantas veces como se quiera, independientemente del orden en que hayan sido declaradas o definidas. Incluso se da el caso, bastante frecuente como veremos más adelante, que una función pueda llamarse a sí misma. Esa operación de autollamada se llama recurrencia.

Si la función debe devolver un valor, con cierta frecuencia interesará que la función que la invoca almacene ese valor en una variable local suya. En ese caso, la llamada a la función será de la forma:

```
variable = nombre_funcion([argumento1][, …, argumentoN]);
```

Aunque eso no siempre se hace necesario, y también con frecuencia encontraremos las llamadas a las funciones como partes de una expresión.

La sentencia `return`.

Hay dos formas ordinarias de terminar la ejecución de una función.

1. Llegar a la última sentencia del cuerpo, antes de la llave que cierra el bloque de esa función.
2. Llegar a una sentencia **`return`**. La sentencia **`return`** fuerza la salida de la función, y devuelve el control del programa a la función que la llamó, en la sentencia inmediatamente posterior a la de la llamada a la función.

Si la función es de un tipo de dato distinto de **`void`**, entonces en el bloque de la función debe recogerse, al menos, una sentencia **`return`**. En ese caso, además, en esa sentencia y a continuación de la palabra **`return`**, deberá ir el valor que devuelve la función: o el identificador de una variable o un literal, siempre del mismo tipo que el tipo de la función o de otro tipo compatible.

Una función tipo **`void`** no necesariamente tendrá la sentencia **`return`**. En ese caso, la ejecución de la función terminará con la sentencia última del bloque. Si una función de tipo **`void`** hace uso de sentencias **`return`**, entonces en ningún caso debe seguir a esa palabra valor alguno: si así fuera, el compilador detectará un error y no compilará el programa.

La sentencia **`return`** puede encontrarse en cualquier momento del código de una función. De todas formas, no tendría sentido recoger ninguna sentencia más allá de una sentencia **`return`** que no estuviera condicionada, pues esa sentencia jamás llegaría a ejecutarse.

En resumen, la sentencia **`return`** realiza básicamente dos operaciones:

1. Fuerza la salida inmediata del cuerpo de la función y se vuelve a la siguiente sentencia después de la llamada.
2. Si la función no es tipo **void**, entonces además de terminar la ejecución de la función, devuelve un valor a la función que la llamó. Si esa función llamante no recoge ese valor en una variable, el valor se pierde, con todas las variables locales de la función abandonada.

La forma general de la sentencia **return** es:

```
return [expresión];
```

Muchos programadores habitúan a colocar la expresión del **return** entre paréntesis. Es opcional, como lo es en la redacción de cualquier expresión.

Si el tipo de dato de la expresión del **return** no coincide con el tipo de la función entonces, de forma automática, el tipo de dato de la expresión se convierte en el tipo de dato de la función.

Código 14.1. Ejemplo función `mostrar()`.

```
// Declaración.
void mostrar(short);
// Definición.
void mostrar(short x)
{
      printf("El valor recibido es %hd. ", x);
}
```

Ha llegado el momento de ver algunos ejemplos. Veamos primero una función de tipo **void**: una que muestre un mensaje por pantalla: cfr. Código 14.1. La llamada a esta función la haríamos, por ejemplo, con la sentencia

```
mostrar(10);
```

invocamos a la función, que ofrece la siguiente salida por pantalla:

```
El valor recibido es 10.
```

Otro ejemplo: Una función que reciba un entero y devuelva el valor de su cuadrado (cfr. Código 14.2.).

Código 14.2. Ejemplo función `cuadrado()`.

```
// Declaración.
unsigned long int cuadrado(short);
// Definición.
unsigned long int cuadrado(short x)
{
      return x * x;
}
```

Una posible llamada a la función `cuadrado`:

```
printf("El cuadrado de %hd es %ld.\n ", a, cuadrado(a));
```

Código 14.3. Ejemplo función `mayor()`.

```
// Declaración.
short mayor(short, short);
// Definición.
short mayor(short x, short y)
{
      if(x > y) return x;
      else return y;
}
// Otra forma de definirla con el siguiente return:
// x > y ? return x : return y;
// Y otra forma más:
// return x > y ? x : y;
```

Un tercer ejemplo, ahora con dos sentencias **return**: una función que reciba como parámetros formales dos valores enteros y devuelve el valor del mayor de los dos (cfr. Código 14.3.).

Desde luego la palabra **else** podría omitirse, porque jamás se llegará a ella si se ejecuta el primer **return**, y si la condición del **if** es falsa, entonces se ejecuta el segundo **return**.

La llamada a esta última función sería, por ejemplo:

```
A = mayor(a , b);
```

Donde la variable A guardará el mayor de los dos valores entre a y b.

Ámbito y vida de las variables.

Ya conocemos el concepto de ámbito de la variable. Y ahora que ya sabemos algo de las funciones, es conveniente presentar cuándo se puede acceder a cada variable, cuándo diremos que está viva, etc.

Veamos un programa ya conocido (cfr. Código 14.4.), el del cálculo del factorial de un entero, resuelto ahora mediante funciones.

En este programa, la función principal `main` tiene definida una variable de tipo **short**, a la que hemos llamado n. En esa función, esa variable es local, y podemos recoger sus características en la cuádrupla:

<n, **short**, R_n, V_n>

La variable, de tipo **short**, n, se crea en la dirección de memoria R_n y guardará el valor que reciba de la función `scanf`.

La función `main` invoca a la función `Factorial`. En la llamada se pasa como parámetro el valor de la variable n. En esa llamada, el valor de la variable n se copia en la variable a de `Factorial`:

<a, **short**, R_a, V_n>

Código 14.4. Ejemplo función `Factorial()`.

```
#include <stdio.h>
long Factorial(short);

int main(void)
{
      short n;

      printf("Introduzca el valor de n ... ");
      scanf(" %hd", &n);

      printf("El factorial de %hd ",n);
      printf("es %ld" , Factorial(n));

      return 0;
}

long Factorial(short a)
{
      long F = 1;
      while(a) F *= a--;
      return F;
}
```

Desde el momento en que se produce la llamada a la función `Factorial`, abandonamos el ámbito de la función `main`. En este momento, la variable `n` está fuera de ámbito y no puede, por tanto hacerse uso de ella. No ha quedado eliminada: estamos en el ámbito de `Factorial` pero aún no han terminado todas las sentencias de `main`. En el cálculo dentro de la función `Factorial` se ve modificado el valor de la variable local `a`. Pero esa modificación para nada influye en la variable `n`, que está definida en otra posición de memoria distinta.

Cuando se termina la ejecución de la función `Factorial`, el control del programa vuelve a la función `main`. La variable `a` y la variable `F` mueren, pero el valor de la variable `F` ha sido recibido como parámetro en la

función printf, y así podemos mostrarlo por pantalla. Ahora, de nuevo en la función principal, volvemos al ámbito de la variable n, de la que podríamos haber hecho uso si hubiera sido necesario.

Veamos ahora otro ejemplo, con un programa que calcule el máximo común divisor de dos enteros (cfr. Código 14.5.). De nuevo, resolvemos el problema mediante funciones.

Código 14.5. Ejemplo función euclides().

```
#include <stdio.h>
long euclides(long, long);

int main(void)                          {
      long n1, n2;

      do                                {
            printf("Valor de n1 ... ");   scanf(" %ld", &n1);
            printf("Valor de n2 ... ");   scanf(" %ld", &n2);
            if(n2 != 0)                 {
                  printf("\nEl mcd de %ld y %ld es %ld\n",
                        n1 , n2 , euclides(n1 , n2));
            }
      }while(n2 != 0);
      return 0;
}

long euclides(long a, long b)       {
      static short cont = 0;
      long mcd;

      while(b)                          {
            mcd = b;
            b = a % b;
            a = mcd;
      }
      printf("Invocaciones a la función ... %hd\n", ++cont);
      return mcd;
}
```

En esta ocasión, además, hemos incluido una variable **static** en la función euclides. Esta variable nos informará de cuántas veces se ha ejecutado la función.

Las variables n1 y n2, de main, dejan de estar accesibles cuando se invoca a la función euclides. En ese momento se copian sus valores en las variables a y b que comienzan a existir precisamente en el momento de la invocación de la función. Además de esas variables locales, y de la variable local mcd, se ha creado otra, llamada cont, que es también local a euclides pero que, a diferencia de las demás variables locales, no desaparecerá cuando se ejecute la sentencia **return** y se devuelva el control de la aplicación a la función main: es una variable declarada **static**. Cuando eso ocurra, perderá la variable cont su ámbito, y no podrá ser accedida, pero en cuanto se invoque de nuevo a la función euclides, allí estará la variable, ya creada, accesible para cuando la función la requiera.

Recapitulación.

Hemos visto cómo crear bloques de código, que pueden luego ser invocados, mediante un nombre para que realicen una determinada función. Gracias a la llamada a uno de estos bloques de código, que hemos llamado funciones, podemos obtener un resultado calculado a partir de valores iniciales que la función recibe como parámetros. La función termina su ejecución entregando un valor concreto de un tipo de dato predeterminado y establecido en su prototipo.

A lo largo de varias páginas del Capítulo 6 de este manual se ha presentado un ejemplo de modularidad, mostrando el desarrollo de tres funciones sencillas más la función main. Quizá ahora convenga retornar a una lectura rápida de ese Capítulo 6 (ahora con un mejor conocimiento del contexto de trabajo) y detenerse especialmente en ese ejercicio del Triángulo de Tartaglia allí presentado y resuelto.

Ejercicios.

14.1. ***Escribir un programa que solicite al usuario dos enteros y calcule, mediante una función, el máximo común divisor. Definir otra función que calcule el mínimo común múltiplo, teniendo en cuenta que siempre se verifica que*** $a \times b = mcd(a,b) \times mcm(a,b)$.

Código 14.6. Solución al Ejercicio 14.1. propuesto.

```
#include <stdio.h>

// Declaración de las funciones ...
short mcd(short, short);
long mcm(short, short);

// Función principal...
int main(void)
{
      short a, b;

      printf("Valor de a ... ");    scanf(" %hd",&a);
      printf("Valor de b ... ");    scanf(" %hd",&b);

      printf("El MCD de %hd y %hd es %hd", a, b, mcd(a, b));
      printf("\ny el MCM es %ld.", mcm(a, b));

      return 0;
}
/* ------------------------------------------------------ */
/*     Definición de las funciones                        */
/* ------------------------------------------------------ */
/* Función cálculo del máximo común divisor. ------------ */
short mcd(short a, short b)
{
      short m;
```

Código 14.6. (Cont.).

```
      while(b)
      {
            m = a % b;
            a = b;
            b = m;
      }
      return a;
}

{
      short m;
      while(b)
      {
            m = a % b;
            a = b;
            b = m;
      }
      return a;
}

/* Función cálculo del mínimo común múltiplo. ----------- */
long mcm(short a, short b)
{
      return a * (long)b / mcd(a, b);
}
```

14.2. ***Haga un programa que calcule el término n (a determinar en la ejecución del programa) de la serie de Fibonacci. El programa deberá utilizar una función, llamada fibonacci, cuyo prototipo sea***

```
unsigned long fibonacci(short);
```

Que recibe como parámetro el valor de n, y devuelve el término n-ésimo de la Serie.

Código 14.7. Solución al Ejercicio 14.2. propuesto.

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

unsigned long fibonacci(short);

int main(void)
{
      short N;

      printf("Término de la serie: ");   scanf(" %hd", &N);
      printf("\nEl término %hd es %lu.", N, fibonacci(N));
      return 0;
}

/* ------------------------------------------------------ */
/*     Definición de las funciones                        */
/* ------------------------------------------------------ */

/* Función Fibonacci. ---------------------------------- */
unsigned long fibonacci(short x)
{
      unsigned long fib1 = 1 , fib2 = 1, Fib = 1;

      while(x > 2)
      {
            Fib = fib1 + fib2;
            fib1 = fib2;
            fib2 = Fib;
            x--;
      }
      return Fib;
}
```

14.3. ***Escriba una función que reciba un entero y diga si es o no es perfecto (devuelve 1 si lo es; 0 si no lo es). Utilice esa función para mostrar los números perfectos entre dos enteros introducidos por el usuario.***

Código 14.8. Solución al Ejercicio 14.3. propuesto.

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

// Declaración de la función perfecto ...
short perfecto(long);

// Función principal ...
int main(void)
{
      long a, b, i;

      printf("Limite inferior ... ");    scanf(" %ld",&a);
      printf("Limite superior ... ");    scanf(" %ld",&b);

      for(i = a ; i <= b; i++)
            if(perfecto(i)) printf("%6ld", i);

      return 0;
}

/* ------------------------------------------------------ */
/*     Definición de las funciones                        */
/* ------------------------------------------------------ */
// Función perfecto ...
short perfecto(long x)
{
      long suma = 1;

      for(long div = 2 ; div <= x / 2 ; div++)
            if(x % div == 0) suma += div;

      return suma == x;
}
```

La función perfecto devuelve simplemente el valor verdadero o falso (`1` ó `0`) que se evalúa en la sentencia **`return`**: `suma == x;`

14.4. ***El binomio de Newton proporciona la expansión de las potencias de una suma:***

$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot x^{n-k} \cdot y^k$ ***, donde*** $\binom{n}{k} = n!/k! \cdot (n - k)!$

Escriba un programa que solicite al usuario los valores de x, y, n y muestre entonces por pantalla el valor $(x+y)^n$ ***usando la expansión del binomio de Newton.***

Sugerencias:

(1) Emplear la función `double pow(double, double);` ***(primer parámetro: base; segundo parámetro: exponente) recogida en*** `math.h`***.***

(2) Será más cómodo definir una función que realice el cálculo del factorial.

Código 14.9. Solución al Ejercicio 14.4. propuesto.

```
#include <stdio.h>
#include <math.h>
long factorial(short);

long factorial(short n)
{
      long F = 1;
      while(n) F *= n--;
      return F;
}

int main(void)
{
      short n, k;
      double x, y, sumando, binomio = 0;

      printf("Valor de n ... "); scanf(" %hd", &n);
      printf("Valor de x ... "); scanf(" %lf", &x);
      printf("Valor de y ... "); scanf(" %lf", &y);
```

Código 14.9. (Cont.).

```
      for(k = 0 ; k <= n ; k++)
      {
            sumando = factorial(n)
            sumando /= factorial(k);
            sumando /= factorial(n – k);
            sumando *= pow(x, n – k);
            sumando *= pow(y, k);
            binomio += sumando;
      }
      printf("Resultado = %lf.", binomio);

      return 0;
}
```

14.5. ***Se llama PRIMO PERFECTO a aquel entero primo n que verifica que ((n - 1)) / 2 también es primo. Por ejemplo, n = 11 es primo y ((n - 1)) / 2 = 5 también es primo: luego n = 11 es un primo perfecto.***

Haga un programa que muestre los primos perfectos menores que 1000.

Ya hemos visto en anteriores capítulos cómo determinar si un entero es primo o compuesto. Ahora también tiene que ser primo, además del número, su mitad menos uno. Hay que hacer dos veces, por tanto, la comprobación. Sin duda, el código se simplificaría mucho si definiéramos una función, llamada, por ejemplo, `int esprimo(long)`, que devolviera un valor distinto de cero cuando el entero que se recibe como parámetro es un valor primo. Código 14.10. ofrece una posible solución. Efectivamente, la definición de la función principal logra una gran simplicidad, gracias a la declaración de la segunda función.

Código 14.10. Solución al Ejercicio 14.5. propuesto.

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

int esprimo(long);

int main(void)
{
          long N;
          for(N = 5 ; N < 1000 ; N += 2)
            if(esprimo(N) && esprimo((N – 1) / 2))
                  printf("%ld\n", N);
          return 0;
}

int esprimo(long x)
{
          short d;
          if(x % 2 == 0) return 0;
          for(d = 3 ; d <= sqrt(x) ; d += 2)
            if(x % d == 0) return 0;
          return 1;
}
```

14.6. ***Suponga que ha definido una función cuyo prototipo es***

```
long isPrime(long);
```

Dicha función devuelve el mismo entero que recibió como parámetro si éste era primo, y devuelve el menor de sus divisores propios, si la entrada correspondía a un número compuesto.

Utilizando esta función, escriba un programa (función principal) que muestre por pantalla todos los factores primos de un entero introducido por el usuario.

El enunciado es parecido al anterior, porque de nuevo utilizamos una función que determina su el entero que recibe como parámetro es primo o compuesto. Pero ahora queremos que la función dé algo más de información: en caso de que el entero no sea primo, queremos que devuelva el entero más pequeño que lo divide. Así, como se verá, podremos obtener la factorización de un entero de una forma sencilla y con una función `main` fácil de implementar. Código 14.11. recoge una posible implementación con una solución al problema propuesto.

Código 14.11. Solución al Ejercicio 14.6. propuesto.

```
#include <math.h>
#include <stdio.h>
long isPrime(long a)
{
      long d;
      for(d = 2 ; d <= sqrt(a) ; d++)
            if(a % d == 0) return d;
      return a;
}

int main(void)
{
      long x;  // Valor de entrada a factorizar.
      long d;  // Recoge la salida de isPrime().

      printf("Introduce x ... ");  scanf(" %ld", &x);

      printf("1\t");  // El 1 es el primer factor.
      while((d = isPrime(x)) != 1)
      {
            printf("%ld\t", d); //Sucesivos factores.
            x /= d;
      }
      return 0;
}
```

14.7. ***Escriba el código de una función, cuyo prototipo es***

```
short digito(char);
```

que recibe un carácter y devuelve el valor -1 si dicho carácter no es numérico; y en caso contrario devuelve su valor como número. Por ejemplo, si recibe el carácter '3' devuelve el valor 3; si recibe el carácter 'p' devuelve el valor -1.

En Código 14.12. se ofrecen dos implementaciones: una usando a su vez la función `isdigit`, de `ctype.h`, y otra sin hacer uso de ella.

Código 14.12. Solución al Ejercicio 14.7. propuesto.

```
#include <ctype.h>
short digito(char a)
{
      return isdigit(a) ? a - '0' : -1;
}
```

```
short digito(char a)
{
      return a >= ‘0’ && a <= ‘9’ ? a - '0' : -1;
}
```

14.8. ***El “método Babilónico” describe un camino original para el cálculo de la raíz cuadrada. Se basa en el hecho de que el lado de un cuadrado es la raíz cuadrada de su área.***

Este método toma los dos lados (*`a` ***y `b`***) de un rectángulo imaginario de superficie*** `x` ***(***`x = a * b`***) y, de forma iterativa, modifica los tamaños de esos lados hasta lograr que sean***

iguales o casi iguales. Inicialmente se toma a = x ***y*** b = 1***; y en cada iteración destinada a convertir el rectángulo en un cuadrado, el nuevo*** a ***vale*** (a + b) / 2***, y luego el nuevo*** b ***vale*** x / a***. El proceso se repite hasta lograr que la diferencia entre*** a ***y*** b ***sea menor que un valor límite de aproximación dado.***

Suponga la siguiente función principal, que recibe del usuario un valor x ***y el valor del límite a la aproximación del cálculo:***

```
#include <stdio.h>
int main(void)
{
      double x, aprox;
      printf("Valor de x ... ");    scanf(“ %lf", &x);
      printf("Aproximacion ... "); scanf(“ %lf", &aprox);
      printf("La RC de %lf es %lf", x, RC(x, aprox));
      return 0;
}
```

Indique el prototipo de la función RC ***y escriba su posible código.***

Código 14.13. Solución al Ejercicio 14.8. propuesto.

```
// Prototipo
double RC(double, double);
// Definición
double RC(double x, double e)
{
      double a = 1 , b = x;
      while((a > b && (a - b) > e) ||
            (a < b && (b - a) > e))
      {
            a = (a + b) / 2;
            b = x / a;
      }
      return a;
}
```

CAPÍTULO 15

RECURSIVIDAD (RECURRENCIA).

"Para entender la recursividad
primero hay que entender la recursividad."

Tratamos ahora de un concepto de gran utilidad para la programación: la recursividad. La recursividad es un comportamiento presente en la misma naturaleza de muchos problemas para los que buscamos soluciones informáticas. En este capítulo queremos presentar el concepto de recursividad o de recurrencia, y mostrar luego las etapas habituales para diseñar algoritmos recursivos.

Para la redacción de este capítulo se ha empleado la siguiente referencia bibliográfica:

- **"Diseño y verificación de algoritmos. Programas recursivos."**
  Francisco Perales López
  Edita: Universitat de les Illes Balears. Palma 1998.
  Col·lecció Materials Didáctics, n. 56.
  Capítulo 3: "Diseño del algoritmos recursivos".

Resolviendo un ejercicio

Supongamos que deseamos hacer un programa sencillo que vaya solicitando del usuario valores enteros y vaya luego mostrando por pantalla esos valores (no le exigimos que realiza ninguna operación ni proceso alguno con esos valores) hasta que el usuario introduzca un entero que sea primo. En ese caso, el programa mostrará ese último entero y terminará su ejecución.

Ya conocemos cómo usar las funciones. Podemos definir una primera función que llamaremos `esPrimo`, que recibe como parámetro un entero **`unsigned long`**, y devuelve un entero corto sin signo que será cero si el entero recibido como parámetro es compuesto, y cualquier valor distinto de cero si el entero recibido como parámetro es primo.

Su prototipo será:

```
unsigned short esPrimo(unsigned long);
```

También podemos definir una segunda función, llamada `esCompuesto`, que se comporte como la anterior pero al revés: devuelve un valor verdadero si el entero recibido como parámetro es primo, y un valor distinto de cero si es compuesto.

Su prototipo será:

```
unsigned short esCompuesto(unsigned long);
```

Y la definición de esta segunda función es muy sencilla, una vez tengamos definida la primera:

```
unsigned short esCompuesto(unsigned long N)
{
      return !esPrimo(N);
}
```

Simplemente devuelve, negado, el valor devuelto por la función `esPrimo`.

La función principal de esta aplicación que acabamos de enunciar puede tener la siguiente forma:

```
int main(void)
{
	unsigned long a;
	do
	{
		printf("Entrada entero ... ");
		scanf(" %lu", &a);
		printf("Valor introducido ... %lu\n", a);
	}while(esCompuesto(a));
	return 0;
}
```

Tenemos pendiente la implementación de la función esPrimo. Será sencilla, porque además ya hemos trabajado este algoritmo (ver, por ejemplo, ejercicio 10.9). Un posible código podría ser el siguiente:

```
unsigned short esPrimo(unsigned long N)
{
	unsigned short d;
	for(d = 2; d <= sqrt(N) && N % d ; d++);
	return (unsigned short)(N % d);
}
```

Devolverá el resto del cociente entre N y el valor de la variable d con el que se haya terminado la serie de iteraciones del for. Si N es compuesto, entonces ese cociente será cero; de lo contrario devolverá el valor del resto, que será distinto de cero.

Esta función busca entre los valores consecutivos de d (entre 2 y la raíz cuadrada de N) alguno que divida a N (cuyo resto sea falso, es decir, cero) Valor tras valor, simplemente incrementa d hasta salir del rango indicado o hasta encontrar uno que divida a N. Luego, ya fuera de la iteración, devuelve el valor del resto: si la iteración ha terminado porque se ha encontrado un valor para d que verifica que N % d es cero (falso), entonces la función esPrimo devolverá el valor 0, que es el del resto del cociente entre esos dos números; si la iteración ha terminado porque no se ha encontrado ningún entero que divida a N, y se ha llegado a un valor de d mayor que la raíz cuadrada de N, entonces ese valor de d no dividirá a N y el resto será distinto de cero (verdadero).

Podemos pretender mejorar el algoritmo de la función esPrimo. Lo haríamos, quizá, si la iteración no recorriera todos los valores de d entre 2 y la raíz de N sino que tomara en consideración únicamente a los valores primos del intervalo. El código de la función esPrimo con esa nueva definición podría ser el siguiente:

```
unsigned short esPrimo(unsigned long N)
{
      unsigned short d;
      for(d = 2; d<=sqrt(N) && N%d ; d = siguientePrimo(d));
      return (unsigned short)(N % d);
}
```

Quedaría ahora pendiente definir la función siguientePrimo. Esta función recibe como parámetro un valor **unsigned long**, y devuelve otro valor **unsigned long**. Su prototipo sería el siguiente:

```
unsigned long siguientePrimo(unsigned long);
```

Y una posible definición podría ser la siguiente:

```
unsigned long siguientePrimo(unsigned long N)
{
      do N++; while(esCompuesto(N));
      return N;
}
```

Simplemente incrementa de uno en uno el valor recibido hasta alcanzar uno que no sea compuesto.

Analicemos el código desarrollado hasta el momento en este Capítulo. La función main invocará a una función llamada esPrimo, que a su vez invoca a una función llamada siguientePrimo, que a su vez invoca a la función esCompuesto, que a su vez invoca a la función esPrimo. Pero llegados a este punto nos encontramos con que la función esPrimo desencadena una secuencia de llamadas a funciones en la que ella misma está implicada. ¿Es eso válido? Si se dan algunas condiciones, la respuesta es que sí. Nuestro ejemplo ha incurrido en llamadas recursivas o recurrentes. De esto vamos a tratar en el presente capítulo. En Código 15.1. queda recogido el ejemplo que se acaba de presentar.

Código 15.1. Ejemplo de tres funciones recursivas entre sí.

```
#include <stdio.h>

unsigned short    esPrimo           (unsigned long);
unsigned short    esCompuesto       (unsigned long);
unsigned long     siguientePrimo    (unsigned long);

int main(void)
{
      unsigned long a;
      do
      {
            printf("Entrada entero ... ");
            scanf(" %lu", &a);
            printf("Valor introducido ... %lu\n", a);
      }while(esCompuesto(a));

      return 0;
}

unsigned short esCompuesto(unsigned long N)
{
      return !esPrimo(N);
}

unsigned short esPrimo(unsigned long N)
{
      unsigned short d;
      for(d = 2; d<=sqrt(N) && N%d ; d = siguientePrimo(d));

      return (unsigned short)(N % d);
}

unsigned long siguientePrimo(unsigned long N)
{
      do N++; while(esCompuesto(N));

      return N;
}
```

El concepto de recursividad.

La palabra **Recursividad** no aparecía en la 22ª Edición (año 2001) del Diccionario de la Real Academia. Sí la encontramos, en el año 2012, como "Artículo Nuevo", como avance de la vigésima tercera edición. Dice: *Cualidad de recursivo*. Y si se busca **Recursivo**, entonces tenemos la definición *Sujeto a reglas o pautas recurrentes*. De nuevo, ésta es una definición de nueva redacción para la 23ª Edición. Así que para comprender el significado de la palabra Recursividad hemos de acudir a otra palabra castellana: **Recurrencia**. El diccionario de la RAE define así esta palabra: 1. *f. Cualidad de recurrente.* 2. *f. Mat. Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes*. Y la palabra **Recurrir** ofrece, como tercera acepción: *Dicho de una cosa: Volver al lugar de donde salió.* Y la palabra **Recurrente**, 2ª acepción: *Que vuelve a ocurrir o a aparecer, especialmente después de un intervalo.*; y 4ª acepción: *Dicho de un proceso: Que se repite.*

La recursividad está presente en muchos sistemas del mundo real. Y por eso, porque muchos problemas que queremos afrontar tienen una esencia recursiva, la recursividad es una herramienta conveniente y muy útil para diseñar modelos informáticos de la realidad.

Se dice que un **sistema** es **recursivo** cuando está parcial o completamente definido en términos de sí mismo.

Quizá todos nos hemos planteado alguna vez una imagen recursiva: una fotografía que muestra a una persona que sostiene entre sus manos esa fotografía, que muestra a esa persona que sostiene entre sus manos esa fotografía, que muestra a esa persona que sostiene entre sus manos esa fotografía, que muestra…

Hay muchos problemas matemáticos que se resuelven mediante técnicas recursivas. Podemos acudir a muchos de los problemas planteados en los capítulos anteriores.

El **algoritmo de Euclides** es recurrente. El máximo común divisor de dos números m y n (mcd(m, n)) se puede definir como el máximo común divisor del segundo (de n) y del resto de dividir el primero con el segundo: mcd(m, n) = mcd(n, m mod n). La secuencia se debe parar cuando se alcanza un valor de segundo entero igual a cero: de lo contrario en la siguiente vuelta se realizaría una división por cero.

Él **cálculo del factorial**: podemos definir el factorial de un entero positivo n (n!) como el producto de n con el factorial de n – 1: n! = n * (n – 1)! Y de nuevo esta definición tiene un límite: el momento en el que se llega al valor cero: el factorial de 0 es, por definición, igual a 1.

La búsqueda de un **término de la serie de Fibonacci**: Podemos definir la serie de Fibonacci como aquella cuyo término n es igual a la suma de los términos (n – 1) y (n – 2). Y de nuevo se tiene el límite cuando n es igual a 1 ó a 2, donde el valor del término es la unidad.

Cálculo de una **fila del triángulo de Tartaglia** (cfr. Capítulo 6). Es conocida la propiedad del triángulo de Tartaglia según la cual el valor de cualquier elemento situado en el primer o último lugar de la fila es igual a uno; y el valor de cualquier otro elemento del triángulo resulta igual a la suma de los elementos de su fila anterior que están encima de la posición que deseamos calcular.

Y así tenemos que para calcular una fila del triángulo de Tartaglia no es necesario acudir al cálculo de binomios ni de factoriales, y basta con conocer la fila anterior, que se calcula si se conoce la fila anterior, que se calcula si se conoce la fila anterior… hasta llegar a las dos primeras filas, que están formadas todo por valores uno. Tenemos, por tanto, y de nuevo, un camino recurrente hecho a base de simples sumas para encontrar la solución que antes habíamos buscado mediante el cálculo de tres factoriales, dos productos y un cociente para cada elemento.

En todos los casos hemos visto un camino para definir un procedimiento recurrente que llega a la solución deseada. En los capítulos anteriores no

hemos llegado a soluciones recurrentes, sino que únicamente hemos hecho uso de estructuras de control iterativas. Ahora, en todas estas soluciones, podemos destacar **dos características básicas de todo procedimiento recurrente o recursivo**:

1. **RECURRENCIA**. Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes. La recurrencia se puede usar en aquellos sistemas donde sepamos llegar a un valor a partir de algunos valores precedentes.

2. **BASE**. Es imprescindible que, además, haya unos valores iniciales preestablecidos hacia los que se converge: unos valores que no se definen por recurrencia. En el algoritmo de Euclides se para el proceso en cuanto se llega a dos valores tales que su módulo es cero. En el algoritmo del cálculo del factorial siempre se llega a un valor mínimo establecido en el cero: `0! = 1`. En el algoritmo de Fibonacci tenemos que los dos primeros elementos de la sucesión son unos. Y para el triángulo de Tartaglia tenemos que todos los extremos de todas las filas son también iguales a uno.

   Y es necesario que la recurrencia del algoritmo converja hacia los valores iniciales base, para los que ya no es necesario invocar, recurrentemente, a la función.

La recursividad es una herramienta de diseño de algoritmos aceptada en la mayoría de los lenguajes de programación. Esos lenguajes permiten la creación de una función que hace referencia a sí misma dentro de la propia definición. Eso supone que al invocar a una función recursiva, ésta genera a su vez una o varias nuevas llamadas a ella misma, cada una de las cuales genera a su vez una o varias llamadas a la misma función, y así sucesivamente. Si la definición está bien hecha, y los parámetros de entrada son adecuados, todas las cadenas de invocaciones terminan felizmente al llegar a uno de los valores base del algoritmo. Esas llamadas finales terminan su ejecución y devuelven el control a la llamada anterior, que finaliza su ejecución y devuelve el

control a la llamada anterior, y así retornando el camino de llamadas emprendido, se llega a la llamada inicial y se termina el proceso.

Cuando un procedimiento recursivo se invoca por primera vez decimos que su **profundidad de recursión** es 1 y el procedimiento se ejecuta a nivel 1. Si la profundidad de recursión de un procedimiento es N y de nuevo se llama a sí mismo, entonces desciende un nivel de recursión y pasa al nivel N + 1. Cuando el procedimiento regresa a la instrucción de donde fue llamado asciende un nivel de recursión.

Si una función f contiene una referencia explícita a sí mismo, entonces tenemos una **recursividad directa**. Si f contiene una referencia a otra función g la cual tiene (directa o indirectamente) una referencia a f, entonces tenemos una **recursividad indirecta**.

Veamos el ejemplo de una función recursiva que calcula el factorial de un entero que recibe como parámetro de entrada:

```
Función factorial (N Entero) → Entero.
IF N = 0
THEN        factorial = 1
ELSE        factorial(N) = N * factorial(N – 1)
END IF
```

Código 15.2. Implementación de la función `factorial` recursiva.

```
long Factorial(short);        // Declaración
long Factorial(short A)       // Definición
{
      if(A == 0) return 1L;
      else return A * Factorial(A – 1);
}
```

```
long Factorial(short A)       // Otra def. + compacta
{
      return A ? (long)A * Factorial(A – 1) : 1L;
}
```

Supongamos que invocamos al procedimiento con el valor `N = 5`. La lista de llamadas que se produce queda recogida en la Tabla 15.1. Hay un total de seis llamadas al procedimiento definido para el cálculo del factorial, hasta que se le invoca con el parámetro `N = 0`, que es el único que en lugar de devolver un valor calculado con una nueva llamada, devuelve un 1. Código 15.2. recoge la implementación del algoritmo.

Nivel de Recursión	N	devuelve	recibe	resultado
1	5	5 * Factorial(4)	24	120
2	4	4 * Factorial(3)	6	24
3	3	3 * Factorial(2)	2	6
4	2	2 * Factorial(1)	1	2
5	1	1 * Factorial(0)	1	1
6	0	1	-	1

Tabla 15.1. Llamadas al procedimiento definido para el cálculo del `Factorial` para el caso de `N = 5`.

Otro ejemplo es el del cálculo de un elemento de la serie de Fibonacci. Los dos primeros valores de la serie son 1. Y la recursividad está claramente definida para cualquiera otro elemento: el término `N` es igual a la suma de los términos `(N – 1)` y `(N – 2)`.

El pseudocódigo es, por tanto, el siguiente:

```
Función Fibonacci (N Entero) → Entero.
IF N = 1 ó N = 2
THEN
      Fibonacci = 1
ELSE
      Fibonacci(N) = Fibonacci(N – 1) + Fibonacci(N – 2)
END IF
```

Y la implementación es la recogida en Código 15.3.

Código 15.3. Implementación de la función `fibonacci` recursiva.

```
long Fibonacci(short);              // Declaración
long Fibonacci(short A)             // Definición
{
      return A == 0 || A == 1 ? 1 :
            Fibonacci(A – 1) + Fibonacci(A – 2);
}
```

Árbol de recursión.

Un árbol de recursión sirve para representar las sucesivas llamadas recursivas que un programa recursivo puede generar. Es un concepto simple, pero de gran utilidad práctica para analizar el comportamiento de los algoritmos recursivos diseñados.

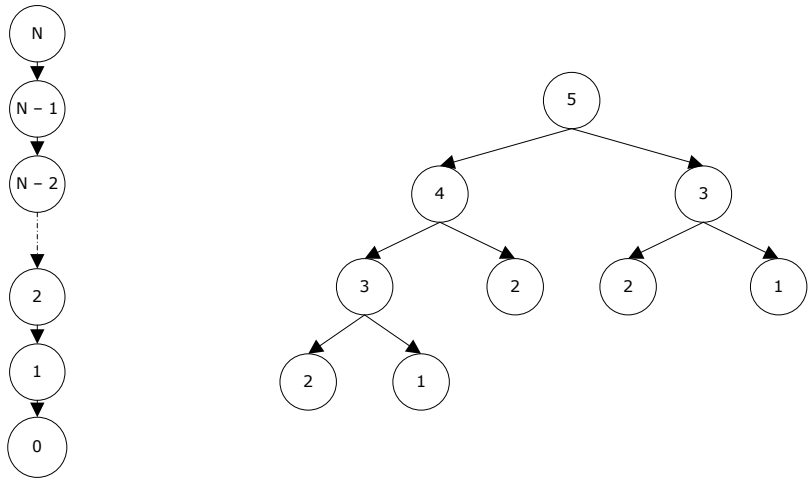


Figura 15.1. (a) Árbol de recursión de la función Factorial para `N = 5`. (b) Árbol de recursión de Fibonacci para `N = 5`.

Para construir un árbol, se representa cada llamada a la función como un nodo (que dibujaremos en forma de un círculo). Arriba se dibuja el nodo inicial con el que es invocada por primera vez la función recurrente. Más allá (hacia abajo) se van insertando las llamadas que recibe la función recursivamente. En cada nodo del árbol (que se corresponde con una llamada) se le adjunta una etiqueta con el valor o valores de los parámetros de entrada.

Por ejemplo, el árbol de recursión de la función `Factorial` tiene el aspecto como el recogido en la Figura 15.1.(a). Para `fibonacci`, el árbol queda recogido en la Figura 15.1.(b). Hemos tomado N = 5.

Recursión e iteración.

El concepto de iteración ya quedó visto en el capítulo 5, al hablar de los algoritmos y de sus modos de representación. Y al inicio de éste capítulo hemos visto que todos los algoritmos que en el capítulo 5 resolvimos mediante la iteración, también pueden ser solventados mediante la recursividad o recurrencia.

La cuestión a plantear es entonces: ¿qué es mejor: recursividad, o estructuras iterativas? Para responder a esta cuestión, un factor importante que cabe plantearse es cuál de las dos formas requiere menor espacio de almacenamiento en memoria. Y otro, cuál de las dos formas resuelve el problema con menor tiempo.

Veamos el algoritmo del cálculo del factorial, expresado como función recurrente (**Función** `FactorialR`) y expresado como función con estructura iterativa (**Función** `FactorialI`)

```
Función FactorialR (N Entero) → Entero
Acciones:
IF N = 0
THEN        FactorialR(N) = 1
ELSE        FactorialR(N) = N * FactorialR(N – 1)
END IF
```

```
Función FactorialI (N Entero) → Entero
Variables
      aux , i, Enteros
Acciones:
IF N = 0
THEN
      FactorialI(N) = 1
ELSE
      aux = 1
      i = 1
      WHILE i menor o igual que N
            aux = aux * i
            i = i + 1
      END WHILE
      FactorialI(N) = aux
END IF
```

Aparentemente, parece que la función recursiva exige menos memoria, pues de hecho no requiere el uso de ninguna variable local propia de la función mientras que la función iterativa requiere de dos variables.

Pero en realidad el programa recursivo va almacenando en una pila de memoria los N números: `N, N – 1, N – 2, ..., 2, 1`, que son los parámetros de llamada antes de cada recursión. A medida que vaya saliendo de las sucesivas llamadas, multiplicará esos números en el mismo orden en que lo hace el programa iterativo. Así pues, el programa recurrente requiere de mayor almacenamiento. Además, también tardará más tiempo en ejecutarse, porque debe almacenar en memoria y recuperar luego de ella todos los números, realizar todas las multiplicaciones, y realizar las distintas llamadas a la función.

Veámoslo sobre la propia función en C (cfr. Código 15.2.). Cada vez que la función es invocada por sí misma, se crea de nuevo una variable `A`, distinta de la variable `A` creada en la anterior invocación. Para cada llamada creamos un juego de variables cuyo ámbito es el de esta llamada, y su vida el tiempo que se tarde en ejecutar la última sentencia (la sentencia **`return`**) del bloque de la función.

Supongamos que queremos conocer el valor del factorial de 3. Invocamos a la función `Factorial` con ese valor como argumento.

`printf("El factorial de %hd es %ld.\n",3, Factorial(3));`

Primera llamada: se crea la variable <A, short, R_1, 3>. Como A es distinto de cero, no se devuelve el entero 1, sino el producto de A por el Factorial de (A – 1). Entonces, antes de terminar la ejecución de la función Factorial y eliminar la variable A localizada en R_1 necesitamos recibir el valor de Factorial de (A – 1).

Segunda llamada: se crea la variable <A, short, R_2, 2>. Con el mismo nombre que en la llamada anterior, son variables diferentes, ubicadas en posiciones de memoria diferentes. En este momento, la variable en ámbito es la ubicada en R_2, la ubicada en R_1 no es accesible: siempre que en esta segunda ejecución de la función Factorial hagamos referencia a la variable A, se entiende que nos referimos a la ubicada en R_2. Como esta variable A no vale 0, entonces la función devuelve el valor del producto de A (la de R_2) por Factorial(A – 1). Y de nuevo, antes de terminar la ejecución de la función Factorial y eliminar la variable A localizada en R_2 necesitamos recibir el valor de Factorial(A – 1).

Tercera llamada: se crea la variable <A, short, R_3, 1>. Con el mismo nombre que en las dos llamadas anteriores, son variables diferentes, ubicadas en posiciones de memoria diferentes. En este momento, la variable en ámbito es la ubicada en R_3, las ubicadas en R_1 y R_2 no son accesibles: siempre que en esta tercera ejecución de la función Factorial hagamos referencia a la variable A, se entiende que nos referimos a la ubicada en R_3. Como esta variable A no vale 0, entonces la función devuelve el valor del producto de A (la de R_3) por Factorial(A – 1). Y de nuevo, antes de terminar la ejecución de la función Factorial y eliminar la variable A localizada en R_3 necesitamos recibir el valor de Factorial(A – 1).

Cuarta llamada: se crea la variable <A, short, R_4, 0>. Con el mismo nombre que en las tres llamadas anteriores, son variables diferentes, ubicadas en posiciones de memoria diferentes. En este momento, la variable en ámbito es la ubicada en R_4; las ubicadas en R_1, R_2, y R_3 no

son accesibles: siempre que en esta cuarta ejecución de la función `Factorial` hagamos referencia a la variable `A`, se entiende que nos referimos a la ubicada en R_4. El valor de esta variable es 0 por lo que la función devuelve el valor 1 y termina su ejecución. La variable `A` ubicada en R_4 termina su existencia y el control del programa vuelve a quien llamó a la función.

Quien llamó a la función fue la propia función `Factorial`, en su tercera llamada. Estaba pendiente, para cerrarse y devolver un valor, a recibir un valor de la función `Factorial`. Y ha recibido el valor 1, que multiplica al valor de `A` que también es 1, y devuelve a quien la llamó. La variable `A` ubicada en R_3 termina su existencia y el control del programa vuelve a quien llamó a la función.

Y quien llamó a la función fue la propia función `Factorial`, en su segunda llamada. Estaba pendiente, para cerrarse y devolver un valor, a recibir un valor de la función `Factorial`. Y ha recibido el valor 1, que multiplica al valor de `A` que es 2, y devuelve a quien la llamó. La variable `A` ubicada en R_2 termina su existencia y el control del programa vuelve a quien llamó a la función.

Y quien llamó a la función fue la propia función `Factorial`, en su primera llamada. Estaba pendiente, para cerrarse y devolver un valor, a recibir un valor de la función `Factorial`. Y ha recibido el valor 2, que multiplica al valor de `A` que es 3, y devuelve a quien la llamó. La variable `A` ubicada en R_1 termina su existencia y el control del programa vuelve a quien llamó a la función.

Y quien llamó a la función `Factorial` fue la función principal, que vuelve a recuperar el control de ejecución del programa y que recibe el valor devuelto por la función que se lo pasa como parámetro a la función `printf` para que muestre ese valor por pantalla:

```
El factorial de 3 es 6.
```

4 ejecuciones, 4 ámbitos, 4 variables distintas, 4 vidas distintas.

Otro ejemplo para estudiar el comportamiento de una función definida de forma recursiva o de forma iterativa lo tenemos con los números de Fibonacci. Mostramos de nuevo la definición recursiva (**Función** FibonacciR) y la definición iterativa (**Función** FibonacciI).

```
Función FibonacciR (N Entero) → Entero
Acciones:
IF N menor o igual que 2
THEN
      FibonacciR(N) = 1
ELSE
      FibonacciR(N) = FibonacciR(N – 1) + FibonacciR(N – 2)
END IF
```

```
Función FibonacciI (N Entero) → Entero
Variables
      a, b, c, i, Enteros.
Acciones:
IF N menor o igual que 2
THEN
      FibonacciI(N) = 1
ELSE
      a = 1
      b = 1
      i = 2
      WHILE N menor o igual que N
            c = a + b
            a = b
            b = c
      END WHILE
      FibonacciI(N) = c
END IF
```

El árbol de recursividad para la función FibonacciR, para N = 5 queda en la Figura 15.2.(b). Y así observamos que para el cálculo del valor de FibonacciR(5) hemos calculado una vez el valor de FibonacciR(4), dos veces el valor de FibonacciR(3), tres veces el de FibonacciR(2), y dos veces el de FibonacciR(1). Así pues, con el árbol de recursión vemos que el programa repite los mismos cálculos una y otra vez. Y el tiempo que tarda la función recursiva en calcular FibonacciR(N) aumenta de forma exponencial a medida que aumenta N.

En cambio, la función iterativa usa un tiempo de orden lineal (aumenta linealmente con el incremento del valor de N), de forma que la diferencia de tiempos entre una y otra forma de implementación es notable.

La recursión siempre puede reemplazarse con la iteración y las pilas de almacenamiento de datos. La pregunta es, entonces: ¿Cuándo usamos recursión, y cuándo usamos iteración?

Si el árbol de recursión tiene muchas ramas, con poca duplicación de tareas, entonces el método más adecuado es la recursión, siempre que además el programa resultante sea más claro, sencillo y fácil de obtener.

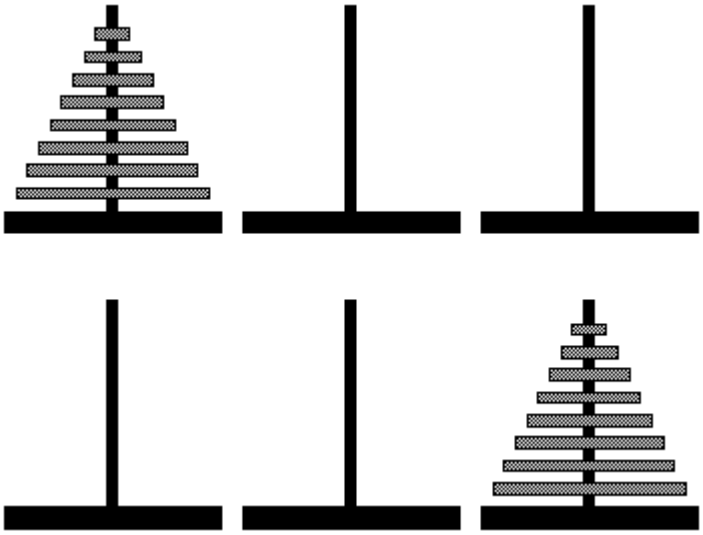
Los algoritmos recursivos son muy apropiados cuando el problema a resolver, la función a calcular o la estructura de datos a procesar están definidos en términos recursivos. La recursión debe tenerse en cuenta como una técnica que nos permite diseñar algoritmos de una forma sencilla, clara y elegante. No se debe utilizar por tanto cuando dificulte la comprensión el algoritmo.

Ejercicio: las torres de Hanoi.

Dice la leyenda que, al crear el mundo, Dios situó sobre la Tierra tres varillas de diamante y sesenta y cuatro discos de oro. Los discos eran todos de diferente tamaño, e inicialmente fueron colocados en orden decreciente de diámetros sobre la primera de las varillas. También creó Dios un monasterio cuyos monjes tenían la tarea de trasladar todos los discos desde la primera varilla a la tercera. Los monjes sólo podían mover, de una varilla a otra, un disco cada vez; y tenían una única limitación: ningún disco podía colocarse sobre otro de diámetro menor. La leyenda decía que cuando los monjes terminasen su tarea, el mundo llegaría a su fin.

La leyenda no es cierta: es fruto de la imaginación de un matemático del siglo XVIII, que inventó el juego de las Torres de Hanoi (así se llama) y

que diseñó así su propia campaña de publicidad y marketing que le permitiera lanzar su invento al mercado. La invención resultó, sin embargo, efectiva: ha perdurado hasta nuestros días, el juego es conocido en todo el mundo,… y nos ha dejado una pregunta: si la leyenda fuera cierta… ¿cuándo sería el fin del mundo?



El mínimo número de movimientos que se necesitan para resolver este problema es de $2^{64}-1$. Si los monjes hicieran un movimiento por segundo, y no parasen en ningún instante hasta completar la total ejecución de la tarea de traslado, los discos estarían en la tercera varilla en poco menos de 585 mil millones de años. Teniendo en cuenta que la tierra tiene sólo unos cinco mil millones, o que el universo está entre quince y veinte mil millones de años, está claro que o los monjes se dan un poco más de prisa, o tenemos universo y mundo para rato.

Se puede generalizar el problema de Hanoi variando el número de discos: desde un mínimo de 3 hasta el máximo que se quiera.

El problema de Hanoi es curioso, y su formulación elegante. Y su solución es muy rápida de calcular. La pega no está en la complicación del procedimiento (que no es complicado), sino en que a medida que aumenta el número de discos crece exponencialmente el número de pasos.

Existe una versión recursiva eficiente para hallar el problema de las torres de Hanoi para pocos discos; pero el coste de tiempo y de memoria se incrementa excesivamente al aumentar el número de discos.

La solución recursiva es llamativamente sencilla de implementar y de construir. Y de una notable belleza, creo yo.

Si numeramos los discos desde 1 hasta N, y si llamamos X al poste donde quedan inicialmente colocado los discos, Z al poste de destino de los discos, e Y al tercer poste intermedio, el algoritmo recurrente que resuelve el problema de las torres de Hanoi quedaría de la siguiente manera:

```
IF N = 1
THEN
        Trasladar disco de X a Z.
ELSE
1.      Trasladar discos desde 1 hasta N – 1 de X a Y,
        tomando como auxiliar a Z.
2.      Trasladar disco N de X a Z.
3.      Trasladar discos desde 1 hasta N – 1 de Y a Z,
        tomando como auxiliar a X.
END IF
```

Podemos definir una función que muestre por pantalla los movimientos necesarios para hacer el traslado de una torre a otra. A esa función la llamaremos `Hanoi`, y recibe como parámetros los siguientes valores:

a) una variable entera que indica el disco más pequeño de la pila que se quiere trasladar.

b) una variable entera que indica el disco más grande de la pila que se desea trasladar. (Hay que tener en cuenta que con el algoritmo que hemos definido siempre se trasladan discos de tamaños consecutivos.)

c) Una variable que indique la torre donde están los discos que trasladamos.

d) Una variable que indica la torre a donde se dirigen los discos.

Un modo de identificar las torres y de tener en todo momento identificado la torre auxiliar con variables enteras es el siguiente: Llamaremos a la torre origen torre `i` y a la torre destino torre `j`. Y exigimos a las dos variables `i`, `j`, dos restricciones: (1) toman sus valores entre los enteros 1, 2 y 3; y (2) no pueden ser nunca iguales.

Y entonces tendremos que la torre auxiliar vendrá siempre definida con la expresión `6 – i – j`, como puede verse en la Tabla 15.2.

i	j	6 – i – j
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Tabla 15.2. Valores de las torres de Hanoi.

La función `Hanoi` queda entonces de la siguiente forma:

```
Función Hanoi(D1, D2, i, j, Enteros)
Acciones:
IF D1 = D2
THEN
      Mover disco D1 de i a j
ELSE
      Hanoi(D1, D2 – 1, i, 6 – i – j)
      Hanoi(D1, D2, i, j)
      Hanoi(D1, D2 – 1, 6 – i – j, j)
END IF
```

La forma que adquiere esta función implementada en C es tan sencilla como lo que se muestra en Código 15.4.

Código 15.4. Implementación de la función Hanoi recursiva.

```
typedef unsigned short int usi;
void Hanoi(usi D1, usi D2, usi i, usi j)
{
      if(D1 == D2)
            printf("Disco %hu de %hu a %hu\n",D1, i, j);
      else
      {
            Hanoi(D1, D2 - 1, i, 6 - i - j);
            Hanoi(D2, D2, i, j);
            Hanoi(D1, D2 - 1, 6 - i - j, j);
      }
}
```

Y si se invoca esta función inicialmente con la sentencia

`Hanoi(1, discos, 1, 3);`

donde discos es el número de discos que tiene la torre de Hanoi, la ejecución de esta función muestra todos los movimientos que debe hacerse con los anillos para pasarlos desde la torre 1 a la torre 3.

Por ejemplo, si tomamos el valor `discos = 4` tendremos la siguiente salida:

```
Disco 1 de 1 a 2              Disco 1 de 2 a 3
Disco 2 de 1 a 3              Disco 2 de 2 a 1
Disco 1 de 2 a 3              Disco 1 de 3 a 1
Disco 3 de 1 a 2              Disco 3 de 2 a 3
Disco 1 de 3 a 1              Disco 1 de 1 a 2
Disco 2 de 3 a 2              Disco 2 de 1 a 3
Disco 1 de 1 a 2              Disco 1 de 2 a 3
Disco 4 de 1 a 3
```

Para terminar esta presentación de las torres de Hanoi queda pendiente una cosa: demostrar que, efectivamente, el número de movimientos para desplazar todos los discos de una torre a otra con las condiciones impuestas en el juego es igual a dos elevado al número de discos, menos uno.

Eso se demuestra fácilmente por inducción (como no):

Base: Se verifica para $N = 1$ (mover un disco): $M(1) = 2^{N-1} - 1 = 1$: efectivamente, el mínimo número de movimientos es 1.

Recurrencia: Supongamos que se cumple para N: $M(N) = 2^N - 1$. Entonces si tenemos $N + 1$ discos, lo que hacemos es desplazar dos veces los N primeros discos y entre medias una vez el disco $N + 1$.

Es decir, tenemos

$M(N + 1) = 2 \cdot M(N) + 1 = 2 \cdot \left(2^N - 1\right) + 1 = 2^{N+1} - 1$ (c.q.d.)

Ejercicio: la función de Ackermann.

La de Ackermann es una función recursiva, definida en el dominio de los naturales. Tiene como argumento dos números naturales y como valor de retorno otro valor natural. Su definición general es:

$$A(m, n) \begin{cases} n + 1, & \text{si } m = 0 \\ A(m - 1, 1), & \text{si } m > 0 \text{ y } n = 0 \\ A(m - 1, A(m, n - 1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

La implementación en C de esta función es trivial. Puede verla en Código 15.5. La función main para invocar a nuestra función de Ackermann es también muy sencilla. Ejecute el programa sugerido en Código 15.5. En la Tabla 15.3. se recogen algunos valores que usted podrá obtener.

Todo parece sencillo. Y, de hecho, lo es: hemos logrado implementar la función en tres líneas de código. Pero si intenta calcular valores distintos a los mostrados en la Tabla 15.3. quizá se lleve una sorpresa. Porque al final tendrá que el programa se aborta y no logra terminar el cálculo que tan fácilmente hemos definido.

En Código 15.5.bis dispone de una implementación de la función de Ackermann con la que podrá ver las llamadas recursivas que realiza y el número de bytes que necesita para su ejecución: 2 nuevas variables de tipo **`unsigned long int`** en cada nueva iteración.

Código 15.5. Implementación de la función recursiva Ackermann.

```
// Declaración
unsigned long Ackermann(unsigned long m, unsigned long n);
// Función main
int main(void)
{
      long m, n;
      printf("Valor de m ... ");    scanf(" %lu", &m);
      printf("Valor de n ... ");    scanf(" %lu", &n);

      printf("\nAckermann(%lu, %lu) = %lu\n\n",
                        m, n, Ackermann(m, n));
            return 0;
}
// Definición
unsigned long Ackermann(unsigned long m, unsigned long n)
{
      if(!m)      return n + 1;
      if(!n)      return Ackermann(m – 1 , 1);
      return Ackermann(m – 1 , Ackermann(m , n – 1));
}
```

A(m , n)	n = 0	n = 1	n = 2	n = 3	n = 4	n = 5
m = 0	1	2	3	4	5	6
m = 1	2	3	4	5	6	7
m = 2	3	5	7	9	11	13
m = 3	5	13	29	61	125	253
m = 4	13	¿?	¿?	¿?	¿?	¿?

Tabla 15.3. Primeros valores de la función de Ackermann.

Ackermann definió su curiosa función en 1928. Logró con ella demostrar una cuestión estrictamente matemática; algo que escapa por completo al objetivo de este manual. Pero nos ofrece la posibilidad de obtener nuestra particular conclusión: el recurso a la recursividad ofrece

soluciones elegantes y, con frecuencia, eficaces; pero también con frecuencia se llega con ella a soluciones (¿soluciones?) que requieren excesivos recursos de computación y que no ofrecen resultado. En el caso de la función de Ackermann el problema no se resuelve buscando una definición no recursiva. No se logrará fácilmente un programa que llegue a un resultado en un tiempo de computación razonable. Pero… ¿sabría implementar la función de Ackermann de forma no recursiva?

Código 15.5.bis. Implementación de la función recursiva Ackermann.

```
unsigned long Ackermann(unsigned long m, unsigned long n)
{
      static long int llamadas = 0;
      static long int memoria = 0;

      memoria += 2 * sizeof(unsigned long);
      printf("%8ld llamadas.\t", ++llamadas);
      printf("%8ld bytes requeridos\r", memoria);

      if(!m)      return n + 1;
      if(!n)      return Ackermann(m – 1 , 1);
      return Ackermann(m – 1 , Ackermann(m , n – 1));
}
```

Recapitulación.

Hemos visto el modelo recursivo para plantear y solventar problemas. Esta forma de trabajo es especialmente útil cuando pretendemos solventar cuestiones basadas en propiedades definidas sobre el conjunto de enteros o un conjunto numerable donde se pueda establecer una correspondencia entre los enteros y los elementos de ese conjunto.

Hemos visto el modo en que se definen por recurrencia una serie de conjuntos, cómo se demuestran propiedades para estos conjuntos basándonos en la demostración por inducción, y cómo se llega así a

replantear muchos problemas, alcanzando una formulación muy elegante y sencilla: la forma recursiva.

La recursividad ofrece habitualmente algoritmos muy sencillos de implementar y de interpretar. La principal limitación que presenta esta forma de resolver problemas es que habitualmente supone un elevado coste de tiempo de computación y de capacidad de almacenamiento de memoria.

Hemos descrito el camino a seguir para alcanzar una solución recurrente ante un problema que nos planteamos. Los ejercicios que planteamos a continuación pueden servir para seguir profundizando en ese itinerario por el que se llega a las definiciones recurrentes.

Ejercicios.

15.1. Definir la operación potencia de forma recursiva.

Simplemente, basta considerar que $a^n = a \times a^{n-1}$, y que $a^0 = 1$. Con estas dos expresiones tenemos ya una base y una recurrencia. El código de la función recurrente tendrá un aspecto similar al de la función factorial. Puede verlo en Código 15.6. La función devuelve el producto de la base por la potencia disminuido en uno el exponente si el exponente es distinto de 0; el valor 1 si el exponente es 0.

Código 15.6. Implementación de la función `potencia` recursiva.

```
#include <stdio.h>

long potencia(short, short);
```

Código 15.6. (Cont.).

```
int main(void)
{
      short b, e;

      printf("Valor de la base:    ");   scanf(“ %hd", &b);
      printf("Valor del exponente: ");   scanf(“ %hd", &e);
      printf("%hd elevado a %hd es: ", b, e);
      printf("%ld\n”, potencia(b, e));
      return 0;
}

long potencia(short b, short e)
{
      return e ? b * potencia(b, e - 1) : (long)1;
}
```

15.2. ***Definir la operación suma de enteros de forma recursiva. Téngase en cuenta, por ejemplo, que si pretendemos sumar dos enteros a y b, podemos tomar: suma(a,b) = a si b es igual a cero; y suma(a,b) es 1 + suma(a,b - 1) en otro caso.***

Ya tenemos, en el mismo enunciado, la definición de la función. La solución es muy simple: está recogida en Código 15.7.

Código 15.7. Implementación de la función suma recursiva.

```
long suma(short a, short b)
{
      return b ? 1L + suma(a , b - 1) : (long)a;
}
```

Evidentemente, para que el programa esté completo faltaría la declaración de la función y la definición de la función principal. No es necesario incluir este código aquí, en la resolución de todos estos ejercicios. Sí deberá escribirlo usted si desea verificar que estas funciones realizan la operación que exigía el enunciado.

15.3. ***Definir la operación producto de forma recursiva. Téngase en cuenta, por ejemplo, que si pretendemos multiplicar dos enteros a y b, podemos tomar:*** `producto(a, b) = 0` ***si*** `b` ***es igual a cero; y*** `producto(a, b)` ***es*** `a + producto(a, b – 1)` ***en otro caso.***

Un posible código de la función podría ser el recogido en Código 15.8.

Código 15.8. Implementación de la función `producto` recursiva.

```
long producto(short a, short b)
{
      return b ? a + producto(a, b - 1) : 0;
}
```

15.4. ***Defina una función que muestre, mediante un proceso recurrente, el código binario de un entero dado.***

Para encontrar este proceso es sencillo buscar los casos más sencillos, que forman la **Base** de la recurrencia: El código binario de $(0)_{10}$ es 0; el código binario de $(1)_{10}$ es 1.

Código 15.9. Implementación de la función `Binario` recursiva.

```
void Binario(unsigned long n)
{
      if(n <= 1) printf("%ld", n);
      else
      {
            Binario(n/2);
            printf("%ld", n % 2);
      }
}
```

Para el proceso de recurrencia baste con decir que si el número es mayor que 1, entonces ya necesita de al menos dos dígitos binarios; y si el número es mayor que 3 entonces requiere ya de al menos tres dígitos. Y, en general, si el número es mayor que $2^n - 1$ entonces el número requiere, para su codificación binaria, de al menos n dígitos.

Y, por lo indicado en los capítulos sobre codificación numérica y sobre codificación interna de la información, sabemos que podemos concatenar dígitos a fuerza de dividir por dos sucesivas veces hasta llegar al valor 0 ó 1.

Podemos entonces definir el siguiente algoritmo recursivo para obtener el código binario de un entero:

Función `Binario` (`n > 0` Entero en base 10) → Muestra Código binario
Acciones:
IF `n = 1`
THEN
    Mostrar `n`
ELSE
    `Binario(n / 2)`
    Mostrar `n mod 2`
END IF

Cuyo código en C podría ser el propuesto en Código 15.9.

Supongamos que invocamos a la función `Binario` con el valor para el parámetro n igual a 12.

Llamada `Binario(12)`. Profundidad de Recursión: 1. Como n = 12 no es 1, … se ejecuta `Binario(6)`.

Llamada `Binario(6)`. Profundidad de Recursión: 2. Como n = 6 no es 1, … se ejecuta `Binario(3)`.

Llamada `Binario(3)`. Profundidad de Recursión: 3. Como n = 3 no es 1, … se ejecuta `Binario(1)`.

Llamada `Binario(1)`. Profundidad de Recursión: 4. Como n = 1 simplemente imprimimos ese valor **1** y salimos de la ejecución. Pasamos a la Profundidad de Recursión 3. La segunda sentencia, después de la llamada `Binario(1)` que ya ha finalizado es la de imprimir el resto de dividir N = 3 por 2: **1**. Se termina así la ejecución de `Binario(3)`. Pasamos a la Profundidad de Recursión 2. La segunda sentencia, después de la llamada `Binario(3)` que ya ha finalizado es la de imprimir el resto de dividir N = 6 por 2: **0**. Se termina así la ejecución de `Binario(6)`. Pasamos a la Profundidad de Recursión 1. La segunda sentencia, después de la llamada `Binario(6)` que ya ha finalizado es la de imprimir el resto de dividir N = 12 por 2: **0**. Se termina así la ejecución de `Binario(12)`. Vuelve el control a la función que invoco inicialmente la función `Binario` con el valor 12.

Han quedado impreso, y por ese orden, los dígitos 1, 1, 0 y 0, que son, efectivamente, el código binario del entero 12 (1100).

15.5. ***Suponga la función cuyo prototipo es:*** `void f(char*, int);` ***y su definición es:***

```
void f(char *cad, int i)
{
      if( cad[i] != '\0' )
      {
            f(cad , i + 1);
            printf("%c", cad[i]);
      }
}
```

Suponga que la invocamos con los siguientes valores de entrada:

```
f("EXAMEN", 0);
```

Indique entonces qué salida ofrece esta función por pantalla.

La función `f` recibe una cadena de texto y el índice de uno de sus elementos. Mientras que el valor del elemento de la cadena en ese índice no sea el carácter nulo, la función `f` volverá a invocarse a sí misma, incrementando en uno el índice.

Cuando llegue al carácter nulo no hace más llamadas. En ese momento, se ejecuta la segunda línea de cada una de las llamadas a la función (la línea con el `printf`), y se van imprimiendo todos los caracteres en el orden inverso en que se han pasado sus índices en las sucesivas llamadas.

Por ejemplo, supongamos que invocamos a la función `f` con una cadena, declarada como **`char`** `a[] = "hola";` y el valor de índice 0: `f(a, 0);`

Si hacemos el árbol de recursión, obtenemos el recogido en la Figura 15.2. En ella ha quedado indicado el valor del índice `i` y el correspondiente valor en esa posición de la cadena de caracteres.

Cuando llega a la llamada con el valor de `i = 4`, al encontrarse con que a[i] es, efectivamente, el carácter nulo, no realiza una nueva llamada a sí misma ni ejecuta la función `printf`. Entonces queda ejecutar la segunda sentencia recogida en la estructura condicional de la llamada anterior, con `i = 3`: imprime por tanto el carácter `'a'` y se termina la ejecución de esta llamada a la función, devolviendo el control a la anterior llamada: aquella que se hizo con el valor de `i = 2`, y entonces imprime el carácter `'l'` y se termina de nuevo la ejecución de esta llamada a la función, devolviendo el control a la anterior llamada: aquella que se hizo con el valor de `i = 1`, y entonces imprime el

carácter `'o'` y se termina de nuevo la ejecución de esta llamada a la función, devolviendo el control a la anterior llamada: aquella que se hizo con el valor de `i = 0`, y entonces imprime el carácter `'h'` y se termina de nuevo la ejecución de esta llamada a la función, devolviendo el control a la función principal o a aquella que invocó originariamente a la función `f`.

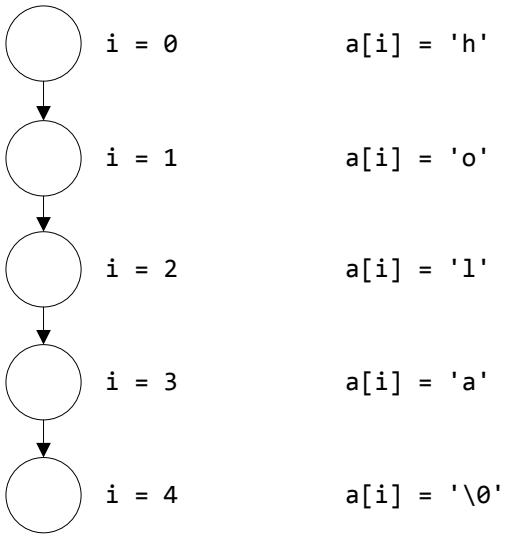


Figura 15.2. Árbol de recursión para el código del ejercicio 15.5.

La salida es, por lo tanto, si la entrada es "EXAMEN": "NEMAXE": la cadena impresa en orden inverso.

15.6. ***Defina una función que devuelva el mayor de una lista de números. El algoritmo no puede comparar en ningún momento más de dos números para decidir el mayor de entre todos los de la lista.***

Vamos a definir una función, que llamaremos M y que recibe como parámetros una lista de enteros y un valor entero que indica la cantidad de elementos de esa lista. La función devuelve un entero: el mayor de entre todos los valores de la lista.

La limitación impuesta en el enunciado nos indica la **base** de nuestro algoritmo de recurrencia: Si la lista tiene un solo elemento, ese elemento es el que devuelve la función; si la lista tiene dos elementos, la función devuelve el mayor de ellos. Si la lista tiene más de dos elementos, entonces la función no puede contestar a la cuestión, porque en ningún momento podemos comparar más de dos elementos.

Supongamos que tenemos n elementos en la lista. Entonces está claro que el mayor de entre ellos es igual al mayor entre el mayor de los n / 2 primeros y el mayor entre los n / 2 segundos. Esta es la base para definir nuestro proceso de **recurrencia**:

Podemos entonces plantear el algoritmo de la siguiente manera:

```
Función M(x_1, x_2, ..., x_n Enteros, n Entero) → Entero
Variables
      m, x, e y, Enteros
Acciones:
IF n = 1
THEN
      M(n) = n
ELSE
      m = n / 2
      x = M(x_1, ..., x_m, m)
      y = M(x_m+1, ..., x_n, n – m)
      IF x > y
      THEN
            M(n) = x
      ELSE
            M(n) = y
      END IF
END IF
```

La función, implementada en C, queda recogida en Código 15.10.

Código 15.10. Implementación de la función `M` recursiva.

```
short M(short *v, short n)
{
      short m;
      short x, y;

      if(n == 1) return *v;
      m = n / 2;
      x = M(v , m);
      y = M(v + m , n - m);
      return x > y ? x: y;
}
```

Y la llamada a esta función o al método es de la forma `M(v, dim);` donde v es una variable de tipo vector o array de enteros y donde `dim` es el número de elementos del vector v.

CAPÍTULO 16

PUNTEROS.

La memoria puede considerarse como una enorme cantidad de posiciones de almacenamiento de información perfectamente ordenadas. Cada posición es un octeto o byte de información. Cada posición viene identificada de forma inequívoca por un número que suele llamarse dirección de memoria. Cada posición de memoria tiene una dirección única. Los datos de nuestros programas se guardan en esa memoria.

La forma en que se guardan los datos en la memoria es mediante el uso de variables. Una variable es un espacio de memoria reservado para almacenar un valor: valor que pertenece a un rango de valores posibles. Esos valores posibles los determina el tipo de dato de esa variable. Dependiendo del tipo de dato, una variable ocupará más o menos bytes de la memoria, y codificará la información de una u otra manera.

Si, por ejemplo, creamos una variable de tipo **`float`**, estaremos reservando cuatro bytes de memoria para almacenar sus posibles valores. Si, por ejemplo, el primero de esos bytes es el de posición

ABD0:FF31 (es un modo de escribir: en definitiva estamos dando 32 bits para codificar las direcciones de la memoria), el segundo byte será el ABD0:FF32, y luego el ABD0:FF33 y finalmente el ABD0:FF34. La dirección de memoria de esta variable es la del primero de sus bytes; en este caso, diremos que toda la variable **float** está almacenada en ABD0:FF31. Ya se entiende que al hablar de variables **float**, se emplean un total de 4 bytes.

Ese es el concepto habitual cuando se habla de la posición de memoria o de la dirección de una variable.

Además de los tipos de dato primitivos ya vistos y usados ampliamente en los capítulos anteriores, existe un C un tipo de dato especial, que ofrece muchas posibilidades y confiere al lenguaje C de una filosofía propia. Es el tipo de dato puntero. Mucho tiene que ver ese tipo de dato con la memoria de las variables. Este capítulo está dedicado a su presentación.

Definición y declaración.

Una variable tipo puntero es una variable que contiene la dirección de otra variable.

Para cada tipo de dato, primitivo o creado por el programador, permite la creación de variables puntero hacia variables de ese tipo de dato. Existen punteros a **char**, a **long**, a **double**, etc. Son nuevos tipos de dato: *puntero a* **char**, *puntero a* **long**, *puntero a* **double**,…

Y como tipos de dato que son, habrá que definir para ellos un dominio y unos operadores.

Para declarar una variable de tipo puntero, la sintaxis es similar a la empleada para la creación de las otras variables, pero precediendo al nombre de la variable del carácter asterisco (*).

```
tipo *nombre_puntero;
```

Por ejemplo:

```
short int *p;
```

Esa variable p así declarada será una variable *puntero a* **short**, que no es lo mismo que *puntero a* **float**, etc.

En una misma instrucción, separados por comas, pueden declararse variables puntero con otras que no lo sean:

```
long a, b, *c;
```

Se han declarado dos variables de tipo **long** y una tercera que es *puntero a* **long**.

Dominio y operadores para los punteros.

El dominio de una variable puntero es el de las direcciones de memoria. En un PC las direcciones de memoria se codifican con 32 bits (4 bytes), y toman valores desde `0000:0000` hasta `FFFF:FFFF`, (expresados en base hexadecimal).

El operador **sizeof**, aplicado a cualquier variable de tipo puntero, devuelve el valor 4.

Una observación importante: si un PC tiene direcciones de 32 bytes… ¿cuánta memoria puede llegar a direccionar?: Pues con 32 bits es posible codificar hasta 2^{32} bytes, es decir, hasta 4×2^{30} bytes, es decir 4 Giga bytes de memoria.

Pero sigamos con los punteros. Ya tenemos el dominio. Codificará, en un formato similar al de los enteros largos sin signo, las direcciones de toda nuestra memoria. Ese será su dominio de valores.

Los operadores son los siguientes:

Operador dirección (`&`): Este operador se aplica a cualquier variable, y devuelve la dirección de memoria de esa variable. Por ejemplo, se puede escribir:

```
long x, *px = &x;
```

Y así se ha creado una variable *puntero a* **long** llamada px, que servirá para almacenar direcciones de variables de tipo **long**. Mediante la segunda instrucción asignamos a ese puntero la dirección de la variable x. Habitualmente se dice que px **apunta a** x.

El operador dirección no es propio de los punteros, sino de todas las variables. Pero no hemos querido presentarlo hasta el momento en que por ser necesario creemos que también ha de ser fácilmente comprendido. De hecho este operador ya lo usábamos, por ejemplo, en la función `scanf`, cuando se le indica a esa función "dónde" queremos que almacene el dato que introducirá el usuario por teclado: por eso, en esa función precedíamos el nombre de la variable cuyo valor se iba a recibir por teclado con el operador &.

Hay una excepción en el uso de este operador: no puede aplicarse sobre una variable que haya sido declarada **register** (cfr. Capítulo 12). El motivo es claro: al ser una variable **register**, le hemos indicado al compilador que no almacene su información en la memoria sino en un registro de la ALU. Y si la variable no está en memoria, no tiene sentido que le solicitemos la dirección de donde no está.

Operador indirección (*): Este operador sólo se aplica a los punteros. Al aplicar a un puntero el operador indirección, se obtiene el contenido de la posición de memoria "apuntada" por el puntero. Supongamos:

```
float pi = 3.14, *pt;
pt = &pi;
```

Con la primera instrucción, se han creado dos variables:

<pi, **float**, R_1, 3.14> y <pt, **float***, R_2, ¿? >

Con la segunda instrucción damos valor a la variable puntero:

<pt, **float***, R_2, R_1>

Ahora la variable puntero pt vale R_1, que es la dirección de memoria de la variable pi.

Hablando ahora de la variable pt, podemos hacernos tres preguntas, todas ellas referidas a direcciones de memoria.

¿Dónde está pt? Porque pt es una variable, y por tanto está ubicada en la memoria y tendrá una dirección. Para ver esa dirección, basta aplicar a pt el operador dirección &. pt está en &pt. Tendremos que la variable pt está en R_2.

1. ¿Qué vale pt? Y como pt es un puntero, pt vale o codifica una determinada posición de memoria. Su valor pertenece al dominio de direcciones y está codificado mediante 4 bytes. En concreto, pt vale la dirección de la variable pi. pt vale &pi. Tendremos, por tanto, que pt vale R_1.
2. ¿Qué valor está almacenado en esa dirección de memoria a donde apunta pt? Esta es una pregunta muy interesante, donde se muestra la gran utilidad que tienen los punteros. Podemos llegar al valor de cualquier variable tanto si disponemos de su nombre como si disponemos de su dirección. Podemos llegar al valor de la posición de memoria apuntada por pt, que como es un puntero a **float**, desde el puntero tomará ese byte y los tres siguientes como el lugar donde se aloja una variable **float**. Y para llegar a ese valor, disponemos del operador indirección. El valor codificado en la posición almacenada en pt es el contenido de pi: *pt es 3.14.

Al emplear punteros hay un peligro de confusión, que puede llevar a un inicial desconcierto: al hablar de la dirección del puntero es fácil no entender si nos referimos a la dirección que trae codificada en sus cuatro bytes, o la posición de memoria dónde están esos cuatro bytes del puntero que codifican direcciones.

Es muy importante que las variables puntero estén correctamente direccionadas. Trabajar con punteros a los que no se les ha asignado una dirección concreta conocida (la dirección de una variable) es muy peligroso. En el caso anterior de la variable pi, se puede escribir:

```
*pt = 3.141596;
```

y así se ha cambiado el valor de la variable `pi`, que ahora tiene algunos decimales más de precisión. Pero si la variable `pt` no estuviera correctamente direccionada mediante una asignación previa… ¿en qué zona de la memoria se hubiera escrito ese número `3.141596`? Pues en la posición que, por defecto, hubiera tenido esos cuatro bytes que codifican el valor de la variable `pt`: quizá una dirección de otra variable, o a mitad entre una variable y otra; o en un espacio de memoria no destinado a almacenar datos, sino instrucciones, o el código del sistema operativo,… En general, las consecuencias de usar punteros no inicializados, son catastróficas para la buena marcha de un ordenador. Detrás de un programa que "cuelga" al ordenador, muchas veces hay un puntero no direccionado.

Pero no sólo hay que inicializar las variables puntero: hay que inicializarlas bien, con coherencia. No se puede asignar a un puntero a un tipo de dato concreto la dirección de una variable de un tipo de dato diferente. Por ejemplo:

```
float x, *px;
long y;
px = &y;
```

Si ahora hacemos referencia a *px… ¿trabajaremos la información de la variable y como **long**, o como **float**? Y peor todavía:

```
float x, *px;
char y;
px = &y;
```

Al hacer referencia a *px… ¿leemos la información del byte cuya dirección es la de la variable y, o también se va a tomar en consideración los otros tres bytes consecutivos? Porque la variable px considera que apunta a variables de 4 bytes, que para eso es un puntero a **float**. Pero la posición que le hemos asignado es la de una variable tipo **char**, que únicamente ocupa un byte.

El error de asignar a un puntero la dirección de una variable de tipo de dato distinto al puntero está, de hecho, impedido por el compilador, y si encuentra una asignación de esas características, no compila.

Operadores aritméticos (+, –, ++, ––): los veremos más adelante.

Operadores relacionales (<, <=, >, >=, ==, !=): los veremos más adelante.

Punteros y vectores.

Los punteros sobre variables simples tienen una utilidad clara en las funciones. Allí los veremos con detenimiento. Lo que queremos ver ahora es el uso de punteros sobre arrays.

Un array, o vector, es una colección de variables, todas del mismo tipo, y colocadas en memoria de forma consecutiva. Si creamos una array de cinco variables **float**, y el primero de los elementos queda reservado en la posición de memoria FF54:AB10, entonces no cabe duda que el segundo estará en FF54:AB14 (FF54:AB10 + 4), y el tercero en FF54:AB18 (FF54:AB10 + 8), y el cuarto en FF54:AB1C (FF54:AB10 + 12) y el último en FF54:AB20 (FF54:AB10 + 16): tenga en cuenta que las variables de tipo **float** ocupan 4 bytes.

Supongamos la siguiente situación:

```
long a[10];
long *pa;
pa = &a[0];
```

Y con esto a la vista, pasamos ahora a presentar otros operadores, ya enunciados en el epígrafe anterior, muy usados con los punteros. Son operadores que únicamente deben ser aplicados en punteros que referencian o apuntan a elementos de un array.

Operadores aritméticos (+, –, ++, ––): Estos operadores son parecidos a los aritméticos ya vistos en el Capítulo 7. Pueden verse en la Tabla 7.5. recogida en aquel capítulo, en la fila número 2.

Estos operadores se aplican sobre punteros sumándoles (o restándoles) un valor entero. No estamos hablando de la suma de punteros o su resta. Nos referimos a la suma (o resta) de un entero a un puntero. No tiene sentido sumar o restar direcciones. De todas formas, el compilador permite este tipo de operaciones. Más adelante, en este capítulo, verá cómo se pueden realizar restas de punteros. Y si la suma de un valor de tipo puntero más uno de tipo entero ofrece como resultado un valor de tipo puntero, se verá que la resta de dos valores de tipo puntero (de punteros que apunten a variables ubicadas dentro del mismo array) da como resultado un valor entero. Desconozco qué comportamiento (y qué utilidad) pueda tener la suma de punteros.

Nos referimos pues a incrementar en un valor entero el valor del puntero. Pero, ¿qué sentido tiene incrementar en 1, por ejemplo, una dirección de memoria? El sentido será el de apuntar al siguiente valor situado en la memoria. Y si el puntero es de tipo **double**, y apunta a un array de variables **double**, entonces lo que se espera cuando se incremente un 1 ese puntero es que el valor que codifica ese puntero (el de una dirección de tipo **double** que, como se sabe, es una variable de 8 bytes) se incremente en 8: porque 8 es el número de bytes que deberemos saltar para dejar de apuntar a una variable **double** a pasar a apuntar a otra variable del mismo tipo almacenada de forma consecutiva en el array.

Es decir, que si `pa` contiene la dirección de `a[0]`, entonces `pa + 1` es la dirección del elemento `a[1]`, `pa + 2` es la dirección del elemento `a[2]`, y `pa + 9` es la dirección del elemento `a[9]`.

De la misma manera se define la operación de resta de entero a un puntero. De nuevo al realizar esa operación, el puntero pasará a apuntar a una variable del array ubicada tantas posiciones anteriores a la posición actual como indique el valor del entero sobre el que se realiza la resta. Si, por ejemplo, `pa` contiene la dirección de `a[5]`, entonces `(pa – 2)` contiene la dirección de `a[3]`.

Y así, también podemos definir los operadores incremento (++) y decremento (––), que pasan a significar sumar 1, o restar 1. Y, de la misma manera que se vio en el Capítulo 7, estos dos operadores se comportarán de forma diferente según que sean sufijos o prefijos a la variable puntero sobre la que operan.

En la Figura 16.1. se muestra un posible trozo de mapa de memoria de dos arrays: uno de elementos de 32 bits (4 bytes) y otro de elementos de 16 bits (2 bytes).

Al ir aumentando el valor del puntero, nos vamos desplazando por los distintos elementos del vector, de tal manera que hablar de `a[0]` es lo mismo que hablar de `*pa`; y hablar de `a[1]` es lo mismo que hablar de `*(pa + 1)`; y, en general, hablar de `a[i]` es lo mismo que hablar de `*(pa + i)`. Y lo mismo si comentamos el ejemplo de las variables de tipo **`short`**.

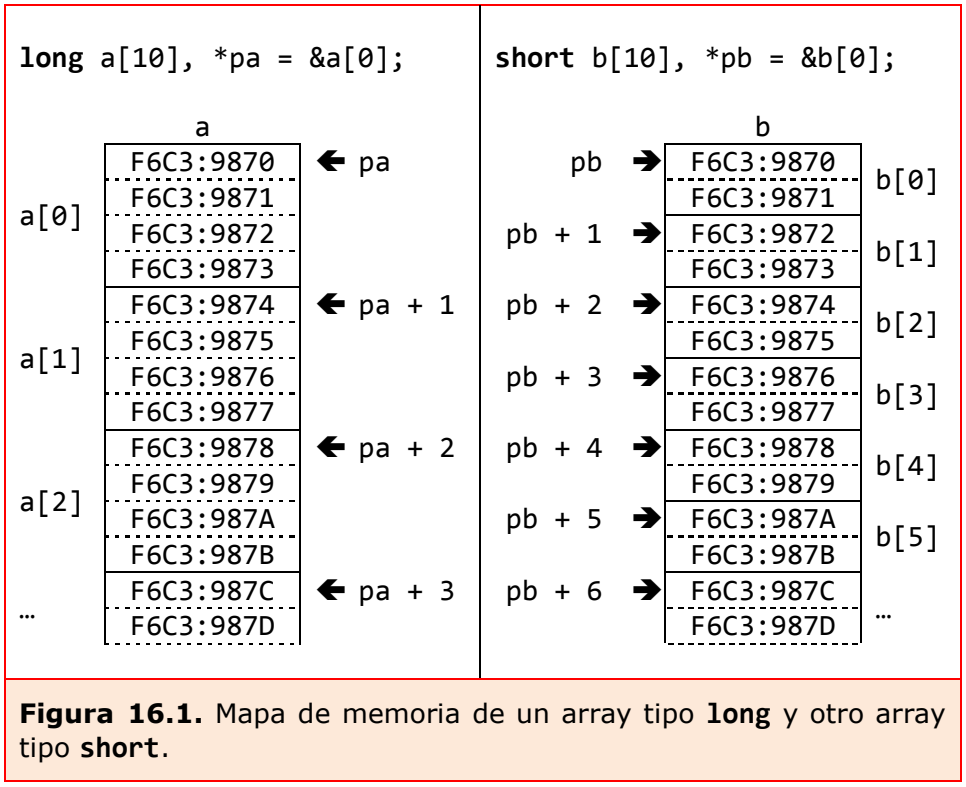


Figura 16.1. Mapa de memoria de un array tipo **`long`** y otro array tipo **`short`**.

La operatoria o aritmética de punteros tiene en cuenta el tamaño de las variables que se recorren. En el programa recogido en Código 16.1., y en la posible salida que ofrece por pantalla y que mostramos, se puede ver este comportamiento de los punteros. Sea cual sea el tipo de dato del puntero y de la variable a la que apunta, si calculamos la resta entre dos punteros situados uno al primer elemento de un array y el otro al último, esa diferencia será la misma, porque la resta de direcciones indica cuántos elementos de este tipo hay (caben) entre esas dos direcciones. En nuestro ejemplo, todas esas diferencias valen 9. Pero si lo que se calcula es el número de bytes entre la última posición (apuntada por el segundo puntero) y la primera (apuntada por el primer puntero), entonces esa diferencia sí dependerá del tamaño de la variable del array.

Código 16.1.

```
#include <stdio.h>
int main(void)
{
      short h[10], *ph1, *ph2;
      double d[10], *pd1, *pd2;

      ph1 = &h[0];      ph2 = &h[9];
      pd1 = &d[0];      pd2 = &d[9];

      printf(" ph2(%p) - ph1(%p) = %hd\n" ,
                              ph2,ph1,ph2 - ph1);
      printf(" pd2(%p) - pd1(%p) = %hd\n" ,
                              pd2,pd1,pd2 - pd1);
      printf("\n\n");
      printf("(long)ph2-(long)ph1=%3ld\n" ,
                              (long)ph2-(long)ph1);
      printf("(long)pd2-(long)pd1=%3ld\n" ,
                              (long)pd2-(long)pd1);

      return 0;
}
```

El programa de Código 16.1. ofrece, por pantalla, el siguiente resultado:

```
ph2(0012FF62) - ph1(0012FF50) = 9
pd2(0012FF20) - pd1(0012FED8) = 9

(long)ph2 - (long)ph1 = 18
(long)pd2 - (long)pd1 = 72
```

Como se advertía antes, sí es posible y tiene su significado y posible utilidad realizar resta de punteros. Más arriba ya decía que no conozco aplicación útil alguna a la suma de valores de tipo puntero.

Repetimos: al calcular la diferencia entre el puntero que apunta al noveno elemento de la matriz y el que apunta al elemento cero, en ambos casos el resultado ha de ser 9: porque en la operatoria de punteros, independientemente del tipo del puntero, lo que se obtiene es el número de elementos que hay entre las dos posiciones de memoria señaladas.

Al convertir las direcciones en valores tipo **`long`**, ya no estamos calculando cuántas variables hay entre ambas direcciones, sino la diferencia entre el valor que codifica la última posición del vector y el valor que codifica la primera dirección. Y en ese caso, el valor será mayor según sea mayor el número de bytes que emplee el tipo de dato referenciado por el puntero. Si es un **`short`**, entre la posición última y la primera hay, efectivamente, 9 elementos; y el número de bytes entre esas dos direcciones es 9 multiplicado por el tamaño de esas variables (que son de 2 bytes): es decir, 18. Si es un **`double`**, entre la posición última y la primera hay, efectivamente y de nuevo, 9 elementos; pero ahora el número de bytes entre esas dos direcciones es 72, porque cada uno de los nueve elementos ocupa ocho bytes de memoria.

Operadores relacionales (<, <=, >, >=, ==, !=): De nuevo estos operadores se aplican únicamente sobre punteros que apunten a posiciones de memoria de variables ubicadas dentro de un array. Estos operadores comparan las ubicaciones de las variables a las que apuntan, y no, desde luego, los valores de las variables a las que apuntan. Si, por

ejemplo, el puntero pa0 apunta a la posición primera del array a (posición de a[0]) y el puntero pa1 apunta a la segunda posición del mismo array (posición de a[1]), entonces la expresión pa0 < pa1 se evalúa como verdadera (distinta de cero).

Índices y operatoria de punteros.

Se puede recorrer un vector, o una cadena de caracteres mediante índices. Y también, mediante operatoria de punteros. Pero además, los arrays y cadenas tienen la siguiente propiedad: Si declaramos ese array o cadena de la siguiente forma:

```
tipo nombre_array[dimensión];
```

El nombre del vector o cadena es nombre_array. Para hacer uso de cada una de las variables, se utiliza el nombre del vector o cadena seguido, entre corchetes, del índice del elemento al que se quiere hacer referencia: nombre_array[índice].

Y ahora introducimos otra novedad: el **nombre del vector o cadena recoge la dirección de la cadena**, es decir, la dirección del primer elemento de la cadena: **decir** nombre_array **es lo mismo que decir** &nombre_array[0].

Y por tanto, y volviendo al código anteriormente visto:

```
long a[10], *pa;
short b[10], *pb;
pa = &a[0];
pb = &b[0];
```

Tenemos que *(pa + i) es lo mismo que a[i]. Y como decir a es equivalente a decir &a[0] entonces, decir pa = &a[0] es lo mismo que decir pa = a, y trabajar con el valor *(pa + i) es lo mismo que trabajar con el valor *(a + i).

Y si podemos considerar que dar el nombre de un vector es equivalente a dar la dirección del primer elemento, entonces podemos considerar

que ese nombre funciona como un puntero constante, con quien se pueden hacer operaciones y formar parte de expresiones, mientras no se le coloque en la parte *Lvalue* de un operador asignación.

Y muchos programadores, en lugar de trabajar con índices, recorren todos sus vectores y cadenas con operatoria o aritmética de punteros.

Veamos un programa sencillo, resuelto mediante índices de vectores (cfr. Código 16.2.) y mediante la operatoria de punteros (cfr. Código 16.3.). Es un programa que solicita al usuario una cadena de caracteres y luego la copia en otra cadena en orden inverso.

En el capítulo en que hemos presentado los arrays hemos indicado que es competencia del programador no recorrer el vector más allá de las posiciones reservadas. Si se llega, mediante operatoria de índices o mediante operatoria de punteros a una posición de memoria que no pertenece realmente al vector, el compilador no detectará error alguno, e incluso puede que tampoco se produzca un error en tiempo de ejecución, pero estaremos accediendo a zona de memoria que quizá se emplea para almacenar otra información. Y entonces alteraremos esos datos de forma inconsiderada, con las consecuencias desastrosas que eso pueda llegar a tener para el buen fin del proceso. Cuando en un programa se llega equivocadamente, mediante operatoria de punteros o de índices, más allá de la zona de memoria reservada, se dice que se ha producido o se ha incurrido en una **violación de memoria**.

Puntero a puntero.

Un puntero es una variable que contiene la dirección de otra variable. Según sea el tipo de variable que va a ser apuntada, así, de ese tipo, debe ser declarado el puntero. Ya lo hemos dicho.

Pero un puntero es también una variable. Y como variable que es, ocupa una porción de memoria: tiene una dirección.

Código 16.2. Código con operatoria de índices.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
	char orig[100], copia[100];
	short i, l;

	printf("Introduzca la cadena ... \n");   gets(orig);
	for(l = strlen(orig) , i = 0 ; i < l ; i++)
		copia[l - i - 1] = orig[i];
	copia[i] = NULL;

	printf("Cadena original: %s\n", orig);
	printf("Cadena copia: %s\n", copia);
	return 0;
}
```

Código 16.3. Código con operatoria de punteros.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
	char orig[100], copia[100];
	short i, l;

	printf("Introduzca la cadena ... \n");   gets(orig);

	for(l = strlen(orig) , i = 0 ; i < l ; i++)
		*(copia + l - i - 1) = *(orig + i);
	*(copia + i) = NULL;

	printf("Cadena original: %s\n", orig);
	printf("Cadena copia: %s\n", copia);
	return 0;
}
```

Se puede, por tanto, crear una variable que almacene la dirección de esa variable puntero. Sería un puntero que almacenaría direcciones de tipo de dato puntero. Un puntero a puntero.

Por ejemplo:

```
float F, *pF, **ppF;
```

Acabamos de crear tres variables: una, de tipo **float**, llamada F. Una segunda variable, de tipo *puntero a* **float**, llamada pF. Y una tercera variable, de tipo *puntero a puntero* **float**, llamada ppF.

Y eso no es un rizo absurdo. Tiene mucha aplicación en C. Igual que se puede hablar de un *puntero a puntero a puntero… a puntero a* **float**.

Y así como antes hemos visto que hay una relación directa entre punteros a un tipo de dato y vectores de este tipo de dato, también veremos ahora que hay una relación directa entre punteros a punteros y matrices de dimensión 2. Y entre punteros a punteros a punteros y matrices de dimensión 3. Y si trabajamos con matrices de dimensión *n*, entonces también lo haremos con punteros a punteros a punteros…

Veamos un ejemplo. Supongamos que creamos la siguiente matriz:

```
double m[4][6];
```

Antes hemos dicho que al crea un array, al hacer referencia a su nombre estamos indicando la dirección del primero de sus elementos. Ahora, al crear esta matriz, la dirección del elemento `m[0][0]` la obtenemos con el nombre de la matriz: Es equivalente decir `m` que decir `&m[0][0]`.

Pero la estructura que se crea al declarar una matriz es algo más compleja que una lista de posiciones de memoria. En el ejemplo expuesto de la matriz **double**, se puede considerar que se han creado cuatro vectores de seis elementos cada uno y colocados en la memoria uno detrás del otro de forma consecutiva. Y cada uno de esos vectores tiene, como todo vector, la posibilidad de ofrecer la dirección de su primer elemento. La Figura 16.2. presenta un esquema de esta

construcción. Aunque resulte desconcertante, no existen los punteros `m`, ni ninguno de los `*(m + i)`. Pero si empleamos el nombre de la matriz de esta forma, entonces trabajamos con sintaxis de punteros.

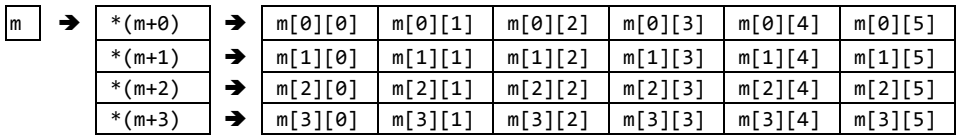


Figura 16.2. Distribución de la memoria en la matriz
double `m[4][6];`

De hecho, si ejecutamos el programa recogido en Código 16.4., obtendremos la siguiente salida:

```
m = 0012FECC
*(m + 0) = 0012FECC     &m[0][0] = 0012FECC
*(m + 1) = 0012FEFC     &m[1][0] = 0012FEFC
*(m + 2) = 0012FF2C     &m[2][0] = 0012FF2C
*(m + 3) = 0012FF5C     &m[3][0] = 0012FF5C
```

Código 16.4.

```
#include <stdio.h>
int main(void)
{
      double m[4][6];
      short i;
      printf("m = %p\n", m);
      for(i = 0 ; i < 4 ; i++)
      {
            printf("*(m + %hd) = %p\t",i, *(m + i));
            printf("&m[%hd][0] = %p\n",i, &m[i][0]);
      }
      return 0:
}
```

Tenemos que `m` vale lo mismo que `*(m + 0)`; su valor es la dirección del primer elemento de la matriz: `m[0][0]`. Después de él, vienen todos los demás, uno detrás de otro: después de `m[0][5]` vendrá el `m[1][0]`, y esa dirección la podemos obtener con `*(m + 1)`; después de `m[1][5]` vendrá el `m[2][0]`, y esa dirección la podemos obtener con `*(m + 2)`; después de `m[2][5]` vendrá el `m[3][0]`, y esa dirección la podemos obtener con `*(m + 3)`; y después de `m[3][5]` se termina la cadena de elementos reservados.

Es decir, en la memoria del ordenador, no se distingue entre un vector de 24 variables tipo **`double`** y una matriz 4 * 6 de variables tipo **`double`**. Es el lenguaje el que sabe interpretar, mediante una operatoria de punteros, una estructura matricial donde sólo se dispone de una secuencia lineal de elementos.

Veamos un programa que calcula el determinante de una matriz de tres por tres (cfr. Código 16.5.). Primero con operatoria de índices, y luego con operatoria de punteros. Quizá la operatoria de punteros en matrices resulta algo farragosa. Pero no encierra dificultad de concepto.

Código 16.5. Determinante matriz 3 X 3. Operatoria de punteros. Operatoria de índices.

```
#include <stdio.h>
int main(void)
{
      double m[3][3] , det;
      short i,j;

      for(i = 0 ; i < 3 ; i++)
            for(j = 0 ; j < 3 ; j++)
            {
                  printf("m[%hd][%hd] = ", i, j);
                  scanf(" %lf",&m[i][j]);
            }
```

Código 16.5. (Cont.).

```
      det = (m[0][0] * m[1][1] * m[2][2]) +
            (m[0][1] * m[1][2] * m[2][0]) +
            (m[0][2] * m[1][0] * m[2][1]) -
            (m[0][2] * m[1][1] * m[2][0]) -
            (m[0][1] * m[1][0] * m[2][2]) -
            (m[0][0] * m[1][2] * m[2][1]);

      printf("El determinante es ... %lf\n\n",det);
      return 0;
}
```

```
      det = *(*(m+0) + 0) * *(*(m+1) + 1) * *(*(m+2) + 2) +
            *(*(m+0) + 1) * *(*(m+1) + 2) * *(*(m+2) + 0) +
            *(*(m+0) + 2) * *(*(m+1) + 0) * *(*(m+2) + 1) -
            *(*(m+0) + 2) * *(*(m+1) + 1) * *(*(m+2) + 0) -
            *(*(m+0) + 1) * *(*(m+1) + 0) * *(*(m+2) + 2) -
            *(*(m+0) + 0) * *(*(m+1) + 2) * *(*(m+2) + 1);
```

Modificador de tipo **const**.

Suponga las dos siguientes expresiones;

```
const double numeroPI = 3.14159265358979323846;
const double numeroE = 2.7182818284590452354;
```

Ambas declaraciones comienzan con la palabra clave **const**. Esta palabra es un modificador de tipo, y así ambas variables creadas (numeroPI y numeroE) verifican que son de tipo **double** y, además, constantes. A partir de esta línea de declaración, el compilador vigilará que ninguna línea de código del programa altere el valor de esas variables: si alguna instrucción pretende variar ese valor, el compilador generara un mensaje de error y el ejecutable no será finalmente creado. Por eso, una variable declarada como **const** no podrá estar en la parte izquierda del operador asignación: no podrá ser *LValue*.

Como se ha visto en estas dos declaraciones, cuando se crea una variable que deba tener un comportamiento constante, es necesario que en la misma instrucción de la declaración se indique su valor. Es el único momento en que una variable **const** puede formar parte de la *LValue* de una asignación: el momento de su declaración.

Ya vimos este modificador de tipo en el Capítulo 7, en el epígrafe titulado "Constantes (variables **const**). Directiva `#define`". Volvemos a este modificador ahora porque juega un papel singular en el comportamiento de los punteros.

Punteros constantes, punteros a constantes y punteros constantes a constantes.

Trabajando con punteros y con el modificador de tipo const, podemos distinguir tres situaciones diferentes:

- PUNTERO A CONSTANTE: Es un puntero cuyo valor es la dirección de una variable (apunta a una variable), no necesariamente **const**, pero que el puntero considerará constante. En este caso, no se podrá modificar el valor de la variable "apuntada" por el puntero; pero sí se podrá variar el valor del puntero: es decir, el puntero podrá "apuntar" a otra variable.

  Por ejemplo:

  ```
  double x = 5, y = 2;
  const double *ptrx = &x;
  double const *ptry = &y;
  ```

  Ambas declaraciones de las variables punteros son equivalentes: ambas declaran un puntero que tratará a la variable "apuntada" como constante.

  Estará por tanto permitido cambiar el valor de las variables x ó y (de hecho no se han declarado como constantes); estará permitido cambiar el valor de los punteros `ptrx` y `ptry` (de hecho esos dos

punteros no son constantes); pero no podremos modificar los valores de las variables x e y mediante la indirección desde los punteros.

```
x *= 2;          // operación permitida.
y /= 2;          // operación permitida.
ptrx = &y;       // operación permitida.
ptry = &x;       // operación permitida.
*ptrx *= 2;      // operación PROHIBIDA.
*ptry /= 2;      // operación PROHIBIDA.
```

Es importante señalar que las variables apuntadas no son necesariamente de tipo **const**. Pero los punteros ptrx y ptry han sido declarados de tal manera que cada uno ellos puede acceder al valor de las variable a la que apunta, pero no puede alterar ese valor.

El uso de este tipo de puntero es muy frecuente en los parámetros de las funciones, y ya los veremos en el Capítulo 17 del manual.

- PUNTERO CONSTANTE: Es un puntero constante (**const**) cuyo valor es la dirección de una variable (apunta a una variable) que no es necesariamente constante. En este caso, el valor del puntero no podrá variarse y, por tanto, el puntero "apuntará" siempre a la misma variable; pero sí se podrá variar el contenido de la variable "apuntada".

  La forma de declarar un puntero contante es la siguiente:

```
double x = 5, y = 2;
double* const ptrx = &x;
double* const ptry = &y;
```

  Con estas nuevas declaraciones, ahora estas son las sentencias permitidas o prohibidas:

```
x *= 2;     // operación permitida.
y /= 2;     // operación permitida.
ptrx = &y;  // operación PROHIBIDA.
ptry = &x;  // operación PROHIBIDA.
*ptrx *= 2; // operación permitida.
*ptry /= 2; // operación permitida.
```

Con un puntero constante está permitido acceder, mediante el operador indirección, al valor de la variable "apuntada", y puede modificarse ese valor de la variable "apuntada", pero no se puede cambiar el valor del puntero: no se puede asignar al puntero la dirección de otra variable.

Es interesante comparar la forma de las declaraciones de los dos tipos de puntero presentados hasta el momento:

```
double* const ptrx = &x;
const double* ptry = &y;
```

El puntero `ptrx` es un puntero constante: no podemos cambiar su valor: siempre apuntará a la variable `x`; el puntero `ptry` es un puntero a constante: sí podemos cambiar el valor del puntero: es decir, podrá apuntar a distintas variables; pero en ningún caso podremos cambiar el valor de esas variables a las que apunta: y eso aunque esas variables no se hayan declarado con el modificador **`const`**.: no es que las variables sean constantes (aunque, desde luego, podrían serlo): es que el puntero no puede hacer modificación sobre ellas.

- PUNTERO CONSTANTE A CONSTANTE: Es un puntero constante (**`const`**) cuyo valor es la dirección de una variable (apunta a una variable) sobre la que no podrá hacer modificación de su valor.

  Es una mezcla o combinación de las dos definiciones anteriores. El modo en que se declara un puntero constante a constante es el siguiente:

  ```
  double x = 5;
  const double * const ptrx = &x
  ```

  El puntero `ptrx` es constante, por lo que no podremos cambiar su valor: "apunta" a la variable `x`, y no podemos asignarle cualquier otra dirección de cualquier otra variable. Y además es un puntero a constante, por lo que no podremos tampoco cambiar, mediante indirección, el valor de la variable `x`.

Vea la siguiente lista de declaraciones y sentencias, algunas de las cuales son correctas y otras no, ya que violan alguna de las condiciones establecidas por la palabra clave **const**.

```
double x = 5, y = 2, z = 7;
const double xx = 25;
const double *ptrx = &x;           // a constante
double* const ptry = &y;           // constante
const double* const ptrz = &z;     // constante a constante.

x = 1;      // PERMITIDO: x no es const.
*ptrx = 1;  // PROHIBIDO: ptrx es un puntero a constante.
ptrx = &xx; // PERMITIDO: ptrx no es un puntero constante.

xx = 30;    // PROHIBIDO: xx es const.

y = 1;      // PERMITIDO: y no es const.
*ptry = 1;  // PERMITIDO: ptry no es un puntero a constante.
ptry = &x;  // PROHIBIDO: ptry es un puntero constante.

z = 1;      // PERMITIDO: z no es const.
ptrz = &x;  // PROHIBIDO: ptrz es un puntero constante.
*ptrz = 1;  // PROHIBIDO: ptrz es un puntero a constante.
```

Es importante utilizar adecuadamente este modificador (**const**) en la declaración y uso de los punteros. En Código 16.6. se recoge un ejemplo sencillo, donde se produce una alteración, a través de un puntero, del valor de una variable declarada como **const**.

Código 16.6.

```
int main(void)
{
      const double x = 3;
      double *ptrx = &x;

      *ptrx = 5;
      printf("El valor de x es ... %.2lf\n", x);
      return 0;
}
```

La variable x es constante: no se debería poder modificar su valor, una vez inicializada al valor 3. Pero no hemos declarado el puntero `ptrx` como **`const`**. Y, por tanto, desde él se puede acceder a la posición de memoria de la variable x y proceder a su cambio de valor.

La práctica correcta habitual será que una variable declarada como **`const`** únicamente pueda ser "apuntada" por un puntero que preserve esa condición. Pero eso no es algo que vigile el compilador, y el código arriba copiado compila correctamente (según cómo tenga configurado el compilador quizá pueda obtener un mensaje de advertencia o warning) y muestra por pantalla en mensaje

```
El valor de x es ... 5.00
```

Punteros fuera del ámbito de la variable a la que "apuntan".

Con un puntero podemos acceder a la información de variables que están fuera del ámbito actual. E incluso podemos variar su valor. De hecho esta capacidad de alcanzar al valor de una variable desde fuera de su ámbito es un uso muy frecuente para los punteros en las llamadas a funciones, como veremos en el Capítulo 17.

Ejercicios.

Uuna buena forma de aprender a manejar punteros es intentar rehacer los ejercicios ya resueltos en los capítulos de vectores y de cadenas de texto (Capítulos 11 y 12) empleando ahora operatoria de punteros.

Recuerde:

- Decir `a[i]`, es equivalente a decir `*(a + i)`
- Decir `&a[i]` es equivalente a decir `a + i`.
- Decir `a[i][j]` es equivalente a decir `*(*a + i) + j)`.
- Y decir `&a[i][j]` es equivalente a decir `(*a + i) + j)`.

CAPÍTULO 17

FUNCIONES Y PARÁMETROS CON PUNTEROS.

Ya hemos visto un primer capítulo que versaba sobre las funciones. Luego hemos presentado otro capítulo que presentaba una forma especial de definir funciones: aquellas que se definían por recursividad: aquellas que recurrían a sí mismas para establecer su propia definición. Pero en ambos casos hemos trabajado con parámetros sencillos: variables declaradas en el ámbito de la función que invocaba a la nuestra implementada.

Pero a veces es necesario utilizar punteros en los parámetros de las funciones. ¿Qué ocurre si deseamos que una función realice cambios en los valores de varias variables definidas en el ámbito de la función que invoca? ¿O cómo procedemos si lo que deseamos pasar como parámetro a una función no es una variable sino una matriz, de por ejemplo, 10 por 10 elementos?

Llamadas por valor y llamadas por referencia.

Introducimos aquí dos nuevos conceptos, tradicionales al hablar de funciones. Y muy importantes. Hacen referencia al modo en que la función recibe los parámetros.

Hasta ahora, en todos los ejemplos previos presentados en los Capítulos 14 y 15, hemos trabajado haciendo **llamadas "por valor"**. Decimos que una función es llamada por valor cuando se copia el valor del argumento en el parámetro formal de la función. Una variable está en la función que llama; y otra variable, distinta, es la que recibe el valor en la función llamada. La función llamada no puede alterar el valor del argumento original de la función que llama. Únicamente puede cambiar el valor de su variable local que ha recibido por asignación el valor de esa variable en el momento en que se realizó la llamada a la función. Así, en la función llamada, cada argumento es efectivamente una variable local inicializada con el valor con que se llamó a la función.

Pero supongamos, por ejemplo, que necesitamos en nuestro programa realizar con mucha frecuencia la tarea de intercambiar el valor de dos variables. Ya sabemos cómo se hace, y lo hemos visto resuelto tanto a través de una variable auxiliar como (para variables enteras) gracias al operador or exclusivo. Sería muy conveniente disponer de una función a la que se le pudieran pasar, una y otra vez, el par de variables de las que deseamos intercambiar sus valores. Pero ¿cómo lograr hacer ese intercambio a través de una función si todo lo que se realiza en la función llamada "muere" cuando termina su ejecución? ¿Cómo lograr que en la función que invoca ocurra realmente el intercambio de valores entre esas dos variables?

La respuesta no es trivial: cuando invocamos a la función (que llamaremos en nuestro ejemplo `intercambio`), las variables que deseamos intercambiar dejan de estar en su ámbito y no llegamos a ellas. Toda operación en memoria que realice la función `intercambio` morirá con su última sentencia: su único rastro será, si acaso, la

obtención de un resultado, el que logra sobrevivir de la función gracias a la sentencia **return**.

Y aquí llegamos a la necesidad de establecer otro tipo de llamadas a funciones: las llamadas "**por referencia**". En este tipo de llamada, lo que se transfiere a la función no es el valor del argumento, sino la dirección de memoria de la variable argumento. Se copia la dirección del argumento en el parámetro formal, y no su valor.

Evidentemente, en ese caso, el parámetro formal deberá ser de tipo puntero. En ese momento, la variable argumento quedará fuera de ámbito, pero a través del puntero correspondiente en los parámetros formales podrá llegar a ella, y modificar su valor.

Código 17.1. muestra el prototipo y la definición de la función `intercambio`. Supongamos que la función que llama a la función intercambio lo hace de la siguiente forma:

```
intercambio(&x,&y);
```

Código 17.1. Función `intercambio`.

```
// Declaración
void intercambio(long* , long*);
// Definición
void intercambio(long*a , long*b)
{
      *a ^= *b;   *b ^= *a;   *a ^= *b;   }
```

```
// Otra definición, con el operador or exclusivo
void intercambio(double*a , double*b)
{
      double aux = *b;
      *b = *a;
      *a = aux;
}
```

Donde lo que se le pasa a la función `intercambio` son las direcciones (no los valores) de las dos variables de las que se desea intercambiar sus valores.

En la función llamante tenemos:

<x, long, R_x, V_x> y <y, long, R_y , V_y>

En la función `intercambio` tenemos:

<a, long*, R_a, R_x> y <b, long*, R_b, R_y>

Es decir, dos variables puntero cuyos valores que se le van asignar serán las posiciones de memoria de cada una de las dos variables usadas como argumento, y que son con las que se desea realizar el intercambio de valores.

La función trabaja sobre los contenidos de las posiciones de memoria apuntadas por los dos punteros. Y cuando termina la ejecución de la función, efectivamente, mueren las dos variables puntero a y b creadas. Pero ya han dejado hecha la faena en las direcciones que recibieron al ser creadas: en R_x ahora queda codificado el valor V_y; y en R_y queda codificado el valor V_x. Y en cuanto termina la ejecución de intercambio regresamos al ámbito de esas dos variables x e y: y nos las encontramos con los valores intercambiados.

Muchos son los ejemplos de funciones que, al ser invocadas, reciben los parámetros por referencia. La función `scanf` recibe el parámetro de la variable sobre la que el usuario deberá indicar su valor con una llamada por referencia. También lo hemos visto en la función `gets`, que recibe como parámetro la dirección de la cadena de caracteres donde se almacenará la cadena que introduzca el usuario.

Por otro lado, siempre que deseemos que una función nos devuelva más de un valor también será necesario utilizar llamadas por referencia: uno de los valores deseamos podremos recibirlo gracias a la sentencia **`return`** de la función llamada; los demás podrán quedar en los

argumentos pasados como referencia: entregamos a la función sus direcciones, y ella, al terminar, deja en esas posiciones de memoria los resultados deseados.

Vectores (arrays monodimensionales) como argumentos.

No es posible pasar, como parámetro en la llamada a una función, toda la información de un array en bloque, valor a valor, de variable a variable. Pero sí podemos pasar la dirección del primer elemento de ese array. Y eso resulta a fin de cuentas una operación con el mismo efecto que si pasáramos una copia del array. Con la diferencia, claro está, de que ahora la función invocada tiene no solo la posibilidad de conocer esos valores del array declarado en la función llamante, sino que también puede modificar esos valores, y que entonces esos valores quedan modificados en la función que llama.

Y, desde luego, esta operación siempre resulta más eficiente que hacer una copia variable a variable, de los valores en la función llamante a los valores en la función llamada o invocada.

El modo más sencillo de hacer eso será:

void ff(**long** [100]); o también **void** funcion(**long** arg[100]);

Y así, la función recibe la dirección de un array de 100 elementos tipo **long**. Dentro del código de esta función podremos acceder a cada una de las 100 variables, porque en realidad, al invocar a la función, se le ha pasado la dirección del array que se quiere pasar como parámetro. Por ejemplo:

```
long a[100];      ff(a);
```

Otro modo de declarar ese parámetro es mediante un puntero del mismo tipo que el del array que recibirá:

void ff(**long***); ó también **void** ff(**long***arg);

Y la invocación de la función se realizará de la misma forma que antes. Desde luego, en esta forma de declaración queda pendiente dar a conocer la dimensión del array: puede hacerse mediante un segundo parámetro. Por ejemplo:

void ff(**long** [],**short**); o también **void** ff(**long** arg[],**short** d);
void ff(**long***,**short**); ó también **void** ff(**long***arg,**short** d);
La llamada de la función usará el nombre del vector como argumento, ya que, como dijimos al presentar los arrays y las cadenas, el nombre de un array o cadena, en C, indica su dirección: decir nombre_vector es lo mismo que decir &nombre_vector[0].

Evidentemente, y como siempre, el tipo de dato puntero del parámetro formal debe ser compatible con el tipo de dato del vector argumento.

Veamos un ejemplo (Código 17.2.). Hagamos una aplicación que reciba un array tipo **float** y nos indique cuáles son el menor y el mayor de sus valores. Entre los parámetros de la función será necesario indicar también la dimensión del vector. A la función la llamaremos extremos.

Código 17.2. Función extremos.

```
// Prototipo o declaración
void extremos(float v[], short d, float*M, float*m);
// Definición
void extremos(float v[], short d, float*M, float*m)
{
      short int i;
      *M = *v;
      *m = *v;

      for(i = 0 ; i < d ; i++)
      {
            if(*M < *(v + i)) *M = *(v + i);
            if(*m > *(v + i)) *m = *(v + i);
      }
}
```

Lo primero que hemos de pensar es cómo pensamos devolver, a la función que llame a nuestra función, los dos valores solicitados. Repito: **DOS** valores solicitados. No podremos hacerlo mediante un **return**, porque así sólo podríamos facilitar uno de los dos. Por eso, entre los parámetros de la función también necesitamos dos que sean las direcciones donde deberemos dejar recogidos el mayor de los valores y el menor de ellos.

El primer parámetro es la dirección del array donde se recogen todos los valores. En segundo parámetro la dimensión del array. El tercero y el cuarto las direcciones donde se consignarán los valores mayor (variable `M`) y menor (variable `m`) del array. Como todos los valores que la función determina ya quedan reservados en las variables que se le pasan como puntero, ésta no necesita "devolver" ningún valor, y se declara, por tanto, de tipo **void**.

Inicialmente hemos puesto como menor y como mayor el primero de los elementos del vector. Y luego lo hemos recorrido, y siempre que hemos encontrado un valor mayor que el que teníamos consignado como mayor, hemos cambiado y hemos guardado ese nuevo valor como el mayor; y lo mismo hemos hecho con el menor. Y al terminar de recorrer el vector, ya han quedado esos dos valores guardados en las direcciones de memoria que hemos recibido como parámetros.

Para llamar a esta función bastará la siguiente sentencia:

```
extremos(vector, dimension, &mayor, &menor);
```

La recepción de un vector como parámetro formal no necesariamente debe hacerse desde el primer elemento del vector. Supongamos que al implementar la función `extremos` exigimos, como especificación técnica de esa función, que el vector tenga una dimensión impar. Y diremos que reciba como parámetros los mismos que antes. Pero ahora la dirección de memoria del vector será la del elemento que esté a la mitad del vector. Si la dimensión es n. el usuario de la función deberá pasar como

argumento la dirección del elemento de índice (n – 1) / 2. Y el argumento segundo deberá ser, en lugar de la dimensión del vector, el valor indicado (n – 1) / 2. Código 17.3. recoge una posible implementación.

Código 17.3. Otra implementación de la función `extremos`.

```
void extremos(float v[], unsigned short d, float*M, float*m)
{
      short int i;

      *M = *v;
      *m = *v;

      for(i = 0 ; i < d ; i++)
      {
            if(*M < *(v + i)) *M = *(v + i);
            if(*M < *(v - i)) *M = *(v - i);
            if(*m > *(v + i)) *m = *(v + i);
            if(*M < *(v - i)) *M = *(v - i);
      }
}
```

Así logramos la misma operación de búsqueda y hemos reducido a la mitad los incrementos de la variable contador `i`. No entramos ahora en analizar la oportunidad de esta nueva implementación; el código puede hacer lo que a nosotros nos convenga más. Lo importante en este caso es dejar bien especificadas las condiciones para el correcto uso de la función. Y vigilar cuáles son los límites del vector para evitar que se acceda a posiciones de memoria fuera de la dimensión del vector.

Es buena práctica de programación trabajar con los punteros, en los pasos por referencia, del siguiente modo: si el parámetro a recibir corresponde a un array, entonces es conveniente que en la declaración y en la definición de la función se indique que ese parámetro recogerá la

dirección de un array, y no la de una variable simple cualquiera. Así, si el prototipo de una función es, por ejemplo,

```
void function(double a[], long *b);
```

se comprende que el primer parámetro recogerá la dirección de un array de tipo **double**, mientras que el segundo recoge la dirección de una variable sencilla tipo **long**.

Y en la definición de la función es conveniente utilizar operadores aritméticos o relacionales sobre aquellos parámetros puntero que se hayan declarado como arrays.

Esta práctica de buena programación facilita luego la comprensión del código. Y también permite al usuario comprender, cuando necesita comprender la función y consulta su prototipo, qué parámetros sirven para pasar la referencia de un array y cuáles son para referencia hacia variables simples.

Matrices (arrays multimensionales) como argumentos.

Para comprender bien el modo en que se puede pasar una matriz como parámetro de una función es necesario comprender cómo define realmente una matriz el compilador de C. Esto ya ha quedado explicado en los capítulos 11 y 16. Podríamos resumir lo que allí se explicaba diciendo que cuando declaramos una matriz, por ejemplo, `double a[4][9];` en realidad hemos creado un array de 4 "cosas" (por llamarlo por ahora de alguna forma): ¿Qué "cosas" son esas?: son arrays de dimensión 9 de variables tipo **double**. Esta matriz es, pues, un array de 4 elementos, cada uno de los cuales es un array de 9 variables tipo **double**. Podríamos decir que tenemos un array de 4 elementos de tipo array de 9 elementos de tipo **double**. Es importante, para comprender el paso entre funciones de matrices por referencia, entender de esta forma lo que una matriz es.

Así las cosas veamos entonces cómo podemos recibir esa matriz antes declarada como parámetro de una función.

La primera forma sería:

```
void ff(double [4][9]);
```
ó también
```
void ff(double arg[4][9]);
```

Y entonces, podremos pasar a esta función cualquier matriz creada con estas dimensiones.

Pero posiblemente nos pueda interesar crear una función donde podamos pasarle una matriz de cualquier tamaño previo. Y puede parecer que una forma de hacerlo podría ser, de la misma forma que antes con el array monodimensional, de una de las siguientes dos formas:

```
void ff(long [][]);
void ff(long arg[][]);
```
(No lo aprenda: son formas erróneas.)

o mediante el uso de punteros, ahora con indirección múltiple:

```
void ff(long**);
void ff(long **arg);
```
(No lo aprenda: son formas erróneas.)

Pero **todas estas formas son erróneas**. ¿Por qué? Pues porque el compilador de C necesita conocer el tipo de dato del array que recibe como parámetro: hemos dicho antes que una matriz hay que considerarla aquí como un array de elementos de tipo array de una determinada cantidad de variables. El compilador necesita conocer el tamaño de los elementos del array. No le basta saber que es una estructura de doble indirección o de matriz. Cuando se trata de una matriz (de nuevo, por ejemplo, `double a[4][9]`), debemos indicarle al compilador el tipo de cada uno de los 4 elementos del array: en este caso, tipo array de 9 elementos de tipo **double**.

Sí es correcto, por tanto, declarar la función de esta forma:

```
void ff(double [][9]); o también
void ff(double arg[][9]);
```

Donde así damos a conocer a la función el tamaño de los elementos de nuestro array. Como ya quedó explicado, para la operatoria de los punteros no importa especialmente cuántos elementos tenga el array, pero sí es necesario conocer el tipo de dato de esos elementos.

La función podría declararse también de esta otra forma:

```
void ff(double (*)[9]);
```
o también
```
void ff(double (*arg)[9]);
```

Donde así estamos indicando que arg es un puntero a un array de 9 elementos de tipo **double**. Obsérvese que el parámetro no es **double** *m[9], que eso significaría un array de 9 elementos tipo puntero a **double**, que no es lo que queremos expresar.

En todas estas declaraciones convendría añadir dos nuevos parámetros que indicaran las dimensiones (número de filas y de columnas de la matriz que se recibe como parámetro. Así, las declaraciones vistas en este apartado podrían quedar así:

```
void ff(double [][9], short, short);
void ff(double (*)[9], short, short);
```

C99: Simplificación en el modo de pasar matrices entre funciones.

La verdad es que todas formas de declaración de parámetros para las matrices son bastante engorrosas y de lectura oscura. Al final se va generando un código de difícil comprensión.

Como se vio en el capítulo 11, con el estándar C99 se ha introducido la posibilidad de declarar los arrays y las matrices indicando sus dimensiones mediante variables. Esto simplifica grandemente la declaración de parámetros como matrices no de tamaño predeterminado en tiempo de compilación, y permite declarar las funciones que requieren recibir parámetros de matrices de la siguiente forma:

```
void ff(short x, short y, double arg [x][y]);
```

Y, desde luego, el problema de pasar una matriz como parámetro en una función queda simplificado.

Así, si deseamos declarar y definir una función que muestre por pantalla una matriz que recibe como parámetro, podemos hacerlo de acuerdo al estándar C90 (cfr. Código 17.4.), o al C99 (cfr. Código 17.5.).

Código 17.4. propone dos formas de declarar y encabezar la función; es necesario concretar el tamaño del array (4 en nuestro ejemplo), porque eso determina el tipo del puntero (array arrays de 4 elementos tipo **double**): sin esa dimensión la declaración quedaría incompleta.

Código 17.4. Matriz como parámetro en una función. Estándar C90.

```
void mostrarMatriz(double (*)[], short, short);
// void mostrarMatriz(double [][4], short, short);
[...]
void mostrarMatriz(double (*m)[4], short f, short c)
// void mostrarMatriz(double m[][4], short f, short c)
{
      short i, j;
      for(i = 0 ; i < f ; i++)
      {
            for(j = 0 ; j < c ; j++)
                  printf("%6.2lf", *(*(m + i) + j));
            printf("\n");
      }
      return;
}
```

Con el estándar C99 (cfr. Código 17.5.) es necesario que la declaración de los parámetros `f` y `c` vayan a la izquierda de la declaración de la matriz: de lo contrario el compilador daría error por desconocer, en el momento de la declaración de la matriz, el significado y valor de esas dos variables.

Código 17.5. Matriz como parámetro en una función. Estándar C99.

```
// Declaración
void mostrarMatriz(short f, short c, double [f][c]);
[...]
// Definición
void mostrarMatriz(short f, short c, double m[f][c])
{
      short i, j;

      for(i = 0 ; i < f ; i++)
      {
            for(j = 0 ; j < c ; j++)
                  printf("%6.2lf", *(*(m + i) + j));
            printf("\n");
      }
      return;
}
```

Argumentos tipo puntero constante.

Ya hemos explicado en el Capítulo 16 cómo los punteros se pueden definir como "constantes" (no se puede cambiar su valor) o como "a constantes" (no se puede variar el valor de la variable apuntada).

Con frecuencia una función debe recibir, como parámetro, la dirección de un array. Y también frecuentemente ocurre que esa función no ha de realizar modificación alguna sobre los contenidos de las posiciones del array sino simplemente leer los valores. En ese supuesto, es práctica habitual declarar el parámetro del array como de tipo **const**. Así se ofrece una garantía al usuario de la función de que realmente no se producirá modificación alguna a los valores de las posiciones del array que pasa como parámetro.

Eso se puede ver en muchos prototipos de funciones ya presentadas: por ejemplo, las vistas en el Capítulo 12. Veamos algunos ejemplos de prototipos de funciones del archivo de cabecera `string.h`:

```
char *strcpy      (char s1[], const char s2[]);
int strcmp        (const char s1[], const char s2[]);
char *strcat      (char s1[], const char s2[]);
```

El prototipo de la función `strcpy` garantiza al usuario que la cadena recibida como segundo parámetro no sufrirá alteración alguna; no así la primera, que deberá ser, finalmente, igual a la cadena recibida como segundo parámetro.

Lo mismo podríamos decir de la función `strcmp`, que lo que hace es comparar las dos cadenas recibidas. En ese caso, la función devuelve un valor entero menor que cero si la primera cadena es menor que la segunda, o mayor que cero si la segunda es menor que la primera, o igual a cero si ambas cadenas son iguales. Y lo que garantiza el prototipo de esta función es que en todo el proceso de análisis de ambas cadenas, ninguna de las dos sufrirá alteración alguna.

Y lo mismo podremos decir de la función `strcat`. El primer parámetro sufrirá alteración pues al final de la cadena de texto contenida se le adjuntará la cadena recibida en el segundo parámetro. Lo que queda garantizado en el prototipo de la función es que el segundo parámetro no sufrirá alteración alguna.

Muchas de las funciones que hemos presentado en las páginas previas en el presente capítulo bien podrían haber declarado alguno de sus parámetros como **`const`**. Hubiera sido mejor práctica de programación.

Y hay que tener en cuenta, al implementar las funciones que reciben parámetros por referencia declarados como punteros a constante, que esas funciones no podrán a su vez invocar y pasar como parámetro esos punteros a otras funciones que no mantengan esa garantía de constancia de los valores apuntados. Por ejemplo, el programa propuesto en Código 17.6. no podría compilar.

La función `function01` recibirá como parámetro un array de constantes: no debe poder alterar los valores de las posiciones del array (cfr. línea

/*01*/). Por eso, el código de la línea 4 es erróneo, y no podrá terminar el proceso de compilado del programa.

Código 17.6. Erróneo. No compila.

```
            #include <stdio.h>

/*01*/      void function01(const long*);
/*02*/      void function02(long*);

            int main(void)
            {
                  long a[10];
/*03*/            function01(a);
                  return 0;
            }

            void function01(const long* vc)
            {
                  // Código ...
/*04*/            *vc = 11;
/*05*/            function02(vc);
                  // Código ...
                  return;
            }
            void function02(long* v)
            {
                  // Código ...
/*06*/            *v = 12;
                  return;
            }
```

La función function02 no tiene como parámetro el puntero a constante (cfr. línea /*02*/): podrá por tanto alterar los valores del array: por eso la línea /*06*/ no presenta problema alguno. Pero… ¿qué ocurre con la sentencia de la línea /*05*/, donde lo que hacemos es pasar a la función function02 un puntero a constante que la función function02 no

tratará como tal? Porque el hecho es que el prototipo de la función `function01` pretendía garantizar que los valores del array iban a ser recibidos como de sólo lectura, pero luego la realidad es que esos valores sufren modificación: véase la línea `/*06*/`, donde la función `function02` modifica de hecho un elemento de ese array que le pasa la función `function01` y que ésta había recibido como a constante. Entonces… ¿Qué ocurre?: pues dependerá de la configuración del compilador que el programa no termine de crearse y se muestre error, o que se presente únicamente una advertencia de que un valor o un array de valores de sólo lectura puede o pueden sufrir, en la llamada a esa función `function02`, una alteración no permitida por el compromiso adquirido en el prototipo de la función `function01`.

Al margen de la configuración de su compilador, sí es conveniente señalar que no es buena práctica de programación ésa de no cumplir con los compromisos presentados en el prototipo de una función. Por tanto, en una función donde se trabaja con parámetros a constante, esos parámetros deberían ser a su vez pasados por referencia a otras funciones sólo si éstas, a su vez, garantizan la no alteración de valores.

Recapitulación.

Completando la visión del uso de funciones ya iniciada en los capítulos 14 y 15, hemos presentado ahora el modo en que a una función se le puede pasar, como parámetro, no un valor concreto de una variable, sino su propia dirección, permitiendo así que la función invocada pueda, de hecho, modificar los valores de las variables definidas en el ámbito de la función que invoca. También hemos aprendido cómo una función puede recibir todo un array o una matriz de valores.

Ejercicios.

17.1. ***Escriba una función que reciba como parámetros un vector de enteros y un entero que recoja la dimensión del vector, y devuelva ese vector, con los valores enteros ordenados de menor a mayor.***

Ya conocemos el algoritmo de ordenación de la burbuja. Lo hemos utilizado anteriormente. Código 17.7. propone una implementación que utiliza esta función.

Código 17.7. Posible solución al Ejercicio propuesto 17.1.

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

#define TAM 100

// Declaración de las funciones ...
void asignar(long *, short);
void ordenar(long *, short);
void mostrar(long *, short);

// Función principal ...
int main(void)          {
      long vector[TAM];

      asignar(vector, TAM); // Asignar valores.
      mostrar(vector, TAM); // Antes de ordenar.
      ordenar(vector, TAM); // Se ordena el vector.
      mostrar(vector, TAM); // Despúes de ordenar.
      return 0;
}
```

Código 17.7. (Cont.).

```
/* ------------------------------------------------------ */
/*     Definición de las funciones                        */
/* ------------------------------------------------------ */
/* Función de asignación. ------------------------------ */
void asignar(long *v, short d)                          {
      short i;
      for(i = 0 ; i < d ; i++)
      {
            printf("Valor %hd ... “ , i + 1);
            scanf("%ld", v + i);
      }
}
/* Función de ordenación. ------------------------------ */
void ordenar(long *v , short d)                         {
      short i, j;
      for(i = 0 ; i < d ; i++)
            for(j = i + 1 ; j < d  ; j++)
                  if(*(v + i) > *(v + j))               {
                        *(v + i) ^= *(v + j);
                        *(v + j) ^= *(v + i);
                        *(v + i) ^= *(v + j);
                  }
}
/* Función q muestra el vector que recibe como parámetro. */
void mostrar(long *v , short d)                         {
      short i;
      printf("\n\n");
      for(i = 0 ; i < d ; i++)
            printf("%5ld", *(v + i));
}
```

En todo momento se ha utilizado en las funciones la aritmética de punteros y el operador indirección. Evidentemente se podría hacer lo mismo con los índices del vector. De hecho la expresión `*(v + i)` es siempre intercambiable por la expresión `v[i]`.

17.2. ***Escriba el código de una función que reciba un entero y busque todos sus divisores, dejándolos en un vector.***

Esta función recibirá como parámetros el entero sobre el que hay que buscar sus divisores, el vector donde debe dejar los enteros, y la dimensión del vector.

La función devuelve un valor positivo igual al número de divisores hallados si el proceso termina correctamente. Deberá devolver un valor negativo si falla en el proceso de búsqueda de los divisores.

Código 17.8. Posible solución al Ejercicio propuesto 17.2.

```
#include <stdio.h>

// Declaración de la función ...
short divisores(long, long*, short);

// Definición de la función ...
short divisores(long N, long *v , short d)
{
      short cont = 1;
      long div;

      v[0] = 1;
      for(div = 2 ; div <= N / 2 ; div++)
            if(N % div == 0)
            {
                  v[cont++] = div;
// Si no caben más divisores, se aborta la operación.
                  if(cont >= d) return -1;
            }
      v[cont++] = N;
      return cont;
}
```

17.3. ***Escriba una función que calcule el máximo común divisor de un conjunto de enteros que recibe en un vector como parámetro. Además, la función recibe otro parámetro que es el número de enteros recogidos en ese vector.***

Código 17.9. Posible solución al Ejercicio propuesto 17.3.

```
// Declaración de las funciones ...
long mcd(long, long);
long MCD(long*,short);

// Definición de las funciones ...
// mcd de dos enteros. Definición por recurrencia.
long mcd(long a, long b)
{
      return b ? mcd(b , a % b) : a;
}

// Función cálculo mcd de varios enteros ...
long MCD(long*n , short d)
{
      short i;
      long m;

// Si algún entero es 0, el mcd lo ponemos a cero.
      for(i = 0 ; i < d ; i++)
            if(!*(n + i)) return 0;
// Si no hay enteros, el mcd lo ponemos a cero.
      if(!d) return 0;
// Si solo hay un entero, el mcd es ese entero.
      if(d == 1) return *(n + 0);
// Cálculo del mcd de los distintos valores del array
      i = 2;
      m = mcd(*(n + 0) , *(n + 1));
      while(i < d) m = mcd(m , *(n + i++));

      return m;
}
```

Hemos definido (cfr. Código 17.9.) dos funciones: una que devuelve el valor del máximo común divisor de dos enteros, y otra, que es la solicitada en esta pregunta, que calcula el máximo común divisor de una colección de enteros que recibe en un array.

La función `MCD` considera varias posibilidades: que uno de los valores introducidos sea un 0 o que no se haya recibido ningún valor (`d == 0`), y en tal caso devuelve como valor calculado el valor 0; y que sólo se reciba un entero (`d == 1`) y en ese caso devuelve ese valor único.

17.4. ***Defina una función que reciba como parámetro un array de enteros largos. Tantos enteros como indique el segundo parámetro de la función. Esta función deberá devolver el valor +1 si todos los valores del array son diferentes, y el valor 0 si hay alguna repetición.***

El prototipo de la función es:

```
short test_valores(long*, long);
```

Código 17.10. Posible solución al Ejercicio propuesto 17.4.

```
short test_valores(long*a, long d)
{
      long i, j;

      for(i = 0 ; i < d ; i++)
            for(j = i + 1 ; j < d ; j++)
                  if(*(a + i) == *(a + j)) return 0;

      return 1;
}
```

17.5. ***Escriba una función cuyo prototipo es***

```
short isNumero(char*, short);
```

que recibe una cadena de texto como primer parámetro de entrada, y la dimensión de esta variable cadena como segundo parámetro, y devuelve un valor verdadero (distinto de cero) si la cadena contiene únicamente caracteres numéricos, y devuelve un valor falso (igual a cero) si la cadena contiene algún carácter que no sea numérico.

En Código 17.11. se muestra la definición de la función solicitada.

Código 17.11. Posible solución al Ejercicio propuesto 17.5.

```
#include <ctype.h>

short isNumero(char* c, short d)
{
      int i;
      for(i = 0 ; i < d && c[i] != NULL ; i++)
            if(!isdigit(c[i])) return 0;
      return 1;
}
```

Si encuentra algún carácter no dígito se interrumpe la ejecución de la iteración gobernada por la estructura **for** y se devuelve el valor 0. Si se logra finalizar la ejecución de todas las iteraciones (se llega al carácter nulo) y no se ha ejecutado esa salida, entonces es que todos los caracteres, hasta llegar a ése carácter nulo, son dígitos, y entonces la función devuelve un 1.

17.6. ***Escriba una función cuyo prototipo es***

```
long numero(char*, short);
```

que recibe como primer parámetro una cadena de texto y como segundo parámetro la dimensión de esa variable cadena, y devuelve (si esa cadena está formada sólo por dígitos) el valor numérico de la cifra codificada; en caso contrario (algún carácter no es dígito) devuelve el valor -1. Por ejemplo, si recibe la cadena "2340" debe devolver el valor 2340; si es "234r" debe devolver el valor -1.

Código 17.12. Posible solución al Ejercicio propuesto 17.6.

```
long numero(char* c, short d)
{
      if(!isnumero(c, d)) return -1;
      long n = 0;
      for(int i = 0 ; i < d && c[i] ; i++)
            n = 10 * n + digito(c[i]);
      return n;
}
```

En la función propuesta en Código 17.12. hemos hecho uso de la función `isNumero` definida en Código 17.11. y de la función `digito` definida en Código 14.12.

Si no son dígitos todos los elementos, directamente devuelve el valor -1. Si son todo dígitos, entonces calculamos el valor numérico que codifica esa cadena gracias a la función `digito`, que ofrece el valor numérico de cada uno de los dígitos. Habrá que multiplicar por 10 cada vez que se incorpore un nuevo dígito en la cifra codificada: ya se resolvió también (cfr. Ejercicio 12.8., Código 12.18.) este algoritmo.

17.7. ***Necesitamos una función que nos diga si, en un determinado array de tipo*** `long`***, está almacenado un valor concreto y, en tal caso, que nos diga en qué posición del array aparece por primera vez.***

Por ejemplo, si tenemos el array

```
long a[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
```

Y queremos saber si está el valor 8, la función debería indicarnos como salida el valor 3.

El prototipo de la función será

```
short IndiceArray(long*, short, long);
```

Los parámetros son (1) la dirección del array, (2) la dimensión del array, y (3) el valor buscado. Devuelve el índice donde se ubica la primera ocurrencia del valor buscado, o el valor -1

Código 17.13. Posible solución al Ejercicio propuesto 17.7.

```
short IndiceArray(long* a , short d, long el)
{
      int i;
      for(i = 0 ; i < d ; i++)
            if(a[i] == el) return i;
      return -1;
}
```

CAPÍTULO 18

ASIGNACIÓN DINÁMICA DE MEMORIA.

Como ya se ha visto en el Capítulo 11, el Estándar C99 permite crear arrays con un tamaño que se establece en tiempo de ejecución.

Pero sigue habiendo comportamientos que convendría resolver. Cabría poder realizar las dos siguientes operaciones: (1) cambiar el tamaño del array a medida que se van necesitando más elementos, o a medida que ya no son necesarios tantos como anteriormente; y (2) eliminar la totalidad de la memoria reservada en un array, aún en el caso de que no se haya terminado el ámbito en el que ese array fue creado.

Esa dos operaciones no son posibles con la memoria llamada estática, que es la parte de la memoria donde se alojan las variables de un array, pero sí se pueden realizar en la llamada memoria dinámica. Disponemos de algunas funciones que nos permiten reservar y liberar esa memoria dinámica en tiempo de ejecución.

Memoria estática y memoria dinámica.

Disponemos de zonas de memoria diferenciadas para la reserva de espacio para estructuras estáticas de datos (variables y arrays) y para estructuras dinámicas. Ambas zonas de memoria (la estática en el stack o pila; la dinámica en el montículo) ya han sido presentadas en el Capítulo 13.

La memoria dinámica añade una propiedad muy importante respecto a la memoria estática hasta ahora conocida: la posibilidad de que su tamaño varíe durante la ejecución del programa. Pero hay una segunda ventaja con la memoria dinámica: la pila de la memoria estática suele tener un tamaño bastante limitado, mientras que el heap o montículo de la dinámica abarca gran parte de la memoria RAM, y su límite queda marcado por el Sistema Operativo.

También tiene la memoria dinámica algunos inconvenientes. Señalamos aquí dos: (1) su uso es más complicado y requiere más código que el empleo de la memoria estática. (2) Su rendimiento es sensiblemente menor.

La biblioteca estándar de C, a través del archivo de cabecera `stdlib.h`, ofrece cuatro funciones para el manejo de la memoria de forma dinámica: las funciones `malloc`, `calloc`, `realloc` y `free`. Vamos a verlas en los siguientes epígrafes de este capítulo.

Función `malloc`.

El prototipo de la función `malloc` es el siguiente:

```
void *malloc(size_t size);
```

Donde el tipo de dato `size_t` lo consideraremos ahora nosotros simplemente como un tipo de dato **`long`**.

La función reserva tantos bytes consecutivos como indique el valor de la variable `size` y devuelve la dirección del primero de esos bytes. Esa

dirección debe ser recogida por una variable puntero: si no se recoge, tendremos la memoria reservada pero no podremos acceder a ella porque no sabremos dónde ha quedado hecha esa reserva.

Como se ve, la dirección se devuelve con el tipo de dato **void***. La función malloc reserva tantos bytes de espacio de memoria como indique su parámetro, pero desconoce el tipo de dato de la información para la que va a reservar esa memoria. En la asignación al puntero que debe recoger la dirección de la primera posición de memoria reservada, se debe indicar, mediante el operador forzar tipo, el tipo de dato.

Veamos un ejemplo:

```
long dim;
float *vector;
printf("Dimensión de su vector ...");    scanf(" %ld",&dim);
vector = (float*)malloc(4 * dim);
```

En el ejemplo, hemos reservado tantos bytes como hace falta para un array tipo **float** de la dimensión indicada por el usuario. Como cada **float** ocupa cuatro bytes hemos multiplicado la dimensión por cuatro. Como se ve, en la asignación, a la dirección de memoria que devuelve la función malloc, le hemos indicado que deberá comportarse como dirección de **float**. De ese tipo es además el puntero que la recoge. A partir de este momento, desde el puntero vector podemos manejarnos por el array creado en tiempo de ejecución.

También se puede recorrer el array con índices, como si de un vector normal se tratase: la variable vector[i] es la misma que la variable *(vector + i).

Es responsabilidad del programador reservar un número de bytes que sea múltiplo del tamaño del tipo de dato para el que hacemos la reserva. Si, por ejemplo, en una reserva para variables **float**, el número de bytes no es múltiplo de 4, el compilador no interrumpirá su trabajo, y generará el ejecutable; pero existe el peligro, en un momento dado, de incurrir en una violación de memoria.

De forma habitual al invocar a la función `malloc` se hace utilizando el operador **`sizeof`**. La sentencia anterior en la que reservábamos espacio para nuestro vector de tipo **`float`** quedaría mejor de la siguiente forma:

```
vector = (float*)malloc(sizeof(float) * dim);
```

Y una última observación sobre la reserva de la memoria. La función `malloc` busca un espacio de memoria heap de la longitud que se le indica en el argumento. Gracias a esta asignación de memoria se podrá trabajar con una serie de valores, y realizar cálculos necesarios para nuestra aplicación. Pero… ¿y si no hay en la memoria un espacio suficiente de bytes consecutivos, libres y disponibles para satisfacer la demanda? En ese caso, no podríamos realizar ninguna de las tareas que el programa tiene determinadas, simplemente porque no tendríamos memoria.

Cuando la función `malloc` lo logra satisfacer la demanda, devuelve el puntero nulo. Es importante, siempre que se crea un espacio de memoria con esta función, y antes de comenzar a hacer uso de ese espacio, verificar que sí se ha logrado hacer buena reserva. En caso contrario, habitualmente lo que habrá que hacer es abortar la ejecución del programa, porque sin memoria, no hay datos ni capacidad de operar con ellos.

Esta verificación se realiza de la siguiente manera:

```
vector = (float*)malloc(dimension * sizeof(float));
if(vector == NULL)
{
      printf("\nNo hay memoria disponible.");
      printf("\nEl programa va a terminar.");
      printf("\nPulse cualquier tecla ... ");
      exit(1);
}
```

Y así, nunca trabajaremos con un puntero cuya dirección es nula. Si no hiciéramos esa verificación, en cuanto se echase mano del puntero `vector` para recorrer nuestra memoria inexistente, el programa

abortaría inmediatamente. La función `exit` termina la ejecución del programa; quedará presentada en el siguiente capítulo sobre funciones; está definida en el archivo `stdlib.h`. Es mejor tomar nosotros la iniciativa, mediante la función `exit`, y decidir nosotros el modo de terminación del programa en lugar de que lo decida un error fatal de ejecución.

Función `calloc`.

Existen otras funciones muy similares de reserva de memoria dinámica. Por ejemplo, la función `calloc`, que tiene el siguiente prototipo:

```
void *calloc(size_t nitems, size_t size);
```

Que recibe como parámetros el número de elementos que se van a reservar y el número de bytes que ocupa cada elemento. La sentencia anterior

```
vector = (float*)malloc(dimension * sizeof(float));
```

ahora, con la función `calloc`, quedaría:

```
vector = (float*)calloc(dimension , sizeof(float));
```

Como se ve, en sustancia, ambas funciones tienen un comportamiento muy similar. También en este caso, obviamente, será conveniente hacer siempre la verificación de que la memoria ha sido felizmente reservada. Lo mejor es acudir a las ayudas de los compiladores para hacer buen uso de las especificaciones de cada función.

Función `realloc`.

Una última función de asignación dinámica de la memoria es la función `realloc`. Su prototipo es el siguiente:

```
void *realloc(void *block, size_t size);
```

Esta función sirve para reasignar una zona de memoria sobre un puntero. El primer parámetro es el del puntero sobre el que se va a hacer el realojo; el segundo parámetro recoge el nuevo tamaño que tendrá la zona de memoria reservada.

Si el puntero `block` ya tiene memoria asignada, mediante una función `malloc` o `calloc`, o reasignada por otra llamada anterior `realloc`, entonces la función varía el tamaño de la posición de memoria al nuevo tamaño que indica la variable `size`. Si el tamaño es menor que el que había, simplemente deja liberada para nuevos usos la memoria de más que antes disponíamos y de la que hemos decidido prescindir. Si el nuevo tamaño es mayor, entonces procura prolongar ese espacio reservado con los bytes siguientes; si eso no es posible, entonces busca en la memoria un espacio libre del tamaño indicado por `size`, y copia los valores asignados en el tramo de memoria anterior en la nueva dirección. La función devuelve la dirección donde queda ubicada toda la memoria reservada. Es importante recoger esa dirección de memoria que devuelve la función `realloc`, porque en su ejecución, la función puede cambiar la ubicación del vector. La llamada podría ser así:

```
vector = (float*)realloc(vector, new_dim * sizeof(float));
```

Si el puntero `block` tiene el valor nulo, entonces `realloc` funciona de la misma forma que la función `malloc`.

Si el valor de la variable `size` es cero, entonces la función `realloc` libera el puntero `block`, que queda nulo. En ese caso, el comportamiento de la función `realloc` es semejante al de la función `free` que vemos a continuación.

Función `free`.

Esta función viene también definida en la biblioteca `stdlib.h`. Su cometido es liberar la memoria que ha sido reservada mediante la función `malloc`, o `calloc`, o `realloc`. Su sintaxis es la siguiente:

```
void free(void *block);
```

Donde `block` es un puntero que tiene asignada la dirección de memoria de cualquier bloque de memoria que haya sido reservada previamente mediante una función de memoria dinámica, como por ejemplo la función `malloc` ya presentada y vista en el epígrafe anterior.

La memoria que reserva el compilador ante una declaración de un vector se ubica en la zona de variables de la memoria. La memoria que se reserva mediante la función `malloc` queda ubicada en otro espacio de memoria distinto, que se llama habitualmente heap. En ningún caso se puede liberar, mediante la función `free`, memoria que haya sido creada estática, es decir, vectores declarados como tales en el programa y que reservan la memoria en tiempo de ejecución. Es esa memoria del heap la que libera la función `free`. Si se aplica esa función sobre memoria estática se produce un error en tiempo de compilación.

Matrices en memoria dinámica.

Para crear matrices, podemos trabajar de dos maneras diferentes. La primera es crear una estructura similar a la indicada en el Cuadro 16.1. Presentamos un ejemplo (Código 18.1.) que genera una matriz de tipo **`float`**. El programa solicita al usuario las dimensiones de la matriz (el número de filas y el número de columnas). A continuación la aplicación reserva un array de tantos punteros como filas tenga la matriz; y sobre cada uno de esos punteros hace una reserva de tantos elementos **`float`** como columnas se deseen en la matriz.

Luego, para ver cómo se puede hacer uso de esa matriz creada mediante memoria dinámica, solicitamos al usuario que dé valor a cada elemento de la matriz y, finalmente, mostramos por pantalla todos los elementos de la matriz.

Por último, se realiza la liberación de la memoria. Para ello primero habrá que liberar la memoria asignada a cada uno de los vectores

creados en memoria dinámica sobre el puntero p, y posteriormente liberar al puntero p del array de punteros.

Código 18.1.

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

int main(void)
{
      float **p;
      short f, c;
      short i, j;

// Introducir las dimensiones ...
      printf("Filas del vector ... ");    scanf(" %hd",&f);
      printf("Columnas del vector  ");    scanf(" %hd",&c);

// Creación de las filas ...
      p = (float**)malloc(f * sizeof(float*));

      if(p == NULL)
      {
            printf("Memoria insuficiente.\n");
            getchar();
            exit(0);
      }

// Creación de las columnas ...
      for( i = 0 ; i < f ; i++)
      {
            *(p + i) = (float*)malloc(c * sizeof(float));

            if(*(p + i) == NULL)
            {
                  printf("Memoria insuficiente.\n");
                  getchar();
                  exit(0);
            }
      }
```

Código 18.1. (Cont.).

```
// Asignación de valores ...
      for(i = 0 ; i < f ; i++)
            for(j = 0 ; j < c ; j++)
            {
                  printf("matriz[%2hd][%2hd] = " , i, j);
                  scanf(" %f" , *(p + i) + j);
            }
// Mostrar la matriz por pantalla ...
      for(i = 0 ; i < f ; i++)
      {
            printf("\n");
            for(j = 0 ; j < c ; j++)
                  printf("%6.2f\t" , *(*(p + i) + j));
      }
// Liberar la memoria ...
      for(i = 0 ; i < f ; i++)
            free(*(p + i));
      free(p);
      return 0;
}
```

Hemos trabajado con operatoria de punteros. Ya se explicó que hablar de `*(*(p + i) + j)` es lo mismo que hablar de `p[i][j]`. Y que hablar de `(*(p + i) + j)` es hablar de `&p[i][j]`.

Una última observación a este código presentado: el puntero `p` es (debe ser así) puntero a puntero. Y, efectivamente, él apunta a un array de punteros que, a su vez, apuntan a un array de elementos **`float`**.

Decíamos que había dos formas de crear una matriz por asignación dinámica de memoria. La segunda es crear un solo array, de longitud igual al producto de filas por columnas. Y si la matriz tiene `f` filas y `c` columnas, considerar los `f` primeros elementos del vector como la primera fila de la matriz; y los segundos `f` elementos, como la segunda fila, y así, hasta llegar a la última fila. El código de esta otra forma de manejar matrices queda ejemplificado en Código 18.2.

Código 18.2.

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
      float *p;
      short f, c;
      short i, j;

// Introducir las dimensiones ...
      printf("Filas del vector ... ");   scanf(" %hd",&f);
      printf("Columnas del vector  ");   scanf(" %hd",&c);

// Creación de la matriz ...
      p = (float*)malloc(f * c * sizeof(float*));
      if(p == NULL)
      {
            printf("Memoria insuficiente.\n");
            printf("Pulse 1 tecla para terminar ... ");
            getchar();
            exit(0);
      }

// Asignación de valores ...
      for(i = 0 ; i < f ; i++)
            for(j = 0 ; j < c ; j++)
            {
                  printf("matriz[%2hd][%2hd] = " , i, j);
                  scanf(" %f" , p + (i * c + j));
            }

// Mostrar la matriz por pantalla ...
      for(i = 0 ; i < f ; i++)
      {
            printf("\n");
            for(j = 0 ; j < c ; j++)
            printf("%6.2f\t" , *(p + (i * c + j)));
      }

// Mostrar los datos como vector lineal ...
```

Código 18.2. (Cont.).

```
      printf("\n\n");
      for(i = 0 ; i < f * c ;  i++)
            printf("%6.2f\t",*(p + i));
// Liberar la memoria ...
      free(p);
      return 0;
}
```

Ahora el puntero es un simple puntero a **float**. Y jugamos con los valores de los índices para avanzar más o menos en el array. Cuando hablamos de *(p + (i * c + j)), donde p es el puntero, i es el contador de filas, j el contador de columnas, y c la variable que indica cuántas columnas hay, estamos recorriendo el array de la siguiente manera: si queremos ir, por ejemplo, a la fila 2 (i = 2) y a la columna 5 (j = 5), y suponiendo que la matriz tiene, por ejemplo, 8 columnas (c = 8) entonces, ese elemento del vector (2, 5) está ubicado en la posición 2 * 8 + 5. es decir, en la posición 21.

Con el valor i = 0 tenemos los elementos de la primera fila, situados en el vector desde su posición 0 hasta el su posición c – 1. Con el valor i = 1 tenemos los elementos de la segunda fila, situados en el vector desde su posición c hasta su posición 2 * c – 1. Y en general, la fila i se sitúa en el vector desde la posición i * c hasta la posición (i + 1) * c – 1.

De nuevo, podremos trabajar con operatoria de índices: hablar de *(p + (i * c + j)) es lo mismo que hablar del elemento del vector p[i * c + j].

CAPÍTULO 19

ALGUNOS USOS CON FUNCIONES.

En varios capítulos anteriores (14, 15 y 17) hemos visto lo básico y más fundamental sobre funciones. Con todo lo dicho allí se puede trabajar perfectamente en C, e implementar multitud de programas, con buena modularidad.

En este nuevo capítulo queremos presentar muy brevemente algunos usos más avanzados de las funciones: distintas maneras en que pueden ser invocadas. Punteros a funciones, vectores de punteros a funciones, el modo de pasar una función como argumento de otra función. Son modos de hacer sencillos, que añaden, a todo lo dicho hasta el momento, posibilidades de diseño de programas.

Otra cuestión que abordaremos en este capítulo es cómo definir aquellas funciones de las que desconozcamos *a priori* el número de parámetros que han de recibir. De hecho, nosotros ya conocemos algunas de esas funciones: la función `printf` puede ser invocada con un solo parámetro (la cadena de caracteres que no imprima ningún valor) o con tantos

como se quiera: tantos como valores queramos que se impriman en nuestra cadena de caracteres. Veremos también aquí la manera de definir funciones con estas características.

Funciones de escape.

Existen ocasiones en que lo mejor que se puede hacer es abortar la ejecución de una aplicación antes de llegar a consecuencias más desastrosas si se continuara la ejecución del programa. A veces más vale abortar misión intentando salvar la mayor parte de los muebles, que dejar que una situación irresoluble arruine la línea de ejecución y entonces se produzca la interrupción de la ejecución de una forma no controlada por nosotros. Para realizar esas operaciones de salida inmediata disponemos de dos funciones, definidas en la biblioteca `stdlib.h`: la función `exit` y la función `abort`.

La función `exit` nos permite abandonar la ejecución de un programa antes de su finalización, volviendo el control al sistema operativo. Antes de regresar ese control, realiza algunas tareas importantes: por ejemplo, si el programa tiene abiertos algunos archivos, la función los cierra antes de abandonar la ejecución de la aplicación, y así esos archivos no se corrompen. Esta función se utiliza cuando no se cumple una condición, imprescindible para la correcta ejecución del programa.

El prototipo de la función es

```
void exit(int status);
```

El parámetro `status` indica el modo en que se debe realizar la operación de finalización inmediata del programa. El valor habitual es el cero, e indica que se debe realizar una salida inmediata llamada normal.

Ahora mismo no vamos a poner ejemplos de uso de esta función. Pero más adelante, en el próximo tema, se verán ocasiones claras en que su uso es muy necesario.

La función abort es semejante. Su prototipo es:

```
void abort(void)
```

Y nos permite abandonar de forma anormal la ejecución del programa, antes de su finalización. Escribe el mensaje "*Abnormal program termination*" y devuelve el control al sistema operativo.

Punteros a funciones.

En los primeros temas de este manual hablábamos de que toda la información de un ordenador se guarda en memoria. No sólo los datos. También las instrucciones tienen su espacio de memoria donde se almacenan y pueden ser leídas. Todo programa debe ser cargado sobre la memoria principal del ordenador antes de comenzar su ejecución.

Y si una función cualquiera tiene una ubicación en la memoria, entonces podemos hablar de la dirección de memoria de esa función. Desde luego, una función ocupará más de un byte, pero se puede tomar como dirección de memoria de una función aquella donde se encuentra la entrada de esa función.

Y si tengo definida la dirección de una función… ¿No podré definir un puntero que almacene esa dirección? La respuesta es que sí, y ahora veremos cómo poder hacerlo. Por tanto, podremos usar un puntero para ejecutar una función. Ese puntero será el que también nos ha de permitir poder pasar una función como argumento de otra función.

La declaración de un puntero a una función es la declaración de una variable, que habitualmente será local. Cuando se declara un puntero a función para poder asignarle posteriormente la dirección de una o u otra función, la declaración de ese puntero a función debe tener un prototipo coincidente con las funciones a las que se desea apuntar.

Supongamos que tenemos las siguientes funciones declaradas al inicio de un programa:

```
tipo_funcion nombre_funcion_1 (tipo1, ..., tipoN);
tipo_funcion nombre_funcion_2 (tipo1, ..., tipoN);
```

Y supongamos ahora que queremos declarar, por ejemplo en la función principal, un puntero que pueda recoger la dirección de estas dos funciones. La declaración del puntero será la siguiente:

```
tipo_funcion (*puntero_a_funcion)(tipo1,…,tipoN);
```

De esta declaración podemos hacer las siguientes observaciones:

1. Los prototipos de la función y de puntero deber ser idénticos. `tipo_funcion` debe coincidir con el tipo de la función a la que va a apuntar el puntero a función. Y la lista de argumentos debe ser coincidente, tanto en los tipos de dato como en el orden.
2. Si `*puntero_a_funcion` NO viniese recogido entre paréntesis entonces no estaríamos declarando un puntero a función, sino una función normal que devuelve un tipo de dato puntero. Por eso los paréntesis no son opcionales.

Una vez tenemos declarado el puntero, el siguiente paso será siempre asignarle una dirección de memoria. En ese caso, la dirección de una función. La sintaxis para esta asignación es la siguiente:

```
puntero_a_funcion = nombre_funcion_1;
```

Donde `nombre_funcion_1` puede ser el nombre de cualquier función cuyo prototipo coincide con el del puntero.

Una observación importante: al hacer la asignación de la dirección de la función, hacemos uso del identificador de la función: no se emplea el operador `&`; tampoco se ponen los paréntesis al final del identificador de la función.

Al ejecutar `puntero_a_funcion` obtendremos un comportamiento idéntico al que tendríamos si ejecutáramos directamente la función. La sintaxis para invocar a la función desde el puntero es la siguiente:

```
resultado = (*puntero_a_funcion)(var_1, ...,var_N);
```

Y así, cuando en la función principal se escriba esta sentencia tendremos el mismo resultado que si se hubiera consignado la sentencia

```
resultado = nombre_a_funcion_1(var_1, ...,var_N);
```

Antes de ver algunos ejemplos, hacemos una última observación. El puntero función es una variable local en una función. Mientras estemos en el ámbito de esa función podremos hacer uso de ese puntero. Desde luego toda función trasciende el ámbito de cualquier otra función; pero no ocurre así con los punteros.

Veamos algún ejemplo. Hacemos un programa (Código 19.1.) que solicita al usuario dos operandos y luego si desea sumarlos, restarlos, multiplicarlos o dividirlos. Entonces muestra el resultado de la operación. Se definen cuatro funciones, para cada una de las cuatro posibles operaciones a realizar. Y un puntero a función al que se le asignará la dirección de la función que ha de realizar esa operación seleccionada.

Código 19.1.

```
#include <stdio.h>

float sum(float, float);
float res(float, float);
float pro(float, float);
float div(float, float);

int main(void)
{
      float a, b;
      unsigned char op;
      float (*f_op)(float, float);

      printf("Primer operador ... ");    scanf(" %f",&a);
      printf("Segundo operador ... ");   scanf(" %f",&b);
      printf("Operación ( + , - , * , / ) ... ");
```

Código 19.1.(Cont.).

```
      do
            op = getchar();
      while(op !='+' && op !='-' && op !='*' && op !='/');

      switch(op)
      {
      case '+':   f_op = sum;       break;
      case '-':   f_op = res;       break;
      case '*':   f_op = pro;       break;
      case '/':   f_op = div;
      }
      printf("\n%f %c %f = %f", a, op, b, (*f_op)(a, b));
      return 0;
}

float sum(float x, float y)  { return x + y; }
float res(float x, float y)  { return x - y; }
float pro(float x, float y)  { return x * y; }
float div(float x, float y)  { return y ? x / y : 0; }
```

El puntero f_op queda definido como variable local dentro de main. Dependiendo del valor de la variable op al puntero se le asignará la dirección de una de las cuatro funciones, todas ellos con idéntico prototipo, igual a su vez al prototipo del puntero.

Vectores de punteros a funciones.

Como todo puntero, un puntero a función puede formar parte de un array. Y como podemos definir arrays de todos los tipos que queramos, entonces podemos definir un array de tipo de dato punteros a funciones. Todos ellos serán del mismo tipo, y por tanto del mismo prototipo de función. La sintaxis de definición será la siguiente:

```
tipo_funcion (*ptr_funcion[dimensión])(tipo1,...,tipoN);
```

Y la asignación puede hacerse directamente en la creación del puntero, o en cualquier otro momento:

```
tipo_funcion (*ptr_funcion[n])(tipo1, ..., tipoN) =
{ funcion_1, función_2, ..., funcion_n };
```

Donde deberá haber tantos nombres de función, todas ellas del mismo tipo, como indique la dimensión del vector. Como siempre, cada una de las funciones deberá quedar declarada y definida en el programa.

El vector de funciones se emplea de forma análoga a cualquier otro vector. Se puede acceder a cada una de esas funciones mediante índices, o por operatoria de punteros.

Podemos continuar con el ejemplo del epígrafe anterior. Supongamos que la declaración del puntero queda transformada en la declaración de una array de dimensión 4:

```
float(*operacion[4])(float,float) = {sum, res, pro, div};
```

Con esto hemos declarado cuatro punteros, cada uno de ellos apuntando a cada una de las cuatro funciones definidas. A partir de ahora será lo mismo invocar a la función `sum` que invocar a la función apuntada por el primer puntero del vector.

Si incorporamos en la función `main` la declaración de una variable `i` de tipo entero, la estructura **`switch`** puede quedar ahora como sigue

```
switch(op)
{
case '+':      i = 0;      break;
case '-':      i = 1;      break;
case '*':      i = 2;      break;
case '/':      i = 3;
}
```

Y ahora la ejecución de la función será como sigue:

```
printf("\n\n%f %c %f = %f", a, op, b, (*operación[i])(a,b));
```

Funciones como argumentos.

Se trata ahora de ver cómo hemos de definir un prototipo de función para que pueda recibir a otras funciones como parámetros. Un programa que usa funciones como argumentos suele ser difícil de comprender y de depurar, pero se adquiere a cambio una gran potencia en las posibilidades de C.

La utilidad de pasar funciones como parámetro en la llamada a otra función está en que se puede hacer depender cuál sea la función a ejecutar del estado a que se haya llegado a lo largo de la ejecución. Estado que no puede prever el programador, porque dependerá de cada ejecución concreta. Y así, una función que recibe como parámetro la dirección de una función, tendrá un comportamiento u otro según reciba la dirección de una u otra de las funciones declaradas y definidas.

La sintaxis del prototipo de una función que recibe como parámetro la dirección de otra función es la habitual: primero el tipo de la función, seguido de su nombre y luego, entre paréntesis, la lista de parámetros. Y entre esos parámetros uno o algunos pueden ser punteros a funciones. El compilador sólo sabrá que nuestra función recibirá como argumento, entre otras cosas, la dirección de otra función que se ajusta al prototipo declarado como parámetro. Cuál sea esa función es cuestión que no se conocerá hasta el momento de la ejecución y de la invocación a esa función.

La forma en que se llamará a la función será la lógica de acuerdo con estos parámetros. El nombre de la función y seguidamente, entre paréntesis, todos sus parámetros en el orden correcto.

Será conveniente seguir con el ejemplo anterior (Código 19.1.), utilizando ahora una quinta función para realizar la operación y mostrar por pantalla su resultado (cfr. Código 19.2.).

Código 19.2.

```
#include <stdio.h>
#include <conio.h>
float sum(float, float);
float res(float, float);
float pro(float, float);
float div(float, float);
void mostrar(float, char, float, float(*f)(float, float));

int main(void)
{
      float a, b;
      unsigned char op;
      float (*f_op[4])(float, float) ={sum,res,pro,div};

      printf("\n1er operador: ");  scanf(" %f",&a);
      printf("2do operador ... "); scanf(" %f",&b);
      printf("Operación ... \n)");
      printf("\n\n1. Suma\n2. Resta”);
      printf(“\n3. Producto\n4. Cociente");
      printf("\n\n\tSu opción (1 , 2 , 3 , 4) ... ");
      do
            op = '0' – getche();
      while(op < 1 || op > 4 );
      mostrar(a, op, b, f_op[(short)(op - '1')]);
      return 0;
}

float sum(float x, float y)  { return x + y; }
float res(float x, float y)  { return x - y; }
float pro(float x, float y)  { return x * y; }
float div(float x, float y)  { return y ? x / y : 0; }

void mostrar(float x,char c,float y,float(*f)(float,float))
{
      if(c == '1')            c = '+';
      else if(c == '2')       c = '-';
      else if(c == '3')       c = '*';
      else                    c = '/';

      printf("\n\n%f %c %f = %f.\n", x, c, y, (*f)(x,y));
}
```

Vamos viendo poco a poco el código. Primero aparecen las declaraciones de cinco funciones: las encargadas de realizar suma, resta, producto y cociente de dos valores **float**. Y luego, una quinta función, que hemos llamado mostrar, que tiene como cuarto parámetro un puntero a función. La declaración de este parámetro es como se dijo: el tipo del puntero de función, el nombre del puntero, recogido entre paréntesis y precedido de un asterisco, y luego, también entre paréntesis, la lista de parámetros del puntero a función. Así ha quedado declarada.

Y luego comienza la función principal, main, donde viene declarado un vector de cuatro punteros a función. A cada uno de ellos le hemos asignado una de las cuatro funciones.

Y hemos recogido el código de toda la función main en un bloque do–**while**, para que se realicen tantas operaciones como se deseen. Cada vez que se indique una operación se hará una llamada a la función mostrar que recibirá como parámetro una de las cuatro direcciones de memoria de las otras cuatro funciones. La llamada es de la forma:

```
mostrar(a, op, b, f_op[(short)(op - '1')]);
```

Donde el cuarto parámetro es la dirección de la operación correspondiente. f_op[0] es la función sum; f_op[1] es la función res; f_op[2] es la función pro; y f_op[3] es la función div. El valor de op – 1 será 0 si op es el carácter '1'; será 1 si es el carácter '2'; será 2 si es el carácter '3'; y será 3 si es el carácter '4'.

Y ya estamos en la función mostrar, que simplemente tiene que ejecutar el puntero a función y mostrar el resultado por pantalla.

Ejemplo: la función qsort.

Hay ejemplos de uso de funciones pasadas como parámetros muy utilizados, como por ejemplo la función qsort, de la biblioteca stdlib.h. Esta función es muy eficaz en la ordenación de grandes cantidades de valores. Su prototipo es:

```
void qsort(void *base, size_t nelem, size_t width,
           int (*fcmp)(const void*, const void*));
```

Es una función que no devuelve valor alguno. Recibe como parámetros el puntero `base` que es quien recoge la dirección del array donde están los elementos a ordenar; `nelem`, que es un valor entero que indica la dimensión del vector pasado como primer parámetro; `width` es el tercer parámetro, que indica el tamaño que tiene cada uno de los elementos del array; y el cuarto parámetro, es una función que devuelve un valor entero y que recibe como parámetros dos direcciones de variables. La función que se pase en este puntero debe devolver un 1 si su primer parámetro apunta a un valor mayor que el segundo parámetro; el valor -1 si es al contrario; y un 0 si el valor de ambos parámetros son iguales.

Hay que explicar porqué los tipos que recoge el prototipo son siempre `void`. El motivo es porque la función `qsort` está definida para ser capaz de ordenar un array de cualquier tipo. Puede ordenar enteros, reales, letras, u otros tipos de dato mucho más complejos, que se pueden crear y que veremos en un capítulo posterior. La función no tiene en cuenta el tipo de dato: simplemente quiere saber dos cosas:

1. El tamaño del tipo de dato; y eso se le facilita a la función a través del tercer parámetro, `width`.
2. Cómo se define la ordenación: como *a priori* no se sabe el tipo de dato, tampoco puede saber la función `qsort` con qué criterio decidir qué valores del dominio del tipo de dato son mayores, o iguales, o menores. Por eso, la función `qsort` requiere que el usuario le facilite, mediante una función muy simple que debe implementar cada usuario de la función `qsort`, ese criterio de ordenación.

Actualmente el algoritmo que da soporte a la función `qsort` es el más eficaz en las técnicas de ordenación de grandes cantidades de valores.

Vamos a ver un ejemplo de uso de esta función (Código 19.3.). Vamos a hacer un programa que ordene un vector bastante extenso de valores

enteros que asignaremos de forma aleatoria. Para ello deberemos emplear también alguna función de generación de aleatorios. Pero esa es cuestión muy sencilla que aclaramos antes de mostrar el código de la función que hemos sugerido.

Código 19.3.

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

#define TAM 10
#define RANGO 1000
int ordenar(void*,void*);

int main(void)
{
      long numeros[TAM] , i;
      randomize();
      for(i = 0 ; i < TAM ; i++)
            numeros[i] = random(RANGO);
// Vamos a ordenar esos numeros ...
      qsort((void*)numeros, TAM, sizeof(long), ordenar);
// Mostramos resultados
      for(i = 0 ; i < TAM ; i++)
            printf("numeros[%4ld] = %ld\n", i, numeros[i]);
      return 0;
}

// La función de ordenación ...
int ordenar(void *a, void *b)
{
      if(*(long*)a > *(long*)b)          return 1;
      else if(*(long*)a < *(long*)b)     return -1;
      return 0;
}
```

Existe una función en stdlib.h llamada random. Esa función pretende ser un generador de números aleatorios. Es un generador bastante

malo, pero para nuestros propósitos sirve. Su prototipo es:

```
int random(int num);
```

Es una función que devuelve un entero aleatorio entre 0 y num – 1. El valor de num debe ser un valor entero.

Cuando en una función se hace uso de la función random, antes debe ejecutarse otra función previa: la función randomize. Esta función inicializa el generador de aleatorios con un valor inicial también aleatorio. Su prototipo es:

```
void randomize(void);
```

Y se ejecuta en cualquier momento del programa, pero siempre antes de la primera vez que se ejecute la función random.

Hemos definido la función ordenar con un prototipo idéntico al exigido por la función qsort. Recibe dos direcciones de memoria (nosotros queremos que sean **long**, pero eso no se le puede decir a qsort) y resuelve cómo discernir la relación mayor que, menor que, o identidad entre cualesquiera dos valores que la función recibirá como parámetros.

La función trata a las dos direcciones de memoria como de tipo de dato **void**. El puntero a **void** ni siquiera sabe qué cantidad de bytes ocupa la variable a la que apunta. Toma la dirección del byte primero de nuestra variable, y no hace más. Dentro de la función, el código ya especifica, mediante el operador forzar tipo, que la variable apuntada por esos punteros será tratada como una variable **long**. Es dentro de nuestra función donde especificamos el tipo de dato de los elementos que vamos a ordenar. Pero la función qsort van a poder usarla todos aquellos que tengan algo que ordenar, independientemente de qué sea ese "algo": porque todo el que haga uso de qsort le explicará a esa función, gracias al puntero a funciones que recibe como parámetro, el modo en que se decide quién va antes y quien va después.

Estudio de tiempos.

A veces es muy ilustrativo poder estudiar la velocidad de algunas aplicaciones que hayamos implementado en C.

En algunos programas de ejemplo de capítulos anteriores habíamos presentado un programa que ordenaba cadenas de enteros. Aquel programa, que ahora mostraremos de nuevo, estaba basado en un método de ordenación llamado método de la burbuja: consiste en ir pasando para arriba aquellos enteros menores, de forma que van quedando cada vez más abajo, o más atrás (según se quiera) los enteros mayores. Por eso se llama el método de la burbuja: porque lo liviano "sube".

Vamos a introducir una función que controla el tiempo de ejecución. Hay funciones bastante diversas para este estudio. Nosotros nos vamos ahora a centrar en una función, disponible en la biblioteca `time.h`, llamada `clock`, cuyo prototipo es:

```
clock_t clock(void);
```

Vamos a considerar por ahora que el tipo de dato `clock_t` es equivalente a tipo de dato **`long`** (de hecho así es). Esta función está recomendada para medir intervalos de tiempo. El valor que devuelve es proporcional al tiempo trascurrido desde el inicio de ejecución del programa en la que se encuentra esa función. Ese valor devuelto será mayor cuanto más tarde se ejecute esta función `clock`, que no realiza tarea alguna más que devolver el valor actualizado del contador de tiempo. Cada breve intervalo de tiempo (bastantes veces por segundo: no vamos ahora a explicar este aspecto de la función) ese contador que indica el intervalo de tiempo transcurrido desde el inicio de la ejecución del programa, se incrementa en uno.

Un modo de estudiar el tiempo trascurrido en un proceso será:

```
time_t t1, t2;
t1 = clock();
(proceso a estudiar su tiempo)
```

```
t2 = clock();
printf("Intervalo transcurrido: %ld", t2 – t1);
```

El valor que imprimirá este código será proporcional al tiempo invertido en la ejecución del proceso del que estudiamos su ejecución. Si esa ejecución es muy rápida posiblemente el resultado sea cero.

Veamos ahora dos programas de ordenación. El primero (Código 19.4.) mediante la técnica de la burbuja. Como se ve, en esta ocasión trabajamos con un vector de cien mil valores para ordenar.

Código 19.4.

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

#define TAM 100000
#define RANGO 10000

int cambiar(long*, long*);

int main(void)
{
      long numeros[TAM] , i , j;
      time_t t1, t2;

      randomize();

      for(i = 0 ; i < TAM ; i++)
            numeros[i] = random(RANGO);

// Ordenar valores...
// Método de la burbuja ...
      t1 = clock();
      for( i = 0 ; i < TAM ; i++)
            for(j = i + 1 ; j < TAM ; j++)
            if(numeros[i] > numeros[j])
                  cambiar(numeros + i, numeros + j);
      t2 = clock();
```

Código 19.4. (Cont.).

```
      printf("t2 - t1 = %ld.\n", t2 - t1);
      return 0;
}

int cambiar(long *a, long *b)
{
      *a ^= *b;
      *b ^= *a;
      *a ^= *b;
}
```

Si lo ejecuta en su ordenador, le aparecerá por pantalla (quizá tarde unos segundos: depende de lo rápido que sea su ordenador) un número. Si es 0, porque la ordenación haya resultado muy rápida, simplemente aumente el valor de TAM y vuelva a compilar y ejecutar el programa.

Código 19.5. presenta otro programa que ordena mediante la función qsort. Es similar a Código 19.3., algo modificado para hacer la comparación de tiempos. Si ejecuta este programa, obtendrá igualmente la ordenación de los elementos del vector. Pero ahora el valor que saldrá por pantalla es del orden de 500 veces más bajo.

Código 19.5.

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

#define TAM 100000
#define RANGO 10000

int ordenar(void*,void*);
```

Código 19.5.

```
int main(void)
{
      long numeros[TAM] , I , j;
      time_t t1, t2;

      randomize();

      for(i = 0 ; i < TAM ; i++)
            numeros[i] = random(RANGO);

// Vamos a ordenar esos numeros ...
// Mediante la función qsort ...
      t1 = clock();
      qsort((void*)numeros, TAM, sizeof(long), ordenar);
      t2 = clock();

      printf("t2 - t1 = %ld.", t2 - t1);
      return 0;
}

int ordenar(void *a, void *b)
{
      if(*(long*)a > * (long*)b)          return 1;
      else if(*(long*)a < *(long*)b)      return -1;
      return 0;
}
```

El algoritmo de ordenación de la burbuja es muy cómodo de implementar, y es eficaz para la ordenación de unos pocos centenares de enteros. Pero cuando hay que ordenar grandes cantidades, no es suficiente con que el procedimiento sea teóricamente válido: además debe ser eficiente. Programar no es sólo poder en un lenguaje una serie de instrucciones. Además de saber lenguajes de programación es conveniente conocer de qué algoritmos se disponen para la solución de nuestros problemas. O echar mano de soluciones ya adoptadas, como en este caso, la implementación de la función `qsort`.

Creación de MACROS.

La directiva `#define` permite la creación de macros. Una macro es un bloque de sentencias a las que se les ha asignado un identificador. Una macro es un bloque de código que se inserta allí donde aparece su identificador. Una macro no es una función, aunque muchas veces, al usar una macro, uno puede creer que está usando funciones.

Veamos un ejemplo sencillo (cfr. Código 19.6.). Vea que `cuadrado` NO es una función, aunque su invocación tenga la misma apariencia. En el código no aparece ni un prototipo con ese nombre, ni su definición. Es una macro: el código, `"x * x"`, aparecerá allí donde en nuestro programa se ponga `cuadrado(x)`.

`#define` es una directiva del compilador. Antes de compilar, se busca en todo el texto todas las veces donde venga escrita la cadena `"cuadrado(expr)"`. Y en todas ellas sustituye esa cadena por la segunda parte de la directiva `define`: en este caso, lo sustituye por la cadena `"expr * expr"`. En general, así se puede calcular el cuadrado de cualquier expresión.

En definitiva una macro es un bloque de código que se va a insertar, previamente a la compilación, en todas aquellas partes de nuestro programa donde se encuentre su identificador.

Una macro puede hacer uso de otra macro. Por ejemplo:

```
#define cuadrado(x) x * x
#define circulo(r) 3.141596 * cuadrado(r)
```

La macro `circulo` calcula la superficie de una circunferencia de radio `r`. Para realizar el cálculo, hace uso de la macro `cuadrado`, que calcula el cuadrado del radio. La definición de una macro debe preceder siempre a su uso. No se podría definir la macro `circulo` como se ha hecho si, previamente a su definición, no estuviera recogida la definición de la macro `cuadrado`.

Código 19.6.

```
#include <stdio.h>
// Definición de la macro ...
#define cuadrado(x) x * x
int main(void)
{
      short a;
      unsigned long b;
      printf("valor de a ... ");    scanf(" %hd",&a);
      printf("El cuadrado de %hd es %lu", a, cuadrado(a));
      return 0;
}
```

Funciones con un número variable de argumentos.

Hasta el momento hemos visto funciones que tienen definido un número de parámetros concreto. Y son, por tanto, funciones que al ser invocadas se les debe pasar un número concreto y determinado de parámetros.

Sin embargo no todas las funciones que hemos utilizado son realmente así de rígidas. Por ejemplo, la función `printf`, tantas veces invocada en nuestros programas, no tiene un número prefijado de parámetros:

```
printf("Aquí solo hay un parametro. ");  // Un parámetro

printf("Aquí hay %ld parámetros. ", 2);  // Dos parámetros

printf("Y ahora %ld%c", 3, '.');         // Tres parámetros
```

Vamos a ver en este epígrafe cómo lograr definir una función en la que el número de parámetros sea variable, en función de las necesidades que tenga el usuario en cada momento.

Existen una serie de macros que permiten definir una función como si tuviera una lista variable de parámetros. Esas macros, que ahora veremos, están definidas en la biblioteca `stdarg.h`.

El prototipo de las funciones con un número variable de parámetros es el siguiente:

```
tipo nombre_funcion(tipo_1,[..., tipo_N], ...);
```

Primero se recogen todos los parámetros de la función que son fijos, es decir, aquellos que siempre deberán aparecer como parámetros en la llamada a la función. Y después de los parámetros fijos y obligatorios (como veremos más adelante, toda función que admita un número variable de parámetros, al menos deberá tener un parámetro fijo) vienen tres puntos suspensivos. Esos puntos deben ir al final de la lista de argumentos conocidos, e indican que la función puede tener más argumentos, de los que no sabemos ni cuántos ni de qué tipo de dato.

La función que tiene un número indeterminado de parámetros, deberá averiguar cuáles recibe en cada llamada. La lista de parámetros deberá ser recogida por la función, que deberá deducir de esa lista cuáles son los parámetros recibidos. Para almacenar y operar con esta lista de argumentos, está definido, en la biblioteca `stdarg.h`, un nuevo tipo de dato de C, llamado `va_list` (podríamos decir que es el tipo de "lista de argumentos"). En esa biblioteca viene definido el tipo de dato y las tres macros empleadas para operar sobre objetos de tipo lista de argumentos. Este tipo de dato tendrá una forma similar a una cadena de caracteres.

Toda función con un número de argumentos variable deberá tener declarada, en su cuerpo, una variable de tipo de dato `va_list`.

```
tipo nombre_funcion (tipo_1,[..., tipo_N], ...)
{
      va_list argumentos /* lista de argumentos */
```

Lo primero que habrá que hacer con esta variable de tipo `va_list` será inicializarla con la lista de argumentos variables recibida en la llamada a la función. Para inicializar esa variable se emplea una de las tres macros definidas en la biblioteca `stdarg.h`: la macro `va_start`, que tiene la siguiente sintaxis:

```
void va_start(va_list ap, lastfix);
```

Donde ap es la variable que hemos creado como de tipo de dato va_list, y donde lastfix es el último argumento fijo de la lista de argumentos.

La macro va_start asigna a la variable ap la dirección del primer argumento variable que ha recibido la función. Necesita, para esta operación, recibir el nombre del último parámetro fijo que recibe la función como argumento. Se guarda en la variable ap la dirección del comienzo de la lista de argumentos variables. Esto obliga a que en cualquier función con número de argumentos variable exista al menos un argumento de los llamados aquí fijos, con nombre en la definición de la función. En caso contrario, sería imposible obtener la dirección del comienzo para una lista de los restantes argumentos.

Ya tenemos localizada la cadena de argumentos variables. Ahora será necesario recorrerla para extraer todos los argumentos que ha recibido la función en su actual llamada. Para eso está definida una segunda macro de stdarg.h: la macro va_arg. Esta rutina o macro extrae el siguiente argumento de la lista.

Su sintaxis es:

```
tipo va_arg(va_list ap, tipo);
```

Donde el primer argumento es, de nuevo, nuestra lista de argumentos, llamada ap, que ya ha quedado inicializada con la macro va_start. Y tipo es el tipo de dato del próximo parámetro que se espera encontrar. Esa información es necesaria: por eso, en la función printf, indicamos en el primer parámetro (la cadena que ha de ser impresa) los especificadores de formato.

La rutina va extrayendo uno tras otro los argumentos de la lista variable de argumentos. Para cada nuevo argumento se invoca de nuevo a la macro. La macro extrae de la lista ap el siguiente parámetro (que será del tipo indicado) y avanza el puntero al siguiente parámetro de la lista.

La macro devuelve el valor extraído de la lista. Para extraer todos los elementos de la lista habrá que invocar a la macro va_arg tantas veces como sea necesario. De alguna manera la función deberá detectar que ha terminado ya de leer en la lista de variables. Por ejemplo, en la función printf, se invocará a la macro va_arg tantas veces como veces haya aparecido en la primera cadena un especificador de formato: un carácter '%' no precedido del carácter '\'.

Si se ejecuta la macro va_arg menos veces que parámetros se hayan pasado en la actual invocación, la ejecución no sufre error alguno: simplemente dejarán de leerse esos argumentos. Si se ejecuta más veces que parámetros variables se hayan pasado, entones el resultado puede ser imprevisible.

Si, después de la cadena de texto que se desea imprimir, la función printf recoge más expresiones (argumentos en la llamada) que caracteres '%' ha consignado en la cadena (primer argumento de la llamada), no pasará percance alguno: simplemente habrá argumentos que no se imprimirán y ni tan siquiera serán extraídos de la lista de parámetros. Pero si hay más caracteres '%' que variables e nuestra lista variable de argumentos, entonces la función printf ejecutará la macro va_arg en busca de argumentos no existentes. En ese caso, el resultado será completamente imprevisible.

Y cuando ya se haya recorrido completa la lista de argumentos, entonces deberemos ejecutar una tercera rutina que restablece la pila de llamada a funciones. Esta macro es necesaria para permitir la finalización correcta de la función y que pueda volver el control de programa a la sentencia inmediatamente posterior a la de la llamada de la función de argumentos variables.

Su sintaxis es: **void** va_end(va_list ap);

Veamos un ejemplo (cfr. Código 19.7.). Hagamos un programa que calcule la suma de una serie de variables double que se reciben. El

primer parámetro de la función indicará cuántos valores intervienen en la suma; los demás parámetros serán esos valores. La función devolverá la suma de todos ellos.

Código 19.7.

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

double sum(long, ...);

int main(void)
{
	double S;
	S = sum(7, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0);
	printf("%f",S);

	return 0;
}

double sum(long v,...)
{
	double suma = 0;
	long i;
	va_list sumandos;
	va_start(sumandos, v);
	for(i = 0 ; i < v ; i++)
		suma += va_arg(sumandos,double);
	va_end(sumandos);
	return suma;
}
```

La función sum recibe un único parámetro fijo, que es el que indica cuántas variables más va a recibir la función. Así cuando alguien quiere sumar varios valores, lo que hace es invocar a la función sum indicándole en un primer parámetro el número de sumandos y, a continuación, esos sumandos que se ha indicado.

Después de inicializar la variable sumandos de tipo `va_list` mediante la macro `va_start`, se van sumando todos los argumentos recibidos en la variable `suma`. Cada nuevo sumando se obtiene de la cadena `sumandos` gracias a una nueva invocación de la macro `va_arg`. Tendré tantas sumas como indique el parámetro fijo recibido en la variable `v`.

Al final, y antes de la sentencia **`return`**, ejecutamos la macro que restaura la pila de direcciones de memoria (`va_end`), de forma que al finalizar la ejecución de la función el programa logrará transferir el control a la siguiente sentencia posterior a la que invocó la función de parámetros variables.

Observación: las funciones con parámetros variables presentan dificultades cuando deben cargar, mediante la macro `va_arg`, valores de tipo **`char`** y valores **`float`**. Hay problemas de promoción de variables y los resultados no son finalmente los esperados.

Argumentos de la línea de órdenes.

Ya hemos explicado que en todo programa, la única función ejecutable es la función principal: la función `main`. Un código sin función `main` no genera un programa ejecutable, porque no tiene la función de arranque.

Vemos que todas las funciones de C pueden definirse con parámetros: es decir, pueden ejecutarse con unos valores de arranque, que serán diferentes cada vez que esa función sea invocada. También se puede hacer eso con la función `main`. En ese caso, quien debe pasar los parámetros de arranque a la función principal será el usuario del programa, a través del sistema operativo, al inicio de su ejecución

En muchos sistemas operativos es posible, cuando se ejecuta un programa compilado de C, pasar parámetros a la función `main`. Esos parámetros se pasan en la **línea de comandos** que lanza la ejecución del programa. Para ello, en esa función principal se debe haber incluido los siguientes parámetros:

```
tipo main(int argc, char *argv[] )
```

Donde argc recibe el número de argumentos de la línea de comandos, y argv es un array de cadenas de caracteres donde se almacenan los argumentos de la línea de comandos.

Los usos más comunes para los argumentos pasados a la función principal son el pasar valores para la impresión, el paso de opciones de programa (muy empleado eso en UNIX, o en DOS), el paso de nombres de archivos donde acceder a información en disco o donde guardar la información generada por el programa, etc.

La función main habrá recibido tantos argumentos como diga la variable argc. El primero de esos argumentos es siempre el nombre del programa. Los demás argumentos deben ser valores esperados, de forma que la función principal sepa qué hacer con cada uno de ellos. Alguno de esos argumentos puede ser una cadena de control que indique la naturaleza de los demás parámetros, o de alguno de ellos.

Los nombres de las variables argc y argv son mera convención: cualquier identificador que se elija servirá de la misma manera.

Y un último comentario. Hasta el momento, siempre que hemos definido la función main, la hemos declarado de tipo **int**. Realmente esta función puede ser de cualquier tipo, incluso de tipo **void**. Desde luego, la sentencia **return** debe devolver un valor del tipo de la función, o ninguno si hemos declarado a la función main como de tipo **void**.

Veamos un ejemplo (cfr. Código 19.8.). Hacemos un programa que al ser invocado se le pueda facilitar una serie de datos personales (nombre, profesión, edad), y los muestre por pantalla. Si el usuario quiere introducir el nombre, debe precederlo con la cadena "-n"; si quiere introducir la edad, deberá precederla la cadena "-e"; y si quiere introducir la profesión, deberá ir precedida de la cadena "-p".

Si una vez compilado el programa (supongamos que se llama programa) lo ejecutamos con la siguiente línea de comando:

```
programa –p estudiante –e 21 –n Isabel
```

Aparecerá por pantalla la siguiente información:

```
Nombre: Isabel
Edad: 21
Profesion: estudiante
```

Código 19.8.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char*argv[])                    {
      char nombre[30];
      char edad[5];
      char profesion[30];
      nombre[0] = profesion[0] = edad[0] = '\0';
      long i;
      for(i = 0 ; i < argc ; i++)                  {
            if(strcmp(argv[i],"-n") == 0)          {
                  if(++i < argc) strcpy(nombre, argv[i]);
            }
            else if(strcmp(argv[i],"-p") == 0)     {
                  if(++i < argc) strcpy(profesion, argv[i]);
            }
            else if(strcmp(argv[i],"-e") == 0)     {
                  if(++i < argc) strcpy(edad, argv[i]);
            }
      }
      printf("Nombre: %s\n", nombre);
      printf("Edad: %s\n", edad);
      printf("Profesion: %s\n", profesion);

      return 0;
}
```

CAPÍTULO 20

ESTRUCTURAS ESTÁTICAS DE DATOS Y DEFINICIÓN DE TIPOS.

En uno de los primeros capítulos hablamos largamente de los tipos de dato. Decíamos que un tipo de dato determina un dominio (conjunto de valores posibles) y unos operadores definidos sobre esos valores.

Hasta el momento hemos trabajado con tipos de dato estándar en C. Pero con frecuencia hemos hecho referencia a que se pueden crear otros diversos tipos de dato, más acordes con las necesidades reales de muchos problemas concretos que se abordan con la informática.

Hemos visto, de hecho, ya diferentes tipos de dato a los que por ahora no hemos prestado atención alguna, pues aún no habíamos llegado a este capítulo: tipo de dato `size_t`, ó `time_t`: cada vez que nos los hemos encontrado hemos despejado con la sugerencia de que se considerasen, sin más, tipos de dato iguales a **`long`**.

En este tema vamos a ver cómo se pueden definir nuevos tipos de dato.

Tipos de dato enumerados.

La enumeración es el modo más simple de crear un nuevo tipo de dato. Cuando definimos un tipo de dato enumerado lo que hacemos es definir de forma explícita cada uno de los valores que formarán parte del dominio de ese nuevo tipo de dato: es una definición por extensión.

La sintaxis de creación de un nuevo tipo de dato enumerado es la siguiente:

```
enum identificador {id_1[, id_2, ..., id_N]};
```

Donde **enum** es una de las 32 palabras reservadas de C. Donde `identificador` es el nombre que va a recibir el nuevo tipo de dato. Y donde `id_1`, etc. son los diferentes identificadores de cada uno de los valores del nuevo dominio creado con el nuevo tipo de dato.

Mediante la palabra clave **enum** se logran crear tipos de dato que son subconjunto de los tipos de dato **int**. Los tipos de dato enumerados tienen como dominio un subconjunto del dominio de **int**. De hecho las variables creadas de tipo **enum** son tratadas, en todo momento, como si fuesen de tipo **int**. Lo que hace **enum** es mejorar la legibilidad del programa. Pero a la hora de operar con sus valores, se emplean todos los operadores definidos para **int**.

Veamos un ejemplo:

```
enum semaforo {verde, amarillo, rojo};
```

Al crear un tipo de dato así, acabamos de definir:

Un dominio: tres valores definidos, con los literales `verde`, `amarillo` y `rojo`. En realidad el ordenador los considera valores 0, 1 y 2.

Una ordenación intrínseca a los valores del nuevo dominio: `verde` menor que `amarillo`, y `amarillo` menor que `rojo`.

Acabamos, pues, de definir un conjunto de valores enteros ordenados, con identificadores propios y únicos.

Luego, se pueden declarar variables con la siguiente sintaxis:

```
enum identificador nombre_variable;
```

En el caso del tipo de dato semaforo, podemos definir la variable cruce:

```
enum semaforo cruce;
```

Otro ejemplo: Código 20.1., que mostrará los valores de todos los colores definidos entre blanco y negro:

```
Los colores definidos son ...
0     1     2     3     4     5     6
```

Código 20.1.

```
#include <stdio.h>
enum judo {blanco, amarillo, naranja,
           verde, azul, marron, negro};

int main(void)
{
      enum judo c;
      printf("Los colores definidos son ... \n");
      for(c = blanco ; c <= negro ; c++)
            printf("%d\t" , c);

      return 0;
}
```

Dar nombre a los tipos de dato.

A lo largo del presente capítulo veremos la forma de crear diferentes estructuras de datos que den lugar a tipos de dato nuevos. Luego, a estos tops de dato se les puede asignar un identificador o un nombre para poder hacer referencia a ellos a la hora de crear nuevas variables.

La palabra clave **typedef** permite crear nuevos nombres para los tipos de dato creados. Una vez se ha creado un tipo de dato y se ha creado el

nombre para hacer referencia a él, ya podemos usar ese identificador en la declaración de variables, como si fuese un tipo de dato estándar en C.

En sentido estricto, las sentencias **typedef** no crean nuevos tipos de dato, sino que asignan un nuevo identificador para esos tipos de dato.

La sintaxis para esa creación de identificadores es la siguiente:

```
typedef tipo nombre_tipo;
```

Así, se pueden definir los tipos de dato estándar con otros nombres, que quizá convengan por la ubicación en la que se va a hacer uso de esos valores definidos por el tipo de dato. Y así tenemos:

```
typedef unsigned long size_t;
typedef long time_t;
```

O podemos nosotros mismos reducir letras:

```
typedef unsigned long int uli;
typedef unsigned short int usi;
typedef signed short int ssi;
typedef signed long int sli;
```

O también:

```
typedef char* CADENA;
```

Y así, a partir de ahora, en todo nuestro programa, nos bastará declarar las variables enteras como uno de esos nuevos cuatro tipos. O declarar una cadena de caracteres como una variable de tipo CADENA. Es evidente que con eso no se ha creado un nuevo tipo de dato, sino simplemente un nuevo identificador para un tipo de dato ya existente.

También se puede dar nombre a los nuevos tipos de dato creados. En el ejemplo del tipo de dato **enum** llamado semaforo, podríamos hacer:

```
typedef enum {verde, amarillo, rojo} semaforo;
```

Y así, el identificador semaforo queda como identificador válido de tipo de dato en C. Luego, cuando queramos una variable de este tipo, ya no diremos `enum semaforo cruce;` Sino simplemente `semaforo cruce;`

Estructuras de datos y tipos de dato estructurados.

Comenzamos ahora a tratar de la creación de verdaderos nuevos tipos de dato. En C, además de los tipos de dato primitivos, se pueden utilizar otros tipos de dato definidos por el usuario. Son tipos de dato que llamamos **estructurados**, que se construyen mediante componentes de tipos más simples previamente definidos o tipos de dato primitivos, que se denominan elementos de tipo constituyente. Las propiedades que definen un tipo de dato estructurado son el número de componentes que lo forman (que llamaremos cardinalidad), el tipo de dato de los componentes y el modo de referenciar a cada uno de ellos.

Un ejemplo de tipo de dato estructurado ya lo hemos definido y utilizado de hecho: las matrices y los vectores. No hemos considerado esas construcciones como una creación de un nuevo tipo de dato sino como una colección ordenada y homogénea de una cantidad fija de elementos, todos ellos del mismo tipo, y referenciados uno a uno mediante índices.

Pero existe otro modo, en C, de crear un tipo de dato estructurado. Y a ese nos queremos referir cuando decimos que creamos un nuevo tipo de dato, y no solamente una colección ordenada de elementos del mismo tipo. Ese tipo de dato se llama **registro**, y está formado por yuxtaposición de elementos que contienen información relativa a una misma entidad. Por ejemplo, el tipo de dato asignatura puede tener diferentes elementos, todos ellos relativos a la entidad asignatura, y no todos ellos del mismo tipo. Y así, ese tipo de dato registro que hemos llamado asignatura tendría un elemento que llamaríamos `clave` y que podría ser de tipo **`long`**; y otro campo se llamaría `descripcion` y sería de tipo **`char`***; y un tercer elemento sería el número de `creditos` y sería de tipo **`float`**, etc. A cada elemento de un registro se le llama **campo**.

Un registro es un tipo de dato estructurado heterogéneo, donde no todos los elementos (campos) son del mismo tipo. El dominio de este tipo de dato está formado por el producto cartesiano de los diferentes

dominios de cada uno de los componentes. Y el modo de referenciar a cada campo dentro del registro es mediante el nombre que se le dé a cada campo.

En C, se dispone de una palabra reservada para la creación de registros: la palabra **`struct`**.

Estructuras de datos en C.

Una estructura de datos en C es una colección de variables, no necesariamente del mismo tipo, que se referencian con un nombre común. Lo normal será crear estructuras formadas por variables que tengan alguna relación entre sí, de forma que se logra compactar la información, agrupándola de forma cabal. Cada variable de la estructura se llama, en el lenguaje C, elementos de la estructura. Este concepto es equivalente al presentado antes al hablar de campos.

La sintaxis para la creación de estructuras presenta diversas formas. Empecemos viendo una de ellas (cfr. Código 20.2.a.).

La definición de la estructura termina, como toda sentencia de C, en un punto y coma.

Una vez se ha creado la estructura, y al igual que hacíamos con las uniones, podemos declarar variables del nuevo tipo de dato dentro de cualquier función del programa donde está definida la estructura:

```
struct nombre_estructura variable_estructura;
```

Y el modo en que accedemos a cada uno de los elementos (o campos) de la estructura (o registro) será mediante el **operador miembro**, que se escribe con el identificador punto (.):

```
variable_estructura.id_1
```

Y, por ejemplo, para introducir datos en la estructura haremos:

```
variable_estructura.id_1 = valor_1;
```

Código 20.2. Declaración de estructuras. (a) creación del tipo de dato. (b) creación de variables. (c) declaración de tipo de dato y creación de algunas variables. (d) creación de un nuevo tipo de dato.

```
struct nombre_estructura
{
      tipo_1 id_1;
      tipo_2 id_2;
      ...
      tipo_N id_N;
};
```

(a)

```
struct
{
      tipo_1 id_1;
      tipo_2 id_2;
      ...
      tipo_N id_N;
}nombre_variable;
```

(b)

```
struct nombre_estructura
{
      tipo_1 id_1;
      tipo_2 id_2;
      ...
      tipo_N id_N;
}var_1, ..., var_k;
```

(c)

```
typedef struct
{
      tipo_1 id_1;
      tipo_2 id_2;
      ...
      tipo_N id_N;
} nombre_estructura;
```

(d)

La declaración de una estructura se hace habitualmente fuera de cualquier función, puesto que el tipo de dato trasciende el ámbito de una función concreta. De todas formas, también se puede crear el tipo de dato dentro de la función, cuando ese tipo de dato no va a ser empleado más allá de esa función. En ese caso, quizá no sea necesario siquiera dar un nombre a la estructura, y se pueden crear directamente las variables que deseemos de ese tipo, con la sintaxis indicada en Código 20.2.b. Y así queda definida la variable `nombre_variable`.

Otro modo de generar la estructura y a la vez declarar las primeras variables de ese tipo, será la sintaxis de Código 20.2.c.

Y así queda definido el identificador `nombre_estructura` y quedan declaradas las variables `var_1, ..., var_k`, que serán locales o globales según se hay hecho esta declaración en uno u otro ámbito.

Lo que está claro es que si la declaración de la estructura se realiza dentro de una función, entonces únicamente dentro de su ámbito el identificador de la estructura tendrá el significado de tipo de dato, y solamente dentro de esa función se podrán utilizar variables de ese tipo estructurado.

El método más cómodo para la creación de estructuras en C es mediante la combinación de la palabra **`struct`** de la palabra **`typedef`**. La sintaxis de esa forma de creación es la indicada en Código 20.2.d.

Y así, a partir de este momento, en cualquier lugar del ámbito de esta definición del nuevo tipo de dato, podremos crear variables con la siguiente sintaxis:

```
nombre_estructura nombre_variable;
```

Veamos algún ejemplo: podemos necesitar definir un tipo de dato que podamos luego emplear para realizar operaciones en variable compleja. Esta estructura, que podríamos llamar `complejo`, tendría la siguiente forma:

```
typedef struct
{
      double real;
      double imag;
}complejo;
```

Y también podríamos definir una serie de operaciones, mediante funciones: por ejemplo, la suma, la resta y el producto de complejos. El programa completo podría ser el recogido en Código 20.3.

Así podemos ir definiendo un nuevo tipo de dato, con un dominio que es el producto cartesiano del dominio de los **`double`** consigo mismo, y con unos operadores definidos mediante funciones.

Las únicas operaciones que se pueden hacer sobre la estructura (aparte de las que podamos definir mediante funciones) son las siguientes: operador dirección (`&`), porque toda variable, también las estructuradas, tienen una dirección en la memoria; operador selección (`.`) mediante el

cual podemos acceder a cada uno de los elementos de la estructura; y operador asignación, que sólo puede de forma que los dos extremos de la asignación (tanto el Lvalue como el Rvalue) sean variables objeto del mismo tipo. Por ejemplo, se puede hacer la asignación:

```
complejo A, B;
A.real = 2;
A.imag = 3;
B = A;
```

Y así, las dos variables valen lo mismo: a real de B se le asigna el valor de real de A; y a imag de B se le asigna el valor de imag de A.

Código 20.3.

```
#include <stdio.h>
typedef struct
{
      double real;
      double imag;
}complejo;

complejo sumac(complejo, complejo);
complejo restc(complejo, complejo);
complejo prodc(complejo, complejo);
void mostrar(complejo);

int main (void)
{
      complejo A, B, C;
      printf("Introducción de datos ... \n");
      printf("Parte real de A: "); scanf(" %lf",&A.real);
      printf("Parte imag de A: "); scanf(" %lf",&A.imag);
      printf("Parte real de B: "); scanf(" %lf",&B.real);
      printf("Parte imag de B: "); scanf(" %lf",&B.imag);

// SUMA ...
      printf("\n\n");   mostrar(A);
      printf(" + ");    mostrar(B);
      C = sumac(A,B);   mostrar(C);
```

Código 20.3. (Cont.).

```
// RESTA ...
      printf("\n\n");   mostrar(A);
      printf(" - ");    mostrar(B);
      C = restc(A,B);   mostrar(C);

// PRODUCTO ...
      printf("\n\n");   mostrar(A);
      printf(" * ");    mostrar(B);
      C = prodc(A,B);   mostrar(C);
      return 0;
}

complejo sumac(complejo c1, complejo c2)
{
      c1.real += c2.real;
      c1.imag += c2.imag;
      return c1;
}

complejo restc(complejo c1, complejo c2)
{
      c1.real -= c2.real;
      c1.imag -= c2.imag;
      return c1;
}

complejo prodc(complejo c1, complejo c2)
{
      complejo S;
      S.real = c1.real + c2.real; - c1.imag * c2.imag;
      S.imag = c1.real + c2.imag + c1.imag * c2.real;
      return S;
}

void mostrar(complejo X)
{
      printf("(% .2lf%s%.2lf * i) ",
             X.real, X.imag > 0 ? " +" : " " , X.imag);
}
```

Otro ejemplo de estructura podría ser el que antes hemos iniciado, al hablar de los registros: una estructura para definir un tipo de dato que sirva para el manejo de asignaturas:

```
typedef struct
{
      long clave;
      char descripcion[50];
      float creditos;
}asignatura;
```

Vectores y punteros a estructuras.

Una vez hemos creado un nuevo tipo de dato estructurado, podemos crear vectores y matrices de este nuevo tipo de dato. Si, por ejemplo, deseamos hacer un inventario de asignaturas, será lógico que creemos un array de tantas variables `asignatura` como sea necesario.

```
asignatura curricula[100];
```

Y así, tenemos 100 variables del tipo `asignatura`, distribuidas en la memoria de forma secuencial, una después de la otra. El modo en que accederemos a cada una de las variables será, como siempre mediante la operatoria de índices: `curricula[i]`. Y si queremos acceder a algún miembro de una variable del tipo estructurado, utilizaremos de nuevo el operador de miembro: `curricula[i].descripción`.

También podemos trabajar con operatoria de punteros. Así como antes hemos hablado de `curricula[i]`, también podemos llegar a esa variable del array con la expresión `*(curricula + i)`. De nuevo, todo es igual.

Donde hay un cambio es en el operador de miembro: si trabajamos con operatoria de punteros, el **operador de miembro** ya no es el punto, sino que está formado por los caracteres "`->`". Si queremos hacer referencia al elemento o campo *descripcion* de una variable del tipo asignatura, la sintaxis será: `*(curricula + i)->descripcion`.

Y también podemos trabajar con asignación dinámica de memoria. En ese caso, se declara un puntero del tipo estructurado, y luego se le asigna la memoria reservada mediante la función `malloc`. Si creamos un array de asignaturas en memoria dinámica, un programa de gestión de esas asignaturas podría ser el recogido en Código 20.4.

Observamos que `(curr + i)` es la dirección de la posición `i`-ésima del vector `curr`. Es, pues, una dirección. Y `(curr + i)->clave` es el valor del campo `clave` de la variable que está en la posición `i`-ésima del vector `curr`. Es, pues, un valor: no es una dirección. Y `(curr + i)->descr` es la dirección de la cadena de caracteres que forma el campo `descr` de la variable que está en la posición `i`-ésima del vector `curr`. Es, pues, una dirección, porque dirección es el campo `descr`: un array de caracteres.

Código 20.4.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{
      long clave;
      char descr[50];
      float cred;
}asig;

int main(void)
{
      asig *curr;
      short n, i;

      printf("Indique nº de asignaturas de su CV ... ");
      scanf(" %hd",&n);

/* La variable n recoge el número de elementos de tipo
   asignatura que debe tener nuestro array. */
```

Código 20.4. (Cont.).

```
      curr = (asig*)malloc(n * sizeof(asig));
      if(curr == NULL)
      {
            printf("Memoria insuficiente.\n");
            printf("Pulse una tecla para terminar ... ");
            getchar();
            exit(0);
      }

      for(i = 0 ; i < n ; i++)
      {
            printf("\n\nAsignatura %hd ... \n",i  + 1);
            printf("clave ......... ");
            scanf(“ %ld",&(curr + i)->clave);
            printf("Descripcion ... ");
            gets((curr + i)->descr);
            printf("creditos ...... ");
            scanf(“ %f",&(curr + i)->cred);
      }

// Listado ...
      for(i = 0 ; i < n ; i++)
      {
            printf("(%10ld)\t",(curr + i)->clave);
            printf("%s\t",(curr + i)->descr);
            printf("%4.1f creditos\n",(curr + i)->cred);
      }
      return 0;
}
```

Que accedamos a la variable estructura a través de un puntero o a través de su identificador influye únicamente en el operador de miembro que vayamos a utilizar. Una vez tenemos referenciado a través de la estructura un campo o miembro concreto, éste será tratado como dirección o como valor dependiendo de que el miembro se haya declarado como puntero o como variable de dato.

Código 20.5.

```
typedef struct
{
      unsigned short dia;
      unsigned short mes;
      unsigned short anyo;
}fecha;

typedef struct
{
      unsigned long clave;
      char descripcion[50];
      double creditos;
      fecha convocatorias[3];
}asignatura;
```

Anidamiento de estructuras.

Podemos definir una estructura que tenga entre sus miembros una variable que sea también de tipo estructura (por ej., cfr. Código 20.5.).

Ahora a la estructura de datos asignatura le hemos añadido un vector de tres elementos para que pueda consignar sus fechas de exámenes en las tres convocatorias. EL ANSI C permite hasta 15 niveles de anidamiento de estructuras.

El modo de llegar a cada campo de la estructura `fecha` es, como siempre, mediante los operadores de miembro.

Tipo de dato ***unión***.

Además de las estructuras, el lenguaje C permite otra forma de creación de un nuevo tipo de dato: mediante la creación de una unión (que se define mediante la palabra clave en C **`union`**: por cierto, con ésta, acabamos de hacer referencia en este manual a la última de las 32 palabras del léxico del lenguaje C).

Una unión es una posición de memoria compartida por dos o más variables diferentes, y en general de distinto tipo. Es una región de memoria que, a lo largo del tiempo, puede contener objetos de diversos tipos. Una unión permite almacenar tipos de dato diferentes en el mismo espacio de memoria. Como las estructuras, las uniones también tienen miembros; pero a diferencia de las estructuras, donde la memoria que ocupan es igual a la suma del tamaño de cada uno de sus campos, la memoria que emplea una variable de tipo unión es la necesaria para el miembro de mayor tamaño dentro de la unión. La unión almacena únicamente uno de los valores definidos en sus miembros.

La sintaxis para la creación de una unión es muy semejante a la empleada para la creación de una estructura. Puede verla en Código 20.6.

Código 20.6. Declaración de un tipo de dato **union**.

```
typedef union
{
      tipo_1 id_1;
      tipo_2 id_2;
      ...
      tipo_N id_N;
} nombre_union;
```

O en cualquiera otra de las formas que hemos visto para la creación de estructuras.

Es responsabilidad del programador mantener la coherencia en el uso de esta variable: si la última vez que se asignó un valor a la unión fue sobre un miembro de un determinado tipo, luego, al acceder a la información de la unión, debe hacerse con referencia a un miembro de un tipo de dato adecuado y coherente con el último que se empleó. No

tendría sentido almacenar un dato de tipo **float** de uno de los campos de la unión y luego querer leerlo a través de un campo de tipo **char**. El resultado de una operación de este estilo es imprevisible.

El tamaño de la estructura es la suma del tamaño de sus miembros. El tamaño de la unión es el tamaño del mayor de sus miembros. En la estructura se tienen espacios disjuntos para cada miembro. No así en la unión.

CAPÍTULO 21

GESTIÓN DE ARCHIVOS.

Hasta el momento, toda la información (datos) que hemos sido capaces de gestionar, la hemos tomado de dos únicas fuentes: o eran datos del programa, o eran datos que introducía el usuario desde el teclado. Y hasta el momento, siempre que un programa ha obtenido un resultado, lo único que hemos hecho ha sido mostrarlo en pantalla.

Y, desde luego, sería muy interesante poder almacenar la información generada por un programa, de forma que esa información pudiera luego ser consultada por otro programa, o por el mismo u otro usuario. O sería muy útil que la información que un usuario va introduciendo por consola quedase almacenada para sucesivas ejecuciones del programa o para posibles manipulaciones de esa información.

En definitiva, sería muy conveniente poder almacenar en algún soporte informático (v.gr., en el disco del ordenador) esa información, y acceder luego a ese soporte para volver a tomar esa información, y así actualizarla, o añadir o eliminar todo o parte de ella.

Y eso es lo que vamos a ver en este tema: la gestión de archivos. Comenzaremos con una breve presentación de carácter teórico sobre los archivos y pasaremos a ver después el modo en que podemos emplear los distintos formatos de archivo.

Tipos de dato con persistencia.

Entendemos por tipo de dato con persistencia, o **archivo**, o **fichero** aquel cuyo tiempo de vida no está ligado al de ejecución del programa que lo crea o lo maneja. Es decir, se trata de una estructura de datos externa al programa, que lo trasciende. Un archivo existe desde que un programa lo crea y mientras que no sea destruido por este u otro programa.

Un archivo está compuesto por registros homogéneos que llamamos **registros de archivo**. La información de cada registro viene recogida mediante **campos**.

Es posible crear ese tipo de dato con persistencia porque esa información queda almacenada sobre una memoria externa. Los archivos se crean sobre dispositivos de memoria masiva. El límite de tamaño de un archivo viene condicionado únicamente por el límite de los dispositivos físicos que lo albergan.

Los programas trabajan con datos que residen en la memoria principal del ordenador. Para que un programa manipule los datos almacenados en un archivo y, por tanto, en un dispositivo de memoria masiva, esos datos deben ser enviados desde esa memoria externa a la memoria principal mediante un proceso de **extracción**. Y de forma similar, cuando los datos que manipula un programa deben ser concatenados con los del archivo se utiliza el proceso de **grabación**.

De hecho, los archivos se conciben como estructuras que gozan de las siguientes características:

1. Capaces de contener grandes cantidades de información.
2. Capaces de y sobrevivir a los procesos que lo generan y utilizan.
3. Capaces de ser accedidos desde diferentes procesos o programas.

Desde el punto de vista físico, o del hardware, un archivo tiene una dirección física: en el disco toda la información se guarda (grabación) o se lee (extracción) en bloques unidades de asignación o «clusters» referenciados por un nombre de unidad o disco, la superficie a la que se accede, la pista y el sector: todos estos elementos caracterizan la dirección física del archivo y de sus elementos. Habitualmente, sin embargo, el sistema operativo simplifica mucho esos accesos al archivo, y el programador puede trabajar con un concepto simplificado de **archivo** o **fichero**: cadena de bytes consecutivos terminada por un carácter especial llamado **EOF** ("*End Of File*"); ese carácter especial (EOF) indica que no existen más bytes de información más allá de él.

Este segundo concepto de archivo permite al usuario trabajar con datos persistentes sin tener que estar pendiente de los problemas físicos de almacenamiento. El sistema operativo posibilita al programador trabajar con archivos de una forma sencilla. El sistema operativo hace de interfaz entre el disco y el usuario y sus programas.

1. Cada vez que accede a un dispositivo de memoria masiva para leer o para grabar, el sistema operativo transporta, desde o hasta la memoria principal, una cantidad fija de información, que se llama **bloque** o **registro físico** y que depende de las características físicas del citado dispositivo. En un bloque o registro físico puede haber varios registros de archivo, o puede que un registro de archivo ocupe varios bloques. Cuantos más registros de archivo quepan en cada bloque menor será el número de accesos necesarios al dispositivo de almacenamiento físico para procesar toda la información del archivo.
2. El sistema operativo también realiza la necesaria transformación de las direcciones: porque una es la posición real o efectiva donde se

encuentra el registro dentro del soporte de información (dirección física o **dirección hardware**) y otra distinta es la posición relativa que ocupa el registro en nuestro archivo, tal y como es visto este archivo por el programa que lo manipula (**dirección lógica** o simplemente **dirección**).

3. **Un archivo es una estructura de datos externa al programa**. Nuestros programas acceden a los archivos para leer, modificar, añadir, o eliminar registros. El proceso de lectura o de escritura también lo gobierna el sistema operativo. Al leer un archivo desde un programa, se transfiere la información, de bloque en bloque, desde el archivo hacia una zona reservada de la memoria principal llamada **buffer**, y que está asociada a las operaciones de entrada y salida de archivo. También se actúa a través del buffer en las operaciones de escritura sobre el archivo.

Archivos y sus operaciones.

Antes de abordar cómo se pueden manejar los archivos en C, será conveniente hacer una breve presentación sobre los archivos con los que vamos a trabajar: distintos modos en que se pueden organizar, y qué operaciones se pueden hacer con ellos en función de su modo de organización.

Hay diferentes modos de estructurar o de organizar un archivo. Las características del archivo y las operaciones que con él se vayan a poder realizar dependen en gran medida de qué modo de organización se adopte. Las dos principales formas de organización que vamos a ver en este manual son:

1. **Secuencial**. Los registros se encuentran en un orden secuencial, de forma consecutiva. Los registros deben ser leídos, necesariamente, según ese orden secuencial. Es posible leer o escribir un cierto número de datos comenzando siempre desde el principio del archivo.

También es posible añadir datos a partir del final del archivo. El acceso secuencial es una forma de acceso sistemático a los datos poco eficiente si se quiere encontrar un elemento particular.

2. **Indexado**. Se dispone de un índice para obtener la ubicación de cada registro. Eso permite localizar cualquier registro del archivo sin tener que leer todos los que le preceden.

La decisión sobre cuál de las dos formas de organización tomar dependerá del uso que se dé al archivo.

Para poder trabajar con **archivos secuenciales**, se debe previamente asignar un nombre o identificador a una dirección de la memoria externa (a la que hemos llamado antes dirección hardware). Al crear ese identificador se define un indicador de posición de archivo que se coloca en esa dirección inicial. Al iniciar el trabajo con un archivo, el indicador se coloca en el primer elemento del archivo que coincide con la dirección hardware del archivo.

Para extraer un registro del archivo, el indicador debe previamente estar ubicado sobre él; y después de que ese elemento es leído o extraído, el indicador se desplaza al siguiente registro de la secuencia.

Para añadir nuevos registros primero es necesario que el indicador se posicione o apunte al final del archivo. Conforme el archivo va creciendo de tamaño, a cada nuevo registro se le debe asignar nuevo espacio en esa memoria externa.

Y si el archivo está realizando acceso de lectura, entonces no permite el de escritura; y al revés: no se pueden utilizar los dos modos de acceso (lectura y escritura) de forma simultánea.

Las operaciones que se pueden aplicar sobre un archivo secuencial son:

1. **Creación** de un nuevo archivo, que será una secuencia vacía: ().
2. **Adición** de registros mediante buffer. La adición almacenar un registro nuevo concatenado con la secuencia actual. El archivo pasa

a ser la secuencia (secuencia inicial + buffer). La información en un archivo secuencial solo es posible añadirla al final del archivo.

3. **Inicialización** para comenzar luego el proceso de extracción. Con esta operación se coloca el indicador sobre el primer elemento de la secuencia, dispuesto así para comenzar la lectura de registros. El archivo tiene entonces la siguiente estructura: Izquierda = (); Derecha = (secuencia); Buffer = primer elemento de (secuencia).
4. **Extracción** o lectura de registros. Esta operación coloca el indicador sobre el primer elemento o registro de la parte derecha del archivo y concatena luego el primer elemento de la parte derecha al final de la parte izquierda. Y eso de forma secuencial: para leer el registro n es preciso leer todos los registros previos del archivo, desde el 1 hasta el n–1. Durante el proceso de extracción hay que verificar, antes de cada nueva lectura, que no se ha llegado todavía al final del archivo y que, por tanto, la parte derecha aún no es la secuencia vacía.

Y no hay más operaciones. Es decir, no se puede definir ni la operación inserción de registro, ni la operación modificación de registro, ni la operación borrado de registro. Al menos diremos que no se realizan fácilmente. La operación de inserción se puede realizar creando de hecho un nuevo archivo. La modificación se podrá hacer si al realizar la modificación no se aumenta la longitud del registro. Y el borrado no es posible y, por tanto, en los archivos secuenciales se define el borrado lógico: marcar el registro de tal forma que esa marca se interprete como elemento borrado.

Archivos de texto y binarios.

Decíamos antes que un archivo es un conjunto de bytes secuenciales, terminados por el carácter especial EOF.

Si nuestro archivo es de texto, esos bytes serán interpretados como caracteres. Toda la información que se puede guardar en un archivo de

texto son caracteres. Esa información podrá por tanto ser visualizada por un editor de texto.

Si se desean almacenar los datos de una forma más eficiente, se puede trabajar con archivos binarios. Los números, por ejemplo, no se almacenan como cadenas de caracteres, sino según la codificación interna que use el ordenador. Esos archivos binarios no pueden visualizarse mediante un editor de texto.

Si lo que se desea es que nuestro archivo almacene una información generada por nuestro programa y que luego esa información pueda ser, por ejemplo, editada, entonces se deberá trabajar con ficheros de caracteres o de texto. Si lo que se desea es almacenar una información que pueda luego ser procesada por el mismo u otro programa, entonces es mejor trabajar con ficheros binarios.

Tratamiento de archivos en el lenguaje C.

Todas las operaciones de entrada y salida están definidas mediante funciones de biblioteca estándar. Para trabajar con archivos con buffer, las funciones están recogidos en `stdio.h`. Para trabajar en entrada y salida de archivos sin buffer están las funciones definidas en `io.h`.

Todas las funciones de `stdio.h` de acceso a archivo trabajan mediante una interfaz que está localizada por un puntero. Al crear un archivo, o al trabajar con él, deben seguirse las normas que dicta el sistema operativo. De trabajar así se encargan las funciones ya definidas, y esa gestión es transparente para el programador.

Esa interfaz permite que el trabajo de acceso al archivo sea independiente del dispositivo final físico donde se realizan las operaciones de entrada o salida. Una vez el archivo ha quedado abierto, se puede intercambiar información entre ese archivo y el programa. El modo en que la interfaz gestiona y realiza ese tráfico es algo que no afecta para nada a la programación.

Al abrir, mediante una función, un archivo que se desee usar, se indica, mediante un nombre, a qué archivo se quiere acceder; y esa función de apertura devuelve al programa una dirección que deberá emplearse en las operaciones que se realicen con ese archivo desde el programa. Esa dirección se recoge en un puntero, llamado **puntero de archivo**. Es un puntero a una estructura que mantiene información sobre el archivo: la dirección del buffer, el código de la operación que se va a realizar, etc. De nuevo el programador no se debe preocupar de esos detalles: simplemente debe declarar en su programa un puntero a archivo, como ya veremos más adelante.

El modo en que las funciones estándar de ANSI C gestionan todo el acceso a disco es algo transparente al programador. Cómo trabaja realmente el sistema operativo con el archivo sigue siendo algo que no afecta al programador. Pero es necesario que de la misma manera que una función de ANSI C ha negociado con el sistema operativo la apertura del archivo y ha facilitado al programador una dirección de memoria, también sea una función de ANSI C quien cierre al final del proceso los archivos abiertos, de forma también transparente para el programador. Si se interrumpe inesperadamente la ejecución de un programa, o éste termina sin haber cerrado los archivos que tiene abiertos, se puede sufrir un daño irreparable sobre esos archivos, y perderlos o perder parte de su información.

También es transparente al programador el modo en que se accede de hecho a la información del archivo. El programa no accede nunca al archivo físico, sino que actúa siempre y únicamente sobre la memoria intermedia o buffer, que es el lugar de almacenamiento temporal de datos. Únicamente se almacenan los datos en el archivo físico cuando la información se transfiere desde el buffer hasta el disco. Y esa transferencia no necesariamente coincide con la orden de escritura o lectura que da el programador. De nuevo, por tanto, es muy importante terminar los procesos de acceso a disco de forma regular y normalizada,

pues de lo contrario, si la terminación del programa se realiza de forma anormal, es muy fácil que se pierdan al menos los datos que estaban almacenados en el buffer y que aún no habían sido, de hecho, transferidos a disco.

Archivos secuenciales con buffer.

Antes de utilizar un archivo, la primera operación, previa a cualquier otra, es la de apertura.

Ya hemos dicho que cuando abrimos un archivo, la función de apertura asignará una dirección para ese archivo. Debe por tanto crearse un puntero para recoger esa dirección.

En la biblioteca `stdio.h` está definido el tipo de dato `FILE`, que es tipo de dato puntero a archivo. Este puntero nos permite distinguir entre los diferentes ficheros abiertos en el programa. Crea la secuencia o interfaz que nos permite la transferencia de información con el archivo apuntado.

La sintaxis para la declaración de un puntero a archivo es la siguiente:

```
FILE *puntero_a_archivo;
```

Vamos ahora a ir viendo diferentes funciones definidas en `stdio.h` para la manipulación de archivos.

- **Apertura de archivo.**

  La función `fopen` abre un archivo y devuelve un puntero asociado al mismo, que puede ser utilizado para que el resto de funciones de manipulación de archivos accedan a este archivo abierto.

  Su prototipo es:

  ```
  FILE *fopen(const char*archivo, const char *modo_apertura);
  ```

  Donde `archivo` es el nombre del archivo que se desea abrir. Debe ir entre comillas dobles, como toda cadena de caracteres. El nombre debe

estar consignado de tal manera que el sistema operativo sepa identificar el archivo de qué se trata. Y donde `modo_apertura` es el modo de acceso para el que se abre el archivo. Debe ir en comillas dobles.

En Tabla 21.1. se recogen los distintos modos de apertura de un archivo secuencial con buffer. Hay muy diferentes formas de abrir un archivo. Queda claro que de todas ellas destacan dos bloques: aquellas que abren el archivo para manipular una información almacenada en binario, y otras que abren el archivo para poder manipularlo en formato texto. Ya iremos viendo ambas formas de trabajar la información a medida que vayamos presentando las distintas funciones.

`"r"`	Abre un archivo de texto para lectura. El archivo debe existir.
`"w"`	Abre un archivo de texto para escritura. Si existe ese archivo, lo borra y lo crea de nuevo. Los datos nuevos se escriben desde el principio.
`"a"`	Abre un archivo de texto para escritura. Los datos nuevos se añaden al final del archivo. Si ese archivo no existe, lo crea.
`"r+"`	Abre un archivo de texto para lectura/escritura. Los datos se escriben desde el principio. El fichero debe existir.
`"w+"`	Abre un archivo de texto para lectura/escritura. Los datos se escriben desde el principio. Si el fichero no existe, lo crea.
`"rb"`	Abre un archivo binario para lectura. El archivo debe existir.
`"wb"`	Abre un archivo binario para escritura. Si existe ese archivo, lo borra y lo crea de nuevo. Los datos nuevos se escriben desde el principio.
`"ab"`	Abre un archivo binario para escritura. Los datos nuevos se añaden al final del archivo. Si ese archivo no existe, lo crea.
`"r+b"`	Abre un archivo binario para lectura/escritura. Los datos se escriben desde el principio. El fichero debe existir.
`"w+b"`	Abre un archivo binario para lectura/escritura. Los datos se escriben desde el principio. Si el fichero no existe, lo crea.

Tabla 21.1. Modos de apertura archivos secuenciales con buffer.

La función fopen devuelve un puntero a una estructura que recoge las características del archivo abierto. Si se produce algún error en la apertura del archivo, entonces la función fopen devuelve un puntero nulo.

Ejemplos simples de esta función serían:

```
FILE *fichero;
fichero = fopen("datos.dat","w");
```

Que deja abierto el archivo datos.dat para escritura. Si ese archivo ya existía, queda eliminado y se crea otro nuevo y vacío.

El nombre del archivo puede introducirse mediante variable:

```
char nombre_archivo[80];
printf("Indique el nombre del archivo ... ");
gets(nombre_archivo);
fopen(nombre_archivo, "w");
```

Y ya hemos dicho que si la función fopen no logra abrir el archivo, entonces devuelve un puntero nulo. Es conveniente verificar siempre que el fichero ha sido realmente abierto y que no ha habido problemas:

```
FILE *archivo;
if(archivo = fopen("datos.dat", "w") == NULL)
      printf("No se puede abrir el archivo\n");
```

Dependiendo del compilador se podrán tener más o menos archivos abiertos a la vez. En todo caso, siempre se podrán tener, al menos ocho archivos abiertos simultáneamente.

- **Cierre del archivo abierto.**

  La función fclose cierra el archivo que ha sido abierto mediante fopen. Su prototipo es el siguiente:

  ```
  int fclose(FILE *nombre_archivo);
  ```

  La función devuelve el valor cero si ha cerrado el archivo correctamente. Un error en el cierre de un archivo puede ser fatal y puede generar todo tipo de problemas. El más grave de ellos es el de la pérdida parcial o total de la información del archivo.

Cuando una función termina normalmente su ejecución, cierra de forma automática todos sus archivos abiertos. De todas formas es conveniente cerrar los archivos cuando ya no se utilicen dentro de la función, y no mantenerlos abiertos en espera de que se finalice su ejecución.

- **Escritura de un carácter en un archivo.**

  Existen dos funciones definidas en `stdio.h` para escribir un carácter en el archivo. Ambas realizan la misma función y ambas se utilizan indistintamente. La duplicidad de definición es necesaria para preservar la compatibilidad con versiones antiguas de C.

  Los prototipos de ambas funciones son:

```
int putc(int c, FILE * archivo);
int fputc(int c, FILE * archivo);
```

  Donde `archivo` recoge la dirección que ha devuelto la función `fopen`. El archivo debe haber sido abierto para escritura y en formato texto. Y donde la variable `c` es el carácter que se va a escribir. Por razones históricas, ese carácter se define como un entero, pero sólo se toma en consideración su byte menos significativo. Si la operación de escritura se realiza con éxito, la función devuelve el mismo carácter escrito.

  En Código 21.1. se presenta un sencillo programa que solicita al usuario su nombre y lo guarda en un archivo llamado `nombre.dat`.

Código 21.1.

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

int main(void)
{
      char nombre[80];
      short int i;
      FILE *archivo;
```

Código 21.1. (Cont.).

```
      printf("Su nombre ... ");    gets(nombre);

      archivo = fopen("nombre.dat", "w");
      if(archivo == NULL)
      {
            printf("No se ha podido abrir el archivo.\n");
            getchar();
            exit(1);
      }
      i = 0;
      while(nombre[i] != NULL)
      {
            fputc(nombre[i],archivo);
            i++;
      }
      fclose(archivo);
      return 0;
}
```

Una vez ejecutado el programa, y si todo ha ido correctamente, se podrá abrir el archivo `nombre.dat` con un editor de texto y comprobar que realmente se ha guardado el nombre en ese archivo.

- **Lectura de un carácter desde un archivo.**

  De manera análoga a las funciones de escritura, existen también funciones de lectura de caracteres desde un archivo. De nuevo hay dos funciones equivalentes, cuyos prototipos son:

  `int fgetc(FILE *archivo);` y `int getc(FILE *archivo);`

  Que reciben como parámetro el puntero devuelto por la función `fopen` al abrir el archivo y devuelven el carácter, de nuevo como un entero. El archivo debe haber sido abierto para lectura y en formato texto. Cuando ha llegado al final del archivo, la función `fgetc`, o `getc`, devuelve una marca de fin de archivo que se codifica como `EOF`.

Código 21.2.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      char nombre[80];
      short int i;
      FILE *archivo;

      archivo = fopen("nombre.dat", "r");
      if(archivo == NULL)
      {
            printf("No se ha podido abrir el archivo.\n");
            getchar();
            exit(1);
      }
      i = 0;
      while((nombre[i++] = fgetc(archivo)) != EOF);
      /* El último elemento de la cadena ha quedado igual
         a EOF. Se cambia al carácter fin de cadena, NULL */
      nombre[--i] = NULL;
      fclose(archivo);
      printf("Su nombre ... %s", nombre);

      return 0;
}
```

En Código 21.2. se presenta un programa que lee desde el archivo nombre.dat el nombre que allí se guardó con el programa presentado en Código 21.1. Este programa mostrará por pantalla el nombre introducido por teclado en la ejecución del programa anterior.

- **Lectura y escritura de una cadena de caracteres.**

  Las funciones fputs y fgets escriben y leen, respectivamente, cadenas de caracteres sobre archivos de disco.

  Sus prototipos son:

```
int fputs(const char *s, FILE * archivo);
char *fgets(char *s, int n, FILE * archivo);
```

La función `fputs` escribe la cadena `s` en el archivo indicado por el puntero `archivo`. Si la operación ha sido correcta, devuelve un valor no negativo. El archivo debe haber sido abierto en formato texto y para escritura o para lectura, dependiendo de la función que se emplee.

La función `fgets` lee una cadena de caracteres desde archivo indicado por el puntero `archivo`. Lee los caracteres desde el inicio hasta un total de `n`, que es el valor que recibe como segundo parámetro. Si antes del carácter n-ésimo ha terminado la cadena, también termina la lectura y cierra la cadena con un carácter nulo.

En el programa que vimos para la función `fputc` podríamos eliminar la variable `i` y cambiar la estructura **`while`** por la sentencia:

```
fputs(nombre,archivo);
```

Y en el programa que vimos para la función `fgetc`, la sentencia podría quedar sencillamente:

```
fgets(nombre, 80, archivo);
```

- **Lectura y escritura formateada.**

  Las funciones `fprintf` y `fscanf` de entrada y salida de datos por disco tienen un uso semejante a las funciones `printf` y `scanf`, de entrada y salida por consola.

  Sus prototipos son:

```
int fprintf(FILE *archivo, const char *formato [,arg, ...]);
int fscanf(FILE *archivo, const char *formato [, dir, ...]);
```

  Donde `archivo` es el puntero a archivo que devuelve la función `fopen`. Los demás argumentos de estas dos funciones ya los conocemos, pues son los mismos que las funciones de entrada y salida por consola. La función `fscanf` devuelve el carácter `EOF` si ha llegado al final del

archivo. El archivo debe haber sido abierto en formato texto y para escritura o para lectura, dependiendo de la función que se emplee.

Veamos un ejemplo de estas dos funciones (cfr. Código 21.3.). Hagamos un programa que guarde en un archivo (que llamaremos `numeros.dat`) los valores que previamente se han asignado de forma aleatoria a un vector de variables **`float`**. Esos valores se almacenan dentro de una cadena de texto. Y luego, el programa vuelve a abrir el archivo para leer los datos y cargarlos en otro vector y los muestra en pantalla.

Código 21.3.

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

#define TAM 10

int main(void)
{
      float or[TAM], cp[TAM];
      short i;
      FILE *ARCH;
      char c[100];

      randomize();
      for(i = 0 ;  i < TAM ; i++)
            or[i] = (float)random(1000) / random(100);

      ARCH = fopen("numeros.dat", "w");
      if(ARCH == NULL)                    {
            printf("No se ha podido abrir el archivo.\n");
            getchar();
            exit(1);
      }

      for(i = 0 ; i < TAM ;  i++)
            fprintf(ARCH,"Valor %04hi-->%12.4f\n",i,or[i]);
      fclose(ARCH);
```

Código 21.3. (Cont.).

```
      ARCH = fopen("numeros.dat", "r");
      if(ARCH == NULL)                    {
            printf("No se ha podido abrir el archivo.\n");
            getchar();
            exit(1);
      }

      printf("Los valores guardados en el archivo son:\n");
      i = 0;
      while(fscanf(ARCH,"%s%s%s%f",c,c,c,cp + i++)!= EOF);
      for(i = 0 ; i < TAM ; i++)
            printf("Valor %04hd --> %12.4f\n",i,cp[i]);
      fclose(ARCH);
      return 0;
}
```

El archivo contiene (en una ejecución cualquiera: los valores son aleatorios) la siguiente información:

```
Valor 0000 -->       9.4667
Valor 0001 -->      30.4444
Valor 0002 -->      12.5821
Valor 0003 -->       0.2063
Valor 0004 -->      16.4545
Valor 0005 -->      28.7308
Valor 0006 -->       9.9574
Valor 0007 -->       0.1039
Valor 0008 -->      18.0000
Valor 0009 -->       4.7018
```

Hemos definido la variable c para que vaya cargando desde el archivo los tramos de cadena de caracteres que no nos interesan para la obtención, mediante la función `fscanf`, de los sucesivos valores **`float`** generados. Con esas tres lecturas de cadena la variable c va leyendo las cadenas "Valor"; la cadena de caracteres que recoge el índice i; la cadena "-->". La salida por pantalla tendrá la misma apariencia que la obtenida en el archivo.

Desde luego, con la función `fscanf` es mejor codificar bien la información del archivo, porque de lo contrario la lectura de datos desde el archivo puede llegar a hacerse muy incómoda.

- **Lectura y escritura en archivos binarios.**

  Ya hemos visto las funciones para acceder a los archivos secuenciales de tipo texto. Vamos a ver ahora las funciones de lectura y escritura en forma binaria.

  Si en todas las funciones anteriores hemos requerido que la apertura del fichero o archivo se hiciera en formato texto, ahora desde luego, para hacer uso de las funciones de escritura y lectura en archivos binarios, el archivo debe hacer sido abierto en formato binario.

  Las funciones que vamos a ver ahora permiten la lectura o escritura de cualquier tipo de dato. Los prototipos son los siguientes:

```
size_t fread(void *buffer, size_t n_bytes, size_t contador,
      FILE *archivo);

size_t fwrite(const void *buffer, size_t n_bytes,
      size_t contador, FILE *archivo);
```

  Donde `buffer` es un puntero a la región de memoria donde se van a escribir los datos leídos en el archivo, o el lugar donde están los datos que se desean escribir en el archivo. Habitualmente será la dirección de una variable. `n_bytes` es el número de bytes que ocupa cada dato que se va a leer o grabar, y `contador` indica el número de datos de ese tamaño que se van a leer o grabar. El último parámetro es el de la dirección que devuelve la función `fopen` cuando se abre el archivo.

  Ambas funciones devuelven el número de elementos escritos o leídos. Ese valor debe ser el mismo que el valor de `contador`; lo contrario indicará que ha ocurrido un error. Es habitual hacer uso del operador **`sizeof`**, para determinar así la longitud (`n_bytes`) de cada elemento a leer o escribir.

El ejemplo anterior (Código 21.3.) puede servir para mostrar ahora el uso de esas dos funciones: cfr. Código 21.4. El archivo numeros.dat será ahora de tipo binario. El programa cargará en forma binaria esos valores y luego los leerá para calcular el valor medio de todos ellos y mostrarlos por pantalla.

Código 21.4.

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

#define TAM 10

int main(void)
{
      float or[TAM], cp[TAM];
      double suma = 0;
      short i;
      FILE *ARCH;

      randomize();
      for(i = 0 ;  i < TAM ; i++)
            or[i] = (float)random(1000) / random(100);

      ARCH = fopen("numeros.dat", "wb");
      if(ARCH == NULL)                    {
            printf("No se ha podido abrir el archivo.\n");
            getchar();
            exit(1);
      }

      fwrite(or,sizeof(float),TAM,ARCH);
      fclose(ARCH);

      ARCH = fopen("numeros.dat", "rb");
      if(ARCH == NULL)                    {
            printf("No se ha podido abrir el archivo.\n");
            getchar();
            exit(1);
      }
```

Código 21.4. (Cont.).

```
      fread(cp,sizeof(float),TAM,ARCH);
      fclose(ARCH);

      for(i = 0 ; i < TAM ; i++)          {
            printf("Valor %04hd --> %12.4f\n",i,cp[i]);
            suma += *(cp + i);
      }
      printf("\n\nLa media es ... %lf", suma / TAM);

      return 0;
}
```

- **Otras funciones útiles en el acceso a archivo.**

  Función `feof`: Esta función (en realidad es una macro) determina el final de archivo. Es conveniente usarla cuando se trabaja con archivos binarios, donde se puede inducir a error y tomar como carácter `EOF` un valor entero codificado.

  Su prototipo es: `int feof(FILE *nombre_archivo);`

  que devuelve un valor diferente de 0 si en la última operación de lectura se ha detectado el valor `EOF`. en caso contrario devuelve el valor 0.

  Función `ferror`: Esta función (en realidad es una macro) determina si se ha producido un error en la última operación sobre el archivo. Su prototipo es:

  `int ferror(FILE * nombre_archivo);`

  Si el valor devuelto es diferente de cero, entonces se ha producido un error; si es igual a cero, entonces no se ha producido error alguno.

  Si deseamos hacer un programa que controle perfectamente todos los accesos a disco, entonces convendrá ejecutar esta función después de cada operación de lectura o escritura.

Función remove: Esta función elimina un archivo. El archivo será cerrado si estaba abierto y luego será eliminado. Quiere esto decir que el archivo quedará destruido, que no es lo mismo que quedarse vacío.

Su prototipo es:

```
int remove(const char *archivo);
```

Donde archivo es el nombre del archivo que se desea borrar. En ese nombre, como siempre, debe ir bien consignada la ruta completa del archivo. Un archivo así eliminado no es recuperable.

Por ejemplo, en nuestros ejemplos anteriores (Código 21.3. y .4.), después de haber hecho la transferencia de datos al vector de **float**, podríamos ya eliminar el archivo de nuestro disco. Hubiera abastado poner la sentencia:

```
remove("numeros.dat");
```

Si el archivo no ha podido ser eliminado (por denegación de permiso o porque el archivo no existe en la ruta y nombre que ha dado el programa) entonces la función devuelve el valor -1. Si la operación de eliminación del archivo ha sido correcta, entonces devuelve un cero.

En realidad, la macro remove lo único que hace es invocar a la función de borrado definida en io.h: la función unlink, cuyo prototipo es:

```
int unlink(const char *filename);
```

Y cuyo comportamiento es idéntico al explicado para la macro remove.

Entrada y salida sobre archivos de acceso aleatorio.

Disponemos de algunas funciones que permiten acceder de forma aleatoria a una u otra posición del archivo.

Ya dijimos que un archivo, desde el punto de vista del programador es simplemente un puntero a la posición del archivo (en realidad al buffer) donde va a tener lugar el próximo acceso al archivo. Cuando se abre el

archivo ese puntero recoge la dirección de la posición cero del archivo, es decir, al principio. Cada vez que el programa indica escritura de datos, el puntero termina ubicado al final del archivo.

Pero también podemos, gracias a algunas funciones definidas en io.h, hacer algunos accesos aleatorios. En realidad, el único elemento nuevo que se incorpora al hablar de acceso aleatorio es una función capaz de posicionar el puntero del archivo devuelto por la función fopen en distintas partes del fichero y poder así acceder a datos intermedios.

La función fseek puede modificar el valor de ese puntero, llevándolo hasta cualquier byte del archivo y logrando así un acceso aleatorio. Es decir, que las funciones estándares de ANSI C logran hacer accesos aleatorios únicamente mediante una función que se añade a todas las que ya hemos visto para los accesos secuenciales.

El prototipo de la función, definida en la biblioteca stdio.h es el siguiente:

```
int fseek(FILE *archivo, long despl, int modo);
```

Donde archivo es el puntero que ha devuelto la función fopen al abrir el archivo; donde despl es el desplazamiento, en bytes, a efectuar; y donde modo es el punto de referencia que se toma para efectuar el desplazamiento. Para esa definición de modo, stdio.h define tres constantes diferentes:

SEEK_SET, que es valor 0.
SEEK_CUR, que es valor 1,
SEEK_END, que es valor 2.

El modo de la función fseek puede tomar como valor cualquiera de las tres constantes. Si tiene la primera (SEEK_SET), el desplazamiento se hará a partir del inicio del fichero; si tiene la segunda (SEEK_CUR), el desplazamiento se hará a partir de la posición actual del puntero; si tiene la tercera (SEEK_END), el desplazamiento se hará a partir del final del fichero.

Código 21.5.

```
printf("Los valores guardados en el archivo son:\n");
i = 0;
while(!feof(ARCH))
{
      fseek(ARCH,16,SEEK_CUR);
      fscanf(ARCH,"%f",cp + i++);
}
```

Para la lectura del archivo que habíamos visto para ejemplificar la función `fscanf` (Código 231.3.), las sentencias de lectura quedarían mejor si se hiciera como recoge Código 21.5.

Donde hemos indicado 16 en el desplazamiento en bytes, porque 16 son los caracteres que no deseamos que se lean en cada línea.

Los desplazamientos en la función `fseek` pueden ser positivos o negativos. Desde luego, si los hacemos desde el principio lo razonable es hacerlos positivos, y si los hacemos desde el final hacerlos negativos. La función acepta cualquier desplazamiento y no produce nunca un error. Luego, si el desplazamiento ha sido erróneo, y nos hemos posicionado en medio de ninguna parte o en un byte a mitad de dato, entonces la lectura que pueda hacer la función que utilicemos será imprevisible.

Una última función que presentamos en este capítulo es la llamada `rewind`, cuyo prototipo es:

```
void rewind(FILE *nombre_archivo);
```

Que "rebobina" el archivo, devolviendo el puntero a su posición inicial, al principio del archivo.