

¡¡APRUEBA TU EXAMEN CON SCHAUM!!

C. Algoritmos, programación y estructuras de datos

Schaum

Luis Joyanes Aguilar • Andrés Castillo Sanz
Lucas Sánchez García • Ignacio Zahonero Martínez

REDUCE TU TIEMPO DE ESTUDIO

EJEMPLOS DETALLADOS, CUESTIONES DE REPASO Y PROBLEMAS DE COMPRENSIÓN

PROBLEMAS RESUELTOS EN LA WEB

EJERCICIOS PRÁCTICOS DE APRENDIZAJE DE PROGRAMACIÓN

Utilízalo para las siguientes asignaturas:

✓ INTRODUCCIÓN A LA PROGRAMACIÓN

✓ METODOLOGÍA DE LA PROGRAMACIÓN

✓ FUNDAMENTOS DE PROGRAMACIÓN

✓ PROGRAMACIÓN I

C

Algoritmos, programación y estructuras de datos

C

Algoritmos, programación y estructuras de datos

**LUIS JOYANES AGUILAR
ANDRÉS CASTILLO SANZ
LUCAS SÁNCHEZ GARCÍA
IGNACIO ZAHONERO MARTÍNEZ**

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática/Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid



**MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO**

La información contenida en este libro procede de una obra original entregada por los autores. No obstante, McGraw-Hill/Interamericana de España no garantiza la exactitud o perfección de la información publicada. Tampoco asume ningún tipo de garantía sobre los contenidos y las opiniones vertidas en dichos textos.

Este trabajo se publica con el reconocimiento expreso de que se está proporcionando una información, pero no tratando de prestar ningún tipo de servicio profesional o técnico. Los procedimientos y la información que se presentan en este libro tienen sólo la intención de servir como guía general.

McGraw-Hill ha solicitado los permisos oportunos para la realización y el desarrollo de esta obra.

C. Algoritmos, programación y estructuras de datos. Serie Schaum

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.



**McGraw-Hill/Interamericana
de de España, S. A. U.**

DERECHOS RESERVADOS © 2005, respecto a la primera edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1ª planta
Basauri, 17
28023 Aravaca (Madrid)

www.mcgraw-hill.es
universidad@mcgraw-hill.com

ISBN: 84-481-4514-3
Depósito legal: M.

Editor: Carmelo Sánchez González
Compuesto en CD-FORM, S.L.
Impreso en

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prólogo	XI
Capítulo 1 Introducción a las computadoras y a los lenguajes de programación	1
1.1 Organización física de una computadora	1
1.2 Redes	5
1.3 El software (los programas)	6
1.4 Lenguajes de programación	7
1.5 El lenguaje C: historia y características	10
Referencias bibliográficas y lecturas suplementarias	11
Ejercicios de repaso	12
Capítulo 2 Fundamentos de programación	13
2.1 Fases en la resolución de problemas	13
2.1.1 Análisis del problema	14
2.1.2 Diseño del algoritmo	14
2.1.3 Codificación de un programa	14
2.1.4 Compilación y ejecución de un programa	14
2.1.5 Verificación y depuración	15
2.1.6 Documentación y mantenimiento	15
2.2 Programación estructurada	16
2.2.1 Recursos abstractos	16
2.2.2 Diseño descendente (<i>Top Down</i>)	16
2.2.3 Estructuras de control	16
2.2.4 Teorema de la programación estructurada	16
2.3 Métodos formales de verificación de programas	16
2.4 Factores de calidad del software	17
Problemas resueltos	18
Problemas propuestos	24
Capítulo 3 El lenguaje C: elementos básicos	25
3.1 Estructura general de un programa en C	25
3.1.1 Directivas del preprocesador	25
3.1.2 Declaraciones globales	25
3.1.3 Función <code>main()</code>	26
3.1.4 Funciones definidas por el usuario	26
3.2 Los elementos de un programa C	27
3.3 Tipos de datos en C	27
3.3.1 Enteros (<code>int</code>)	27
3.3.2 Tipos de coma flotante (<code>float/double</code>)	28
3.3.3 Caracteres (<code>char</code>)	29
3.4 El tipo de dato lógico	29
3.5 Constantes	29
3.6 Variables	30
3.7 Entradas y salidas	30

Problemas resueltos.....	31
Problemas propuestos.....	35
Capítulo 4 Operadores y expresiones	37
4.1 Operadores y expresiones	37
4.2 El operador de asignación	37
4.3 Operadores aritméticos.....	38
4.4 Operadores de incrementación y decrementación	39
4.5 Operadores relacionales	39
4.6 Operadores lógicos.....	40
4.7 Operadores de manipulación de bits	40
4.7.1 Operadores de asignación adicionales	41
4.7.2 Operadores de desplazamiento de bits (>>, <<).....	41
4.8 Operador condicional	41
4.9 Operador coma ,	42
4.10 Operadores especiales (), []	42
4.11 El operador sizeof.....	42
4.12 Conversiones de tipos	42
4.13 Prioridad y asociatividad.....	43
Problemas resueltos.....	44
Problemas propuestos.....	53
Capítulo 5 Estructuras de selección: sentencias if y switch.....	55
5.1 Estructuras de control.....	55
5.2 La sentencia if con una alternativa.....	55
5.3 Sentencia if de dos alternativas: if-else.....	56
5.4 Sentencia de control switch.....	57
5.5 Expresiones condicionales: el operador ?:.....	57
5.6 Evaluación en cortocircuito de expresiones lógicas	58
Problemas resueltos.....	58
Problemas propuestos.....	69
Capítulo 6 Estructuras de control: bucles.....	71
6.1 La sentencia while	71
6.1.1 Miscelánea de control de bucles while.....	72
6.2 Repetición: el bucle for	73
6.3 Repetición: el bucle do...while	74
6.4 Comparación de bucles while, for y do-while	74
Problemas resueltos.....	75
Problemas propuestos.....	92
Capítulo 7 Funciones	95
7.1 Concepto de función	95
7.2 Estructura de una función	95
7.3 Prototipos de las funciones	96
7.4 Parámetros de una función.....	97
7.5 Funciones en línea, macros con argumentos	98
7.6 Ámbito (alcance).....	98
7.7 Clases de almacenamiento	99
7.8 Concepto y uso de funciones de biblioteca.....	100
7.9 Miscelánea de funciones	100
Problemas resueltos.....	101
Problemas propuestos.....	121

Capítulo 8 Recursividad	123
8.1 La naturaleza de la recursividad	123
8.2 Funciones recursivas	123
8.3 Recursión versus iteración	124
8.4 Recursión infinita	125
8.5 Algoritmos divide y vencerás	125
Problemas resueltos.....	125
Problemas propuestos.....	135
Capítulo 9 Arrays (listas y tablas)	137
9.1 Arrays	137
9.2 Inicialización de un array	138
9.3 Arrays de caracteres y cadenas de texto	138
9.4 Arrays multidimensionales.....	139
9.5 Utilización de arrays como parámetros.....	140
Problemas resueltos.....	140
Problemas propuestos.....	159
Capítulo 10 Algoritmos de ordenación y búsqueda	161
10.1 Ordenación	161
10.2 Ordenación por burbuja	161
10.3 Ordenación por selección.....	162
10.4 Ordenación por inserción	162
10.5 Ordenación Shell	163
10.6 Ordenación rápida (QuickSort).....	163
10.7 Búsqueda en listas: búsqueda secuencial y binaria.....	163
Problemas resueltos.....	165
Problemas propuestos.....	170
Capítulo 11 Estructuras y uniones	173
11.1 Estructuras	173
11.2 Uniones	175
11.3 Enumeraciones	176
11.4 Sinónimo de un tipo de datos: Typedef	177
11.5 Campos de bit.....	179
Problemas resueltos.....	180
Problemas propuestos.....	190
Problemas de programación de gestión	190
Capítulo 12 Punteros (apuntadores)	191
12.1 Concepto de puntero (apuntador).....	191
12.2 Punteros NULL y VOID	192
12.3 Punteros y arrays	192
12.4 Aritmética de punteros	194
12.5 Punteros como argumentos de funciones.....	195
12.6 Punteros a funciones	196
Problemas resueltos.....	197
Problemas propuestos.....	209
Capítulo 13 Asignación dinámica de memoria	211
13.1 Gestión dinámica de la memoria.....	211
13.2 Función malloc()	212
13.3 Liberación de memoria, función free()	213
13.4 Funciones calloc() y realloc()	213
Problemas resueltos.....	214

Problemas propuestos.....	223
Capítulo 14 Cadenas	225
14.1 Concepto de cadena	225
14.2 Inicialización de variables de cadena.....	226
14.3 Lectura de cadenas	226
14.4 Las funciones de <code>STRING.H</code>	227
14.5 Conversión de cadenas a números	229
Problemas resueltos.....	231
Problemas propuestos.....	240
Capítulo 15 Entrada y salida por archivos	243
15.1 Flujos.....	243
15.2 Apertura de un archivo.....	244
15.3 Funciones de lectura y escritura.....	244
15.4 Archivos binarios de C.....	246
15.5 Datos externos al programa con argumentos de <code>main()</code>	248
Problemas resueltos.....	249
Problemas propuestos.....	265
Capítulo 16 Organización de datos en un archivo.....	267
16.1 Registros.....	267
16.2 Organización de archivos	268
16.2.1 Organización secuencial	268
16.2.2 Organización directa.....	270
16.3 Archivos con direccionamiento hash	271
16.4 Archivos secuenciales indexados	272
16.5 Ordenación de archivos: ordenación externa	274
Problemas resueltos.....	277
Problemas propuestos.....	293
Capítulo 17 Tipos abstractos de datos TAD/objetos	295
17.1 Tipos de datos.....	295
17.2 Tipos abstractos de datos	296
17.3 Especificación de los TAD	298
Problemas resueltos.....	298
Problemas propuestos.....	309
Capítulo 18 Listas enlazadas	311
18.1 Fundamentos teóricos.....	311
18.2 Clasificación de las listas enlazadas	311
18.3 Operaciones en listas enlazadas	312
18.3.1 Inserción de un elemento en una lista	313
18.3.2 Eliminación de un nodo en una lista	313
18.4 Lista doblemente enlazada	314
18.4.1 Inserción de un elemento en una lista doblemente enlazada	314
18.4.2 Eliminación de un elemento en una lista doblemente enlazada.....	315
18.5 Listas circulares.....	316
Problemas resueltos.....	317
Problemas propuestos.....	344
Capítulo 19 Pilas y colas	347
19.1 Concepto de pila.....	347
19.2 Concepto de cola.....	348

Problemas resueltos.....	350
Problemas propuestos.....	366
Capítulo 20 Árboles.....	369
20.1 Árboles generales	369
20.2 Árboles binarios	370
20.3 Estructura y representación de un árbol binario	371
20.4 Árboles de expresión.....	371
20.5 Recorridos de un árbol	372
20.6 Árbol binario de búsqueda	372
20.7 Operaciones en árboles binarios de búsqueda	373
Problemas resueltos.....	374
Problemas propuestos.....	389
Apéndice A. Compilación de programas C en UNIX y LINUX.....	391
Apéndice B. Compilación de programas C en Windows	395
Apéndice C. Recursos Web de programación	399
Índice.....	405

Prólogo

Introducción

Desde que **Kernighan** y **Ritchie** escribieran en 1975 su mítico libro *Programación en C*, con el que tantos y tantos lectores y estudiantes del mundo entero hemos aprendido C y siguen aprendiendo, ¿qué ha cambiado desde entonces en el mundo de la programación? Realmente, poco y mucho. C sigue siendo el lenguaje más utilizado para aprender fundamentos y técnicas de programación tanto en la universidad como en los institutos tecnológicos y centros de formación profesional. C++ sigue reinando en el aprendizaje de la programación orientada a objetos, aunque Java ya es un gran rival. **Java** y **C#** se han erigido como lenguajes por excelencia en el mundo profesional de la programación, la ingeniería de software, la ingeniería *Web* y las telecomunicaciones.

C es un lenguaje ideal para aprender la programación de computadoras. C es un lenguaje muy compacto ya que su sintaxis es sencilla y fácil para aprender a escribir aplicaciones reales. Es también un lenguaje muy potente ya que se utiliza mucho en programación en todos los niveles, desde controladores de dispositivos y componentes de sistemas operativos hasta aplicaciones a gran escala. Existen compiladores de C para cualquier entorno de programación y de sistemas operativos, tales como **Windows, Unix, Linux, Mac**, etc., de modo que cuando usted haya aprendido C estará en condiciones de programar en cualquier contexto y entorno actual. También observará que C es una base excelente para continuar su formación en programación orientada a objetos con C++ y luego migrar a Java o C# en función del ambiente profesional en que se desenvuelva.

Todas las carreras universitarias de *Ciencias e Ingeniería*, así como los estudios de *Formación Profesional* (sobre todo en España los ciclos superiores) requieren un curso básico de algoritmos y de programación con un lenguaje potente y profesional pero que sea simple y fácil de utilizar. C es idóneo para aprender a programar directamente las técnicas algorítmicas y de programación en asignaturas tales como *Introducción a la Programación, Fundamentos de Programación o Metodología de la Programación* o con otros nombres tales como *Algoritmos, Programación I*, etc. C sigue siendo el lenguaje universal más utilizado y recomendado en planes de estudio de universidades y centros de formación de todo el mundo. Organizaciones como **ACM, IEEE**, colegios profesionales, siguen recomendando la necesidad del conocimiento en profundidad de técnicas y de lenguajes de programación estructurada con el objetivo de “acomodar” la formación del estudiante a la concepción, diseño y construcción de algoritmos y de estructuras de datos. El conocimiento profundo de algoritmos unido a técnicas fiables, rigurosas y eficientes de programación preparan al estudiante o al autodidacta para un alto rendimiento en programación y la preparación para asumir los retos de la programación orientada a objetos en una primera fase y las técnicas y métodos inherentes a ingeniería de software en otra fase más avanzada.

La mejor manera para aprender a programar una computadora es pensar y diseñar el algoritmo que resuelve el problema, codificar en un lenguaje de programación (C en nuestro caso) y depurar el programa una y otra vez hasta entender la gramática y sus reglas de sintaxis, así como la lógica del programa. Nunca mejor dicho, aprender practicando. El lenguaje C se presta a escribir, compilar, ejecutar y verificar errores. Por esta razón hemos incluido en la estructura del libro las introducciones teóricas imprescindibles con el apoyo de numerosos ejemplos, luego hemos incorporado numerosos ejercicios y problemas de programación con un análisis del problema y sus códigos fuente, y en numerosas ocasiones se presenta la salida o ejecución de los respectivos programas.

La estructura del libro en la colección Schaum

Esta edición al ha sido escrita dentro de la prestigiosa colección *Schaum* de McGraw-Hill, como un manual práctico para la enseñanza de la programación de computadoras estudiando con el lenguaje de programación C. Debido a los objetivos que tiene esta antigua colección, el enfoque es eminentemente práctico con el necesario estudio teórico que permita avanzar de modo rápido y eficaz al estudiante en su aprendizaje de la programación en C. Pensando en colección los cuatro autores hemos escrito este libro con un planteamiento eminentemente teórico-práctico como son todos los pertenecientes a esta

colección con el objetivo de ayudar a los lectores a superar sus exámenes y pruebas prácticas en sus estudios de formación profesional o universitarios, y mejorar su aprendizaje de modo simultáneo pero con unos planteamientos prácticos: analizar los problemas, escribir los códigos fuente de los programas y depurar estos programas hasta conseguir el funcionamiento correcto y adecuado.

Hemos añadido un complemento práctico de ayuda al lector. En la página oficial del libro (<http://www.mhe.es/joyanes>), encontrará el lector todos los códigos fuente incluidos en la obra y que podrá descargar de Internet. Pretendemos no solo evitar su escritura desde el teclado para que se centre en el estudio de la lógica del programa y su posterior depuración (edición, compilación, ejecución, verificación y pruebas) sino también para que pueda contrastar el avance adecuado de su aprendizaje. También en la página Web encontrará otros recursos educativos que confiamos le ayudarán a progresar de un modo eficaz y rápido.

¿Qué necesita para utilizar este libro?

C. Algoritmos, Programación y Estructura de Datos, está diseñado para enseñar métodos de escritura de programas útiles tan rápido y fácil como sea posible, aprendiendo a la par tanto la sintaxis y funcionamiento del lenguaje de programación como las técnicas de programación y los fundamentos de construcción de algoritmos básicos. El contenido se ha escrito pensando en un lector que tiene conocimientos básicos de algoritmos y de programación C, y que desea aprender a conocer de modo práctico técnicas de programación.

El libro es eminentemente práctico con la formación teórica necesaria para obtener el mayor rendimiento en su aprendizaje. Pretende que el lector utilice el libro para aprender de un modo práctico las técnicas de programación en C, necesarias para convertirle en un buen programador de este lenguaje.

Para utilizar este libro y obtener el máximo rendimiento se necesitará una computadora con un compilador C, una biblioteca instalada de modo que se puedan ejecutar los ejemplos del libro y un editor de texto para preparar sus archivos de código fuente. Existen numerosos compiladores de C en el mercado y también numerosas versiones *shareware* (libres de costes) disponibles en Internet. Idealmente, se debe elegir un compilador que sea compatible con la versión estándar de C del American National Standards Institute (ANSI), **C99**, que es la versión empleada en la escritura de este libro. La mayoría de los actuales compiladores disponibles de C++, comerciales o de dominio público, soportan C, por lo que tal vez ésta pueda ser una opción muy recomendable (en el Apéndice C, encontrará numerosos lugares de Internet, donde podrá encontrar compiladores, incluso gratuitos, para practicar con los numerosos ejemplos y ejercicios que incluimos en esta obra.)

Aunque el libro está concebido como un libro de problemas con los fundamentos teóricos mínimos imprescindibles para avanzar en su formación y se puede y debe utilizar de modo independientes, existe una posibilidad de utilizar este libro con otro de los mismos autores *Programación en C, 2ª edición*, publicado en 2005, también por McGraw-Hill. Este otro libro se escribió tanto en la 1ª como en la 2ª edición, como un libro eminentemente didáctico para cursos universitarios o profesionales de introducción a la programación y sigue un contenido similar a la obra que el lector tiene en sus manos, por lo que ambos pueden ser complementarios, uno eminentemente teórico-práctico y otro, el publicado en la colección Schaum, eminentemente práctico.

Usted puede utilizar cualquier editor de texto, tales como *Notepad* o *Vi*, para crear sus archivos de programas fuente, aunque será mucho mejor utilizar un editor específico para editar código C, como los que suelen venir con los entornos integrados de desarrollo, bien para Windows o para Linux. Sin embargo, no deberá utilizar un procesador de textos, tipo Microsoft Word, ya que normalmente los procesadores de texto o de tratamiento de textos comerciales, incrustan o “*embeben*” códigos de formatos en el texto que no serán entendidos por su compilador.

De cualquier forma, si usted sigue un curso reglado, el mejor método para estudiar este libro es seguir los consejos de su maestro y profesor tanto para su formación teórica como para su formación práctica. Si usted es un autodidacta o estudia de modo autónomo, la recomendación entonces será que compile, ejecute y depure de errores sus programas, tanto los propuestos en el libro, como los que usted diseñe, a medida que vaya leyendo el libro, tratando de entender la lógica del algoritmo y la sintaxis del lenguaje en cada ejercicio que realice.

¿Cómo está organizado el libro?

Todos los capítulos siguen una estructura similar: breve *Introducción* al capítulo; *fundamentos teóricos básicos* necesarios para el aprendizaje con numerosos ejemplos; *problemas resueltos en C*, donde se incluye el análisis del problema y el algoritmo (código en C); y por último, todos los capítulos contienen una colección de problemas propuestos, cuyo objetivo es facilitar al lector la medición de su aprendizaje.

Capítulo 1. Introducción a la ciencia de la computación y a la programación. Explica y describe los conceptos fundamentales de la computación y de los lenguajes de programación.

Capítulo 2. Fundamentos de programación. Se introduce al lector en los conceptos fundamentales de algoritmos y sus herramientas de representación. Así mismo se describen los tipos clásicos de programación con especial énfasis en la programación estructurada soporte del lenguaje C.

Capítulo 3. El lenguaje C. Elementos básicos. Introduce a la estructura y los componentes principales de un programa en C: datos, constantes, variables y las operaciones básicas de entrada/salida.

Capítulo 4. Operadores y expresiones. Se aprende el uso de los operadores aritméticos, relacionales y lógicos para la manipulación de operaciones y expresiones en C. Se estudian también operadores especiales y conversiones de tipos, junto con reglas de prioridad y *asociatividad* de los operadores en las expresiones y operaciones matemáticas.

Capítulo 5. Estructuras de selección: sentencias *if* y *switch*. Introduce a las sentencias de selección fundamentales en cualquier programa. Se examina el uso de sentencias compuestas o bloques así como el uso de operadores condicionales y evaluación de expresiones lógicas.

Capítulo 6. Estructuras de control: bucles. Se aprende el concepto de bucle o lazo y el modo de controlar la ejecución de un programa mediante las sentencias *for*, *while* y *do-while*. También se explica el concepto de anidamiento de bucles y bucles vacíos; se proporcionan ejemplos útiles para el diseño eficiente de bucles.

Capítulo 7. Funciones. Examina las funciones en C, una parte importante de la programación. Aprende programación estructurada – un método de diseño de programas que enfatiza en el enfoque descendente para la resolución de problemas mediante la descomposición del problema grande en problemas de menor nivel que se implementan a su vez con funciones.

Capítulo 8. Funciones recursivas. La recursividad o propiedad de una función o expresión de llamarse a sí misma es una de las técnicas más importantes en la construcción de algoritmos.

Capítulo 9. Arrays (listas y tablas). Explica un método sencillo pero potente de almacenamiento de datos. Se aprende como agrupar datos similares en *arrays* o “arreglos” (listas y tablas) numéricas

Capítulo 10. Ordenación y búsqueda. Enseña los métodos para ordenar listas y tablas, así como búsqueda de datos en listas y tablas. Se estudian los algoritmos clásicos más sencillos y eficientes tanto de ordenación como de búsqueda

Capítulo 11. Estructuras y uniones. Se describen conceptos básicos de estructuras, uniones y enumeraciones: declaración, definición, iniciación, uso y tamaño. Las operaciones fundamentales de acceso a estructuras, *arrays* de estructuras y estructuras anidadas se analizan también en este capítulo; se muestra de modo práctico como usar estructuras y uniones para conseguir las necesidades del programa; se explican las diferencias entre estructuras y uniones, así como el uso de la palabra reservada *typedef*.

Capítulo 12. Punteros (Apuntadores). Presenta una de las características más potentes y eficientes del lenguaje C, los *punteros*. Se describe con detalle los punteros, *arrays* de punteros, punteros de cadena, aritmética de punteros, punteros constantes, punteros como argumentos de funciones, punteros a funciones y a estructuras. De un modo práctico aprende el modo de utilizar punteros a punteros y cómo se pueden utilizar los *arrays* de punteros para manipular las cadenas, que se estudiarán en profundidad en el capítulo 14.

Capítulo 13. Asignación dinámica de memoria. Se describe la gestión dinámica de la memoria y las funciones asociadas para esas tareas: *alloc()*, *free()*, *calloc()* y *realloc()*. Se proporcionan reglas de funcionamiento de esas funciones y reglas para asignación de memoria.

Capítulo 14. Cadenas. Se describe el concepto de cadena (*string*) así como las relaciones entre punteros, *arrays* y cadenas en C. Se introducen conceptos básicos de manipulación de cadenas junto con operaciones básicas tales como longitud, concatenación, comparación, conversión y búsqueda de caracteres y cadenas. Se describen las funciones más notables de la biblioteca *string.h*.

Capítulo 15. Entrada y salida de archivos. Se estudia el concepto de flujo (*stream*) y los diferentes métodos de apertura de archivos, junto con los conceptos de archivos binarios y funciones para el acceso aleatorio. Muestra de un modo práctico como C utiliza los flujos, examina los flujos predefinidos y el modo práctico de trabajar con la pantalla, la impresora y el teclado.

Capítulo 16. Organización de datos en un archivo. Los conceptos clásicos de registros y organización de archivos se estudian en el capítulo. Dos tipos de archivo especiales tales como los secuenciales indexados y con direccionamiento hash son motivo de estudio específico. Por último se analizan métodos de ordenación de archivo tanto externa como por mezcla directa.

Capítulo 17. Tipos de datos (TAD/Objetos). La programación orientada a objetos es el paradigma más importante después del paradigma estructurado. El rol de la abstracción, la modularidad y los tipos abstractos de datos son analizados en este capítulo. Se describen la especificación e implementación de tipos abstractos de datos en C como primer nivel de objetos.

Capítulo 18. Listas enlazadas. Una lista enlazada es una estructura de datos que mantiene una colección de elementos, pero el número de ellos no se conoce por anticipado o varía en un rango amplio. La lista enlazada se compone de elementos que contienen un valor y un puntero. El capítulo describe los fundamentos teóricos, tipos de listas y operaciones que se pueden realizar en la lista enlazada.

Capítulo 19. Pilas y colas. Las estructuras de datos más utilizadas desde el punto de vista de abstracción e implementación son las pilas y colas. Su estructura, diseño y manipulación de los algoritmos básicos se explican en el capítulo.

Capítulo 20. Árboles. Las estructuras de datos no lineales y dinámicas son muy utilizadas en programación. Los árboles son una de las estructuras más conocidas en algoritmia y en programación ya que son la base para las técnicas de programación avanzada.

Apéndices

Compilación de programas C en UNIX y Linux
Compilación de programas C en Windows
Recursos de programación Web

C y la Web

C. Algoritmos, Programación y Estructura de Datos es básicamente un libro práctico para aprender a programar con gran cantidad de problemas resueltos y sus códigos fuente respectivos. Por esta razón y con el objetivo principal de que el lector se centre en el enunciado del problema, en la resolución del algoritmo y en su codificación y dado el carácter reiterativo que en muchos casos requiere la codificación, se han incluido todos los códigos fuente de los ejercicios y problemas del libro en su dirección Web oficial: <http://www.mhe.es/joyanes>. De igual forma, y como ayuda al lector y al maestro/profesor, en el portal del libro se incluyen también ejercicios y problemas complementarios, tutoriales, bibliografía complementaria, etc. Así mismo el profesor tiene a su disposición, si así lo desea una página específica donde encontrará material didáctico complementario que puede ser de su interés: diapositivas (acetatos, *slides*)

AGRADECIMIENTOS

A nuestro editor Carmelo Sánchez que con sus sabios consejos técnicos y editoriales siempre contribuye a la mejor edición de nuestros libros. Nuestro agradecimiento eterno, amigo y editor. También y como siempre, a todos nuestros compañeros del departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Facultad de Informática y de la Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid que en esta ocasión y como siempre nos animan, aconsejan y asesoran en la distribución de los temas y nos dan sus opiniones para mejora de nuestras obras. Gracias, colegas y amigos. Naturalmente a nuestros lectores, razón de ser de nuestro libro. Confiamos no defraudar la confianza depositada en esta obra y aspiramos a que su progresión en el aprendizaje de la programación sea todo lo rápida y eficiente que deseamos. Por último, un agradecimiento especial a todos los profesores y maestros que han utilizado nuestra obra *Programación en C*; muchos nos habéis animado a escribir este nuevo libro de modo complementario y que sirviera tanto de modo independiente como unido al libro citado en talleres y prácticas de programación. Nuestro agradecimiento más sincero a todos y nuestra disponibilidad si así lo consideran oportuno.

Los autores

En Madrid, Mayo de 2005

Introducción a las computadoras y a los lenguajes de programación

Las computadoras (ordenadores) electrónicas modernas son uno de los productos más importantes del siglo XXI ya que se han convertido en un dispositivo esencial en la vida diaria y han cambiado el modo de vivir y de hacer negocios. El papel de los programas de computadoras es fundamental; sin una lista de instrucciones a seguir, la computadora es virtualmente inútil. Los lenguajes de programación permiten escribir esos programas.

En el capítulo se introduce a conceptos importantes tales como la organización de una computadora, el hardware, el software y sus componentes, y a los lenguajes de programación más populares y en particular a C. En esta obra, usted comenzará a estudiar la ciencia de la computación, la ingeniería informática o la ingeniería de sistemas a través de uno de los lenguajes de programación más versátiles disponibles hoy día, el lenguaje C, y también la metodología a seguir para la resolución de problemas con computadoras.

1.1 Organización física de una computadora

Una **computadora** (también **ordenador**) es un dispositivo electrónico, utilizado para procesar información y obtener resultados, capaz de ejecutar cálculos y tomar decisiones a velocidades millones o cientos de millones más rápidas que puedan hacerlo los seres humanos. En el sentido más simple una computadora es “un dispositivo” para realizar cálculos o computar. El término sistema de computadora o simplemente computadora se utiliza para enfatizar que, en realidad, son dos partes distintas: **hardware** y **software**. El **hardware** es el computador en sí mismo. El **software** es el conjunto de programas que indican a la computadora las tareas que debe realizar. Las computadoras procesan datos bajo el control de un conjunto de instrucciones denominadas **programas de computadora**. Estos programas controlan y dirigen a la computadora para que realice un conjunto de acciones (instrucciones) especificadas por personas especializadas, los **programadores de computadoras**.

Los datos y la información se pueden introducir en la computadora por su **entrada** (*input*) y a continuación se procesan para producir su **salida** (*output*, resultados), como se observa en la Figura 1.1. La computadora se puede considerar como una unidad en la que se ponen ciertos datos (*datos de entrada*), se procesan estos datos y produce un resultado (*datos de salida o información*). Los datos de entrada y los datos de salida pueden ser, realmente, de cualquier tipo, texto, dibujos, sonido, imágenes,.... El sistema más sencillo para comunicarse una persona con la computadora es mediante un teclado, una pantalla (monitor) y un ratón (*mouse*). Hoy día existen otros dispositivos muy populares tales como escáneres, micrófonos, altavoces, cámaras de vídeo, etc.; de igual manera, mediante *módems*, es posible conectar su computadora con otras computadoras a través de la red **Internet**.

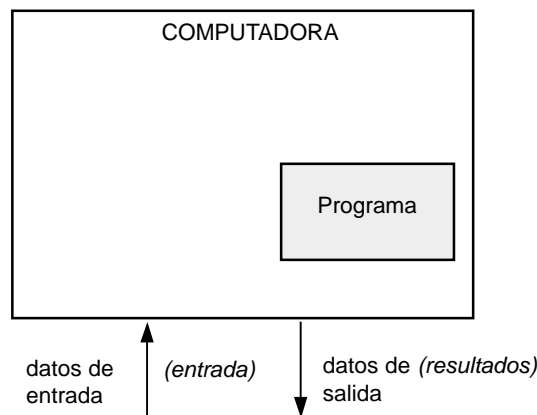


Figura 1.1 Proceso de información en una computadora.

Los dos componentes principales de una computadora son: *Hardware* y *Software*. **Hardware** es el equipo físico o los dispositivos asociados con una computadora. El conjunto de instrucciones que indican a la computadora aquello que debe hacer se denomina **software** o **programas**. Este libro se centra en la enseñanza y aprendizaje de la programación o proceso de escribir programas.

Una computadora consta fundamentalmente de cinco componentes principales: *dispositivo(s) de entrada*; *dispositivos de salida*; *unidad central de proceso (UCP)* o *procesador* (compuesto de la **UAL**, Unidad Aritmética y Lógica, y la **UC**, Unidad de Control); *memoria principal o central*; *memoria secundaria o externa* y los *programas*.

Las computadoras sólo entienden un lenguaje binario digital o lenguaje máquina. El programa se debe transferir primero de la *memoria secundaria* a la *memoria principal* antes de que pueda ser ejecutado. Los datos se deben proporcionar por alguna fuente. La persona que utiliza un programa (**usuario** de programa) puede proporcionar datos a través de un dispositivo de entrada. Los datos pueden proceder de un **archivo (fichero)**, o pueden proceder de una máquina remota vía una conexión de red.

Los paquetes de datos (de 8, 16, 32, 64 o más bits a la vez) se mueven continuamente entre la UCP (CPU) y todos los demás componentes (memoria RAM, disco duro, etc.). Estas transferencias se realizan a través de *buses*. Los **buses** son los canales de datos que interconectan los componentes del PC.

DISPOSITIVOS DE ENTRADA/SALIDA (E/S)

Los dispositivos de *Entrada/Salida* (E/S) [*Input/Output (I/O)* en inglés] permiten la comunicación entre la computadora y el usuario. Los *dispositivos de entrada*, como su nombre indica, sirven para introducir datos (información) en la computadora para su proceso. Dispositivos de entrada típicos son los **teclados**; otros son: **lectores de tarjetas** - ya en desuso -, **lápices ópticos**, **palancas de mando (joystick)**, **lectores de códigos de barras**, **escáneres**, **micrófonos**, etc. Hoy día tal vez el dispositivo de

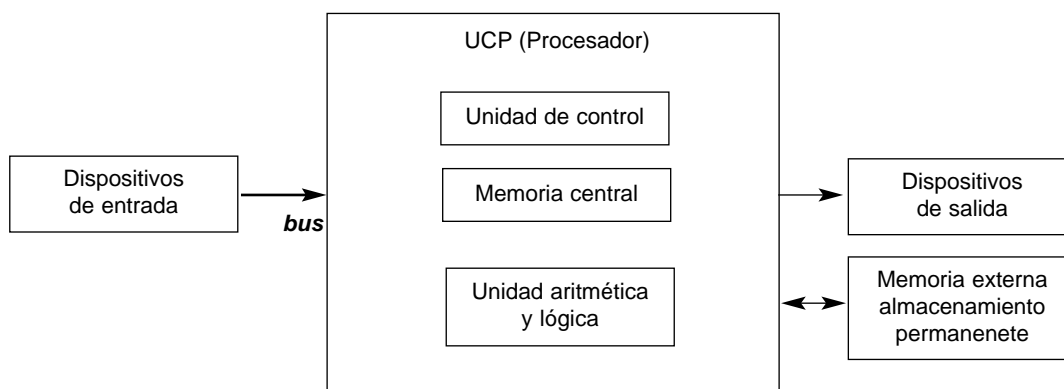


Figura 1.2 Organización física de una computadora.

entrada más popular es el **ratón** (*mouse*) que mueve un puntero sobre la pantalla que facilita la interacción usuario-máquina. Los *dispositivos de salida* permiten representar los resultados (salida) del proceso de los datos. El dispositivo de salida típico es la **pantalla** o **monitor**. Otros dispositivos de salida son: **impresoras** (imprimen resultados en papel), **trazadores gráficos** (*plotters*), **reconocedores de voz**, **altavoces**, etc.

El teclado y la pantalla constituyen - en muchas ocasiones - un único dispositivo, denominado **terminal**. En ocasiones a la impresora se la conoce como **dispositivo de copia dura** (“*hard copy*”), debido a que la escritura en la impresora es una copia permanente (dura) de la salida, y a la pantalla se le denomina en contraste: **dispositivo de copia blanda** (“*soft copy*”), ya que se pierde la pantalla actual cuando se visualiza la siguiente.

Los dispositivos de entrada/salida y los dispositivos de almacenamiento secundario o auxiliar (memoria externa) se conocen también con el nombre de *dispositivos periféricos* o simplemente **periféricos** ya que, normalmente, son externos a la computadora. Estos dispositivos son unidad de discos (disquetes, CD-ROM, DVD, cintas, etc.), videocámaras, teléfonos celulares (móviles), etc.

MEMORIA

La **memoria principal** es uno de los componentes más importantes de una computadora y sirve para el almacenamiento de información (datos y programas). La memoria central de una computadora es una zona de almacenamiento organizada en centenares o millares de unidades de almacenamiento individual o celdas. El término *bit* (*dígito binario*) se deriva de las palabras inglesas “**binary digit**” y es la unidad de información más pequeña que puede tratar una computadora. El término **byte** es muy utilizado en la jerga informática y, normalmente, se suelen conocer a las palabras de 16 bits como palabras de 2 *bytes*, y a las palabras de 32 bits como palabras de 4 *bytes*.

La memoria central de una computadora puede tener desde unos centenares de millares de *bytes* hasta millones de *bytes*. Como el *byte* es una unidad elemental de almacenamiento, se utilizan múltiplos para definir el tamaño de la memoria central: **Kilo-byte (KB)** igual a 1.024 bytes (2^{10}), **Megabyte (MB)** igual a 1.024 x 1.024 bytes ($2^{20}=1.048.576$), **Gigabyte (GB)** igual a 1.024 MB ($2^{30}=1.073.741.824$). Las abreviaturas **MB** y **GB** se han vuelto muy populares como unidades de medida de la potencia de una computadora.

EJEMPLO 1.1 Unidades de medida de almacenamiento

Byte	Byte (B)	<i>equivale a</i>	8 bits
Kilobyte	Kbyte (KB)	<i>equivale a</i>	1.024 bytes (2^{10})
Megabyte	Mbyte (MB)	<i>equivale a</i>	1.024 Kbytes (2^{20})
Gigabyte	Gbyte (GB)	<i>equivale a</i>	1.024 Mbytes (2^{30})
Terabyte	Tbyte (TB)	<i>equivale a</i>	1.024 Gbytes (2^{40})
Petabyte	Pbyte (PB)	<i>equivale a</i>	1.024 Tbytes (2^{50})
Exabyte	Ebyte (EB)	<i>equivale a</i>	1.024 Ebytes (2^{60})
Zettabyte	Zbyte (ZB)	<i>equivale a</i>	1.024 Ebytes (2^{70})
Yotta	Ybyte (YB)	<i>equivale a</i>	1.024 Ybytes (2^{80})

$$1 \text{ TB} = 1.024 \text{ GB} \quad 1 \text{ GB} = 1.024 \text{ MB} = 1.048.576 \text{ KB} = 1.073.741.824 \text{ B}$$

ESPACIO DE DIRECCIONAMIENTO

Para tener acceso a una palabra en la memoria se necesita un identificador o dirección. Cada celda o *byte* tiene asociada una única *dirección* que indica su posición relativa en memoria y mediante la cual se puede acceder a la posición para almacenar o recuperar información. La información almacenada en una posición de memoria es su contenido. El contenido de estas direcciones o posiciones de memoria se llaman **palabras**, que como ya se ha comentado pueden ser de 8, 16, 32 y 64 bits. Por consiguiente, si trabaja con una máquina de 32 bits, significa que en cada posición de memoria de su computadora puede alojar 32 bits, es decir 32 dígitos, bien ceros o unos.

El número de posiciones únicas identificables en memoria se denomina **espacio de direccionamiento**. Por ejemplo en una memoria de 64 *kilobytes* (KB) y un tamaño de palabra de un *byte* tienen un espacio de direccionamiento que varía de 0 a 65.535 direcciones de memoria (64KB, $64 \times 1.024 = 65.536$)

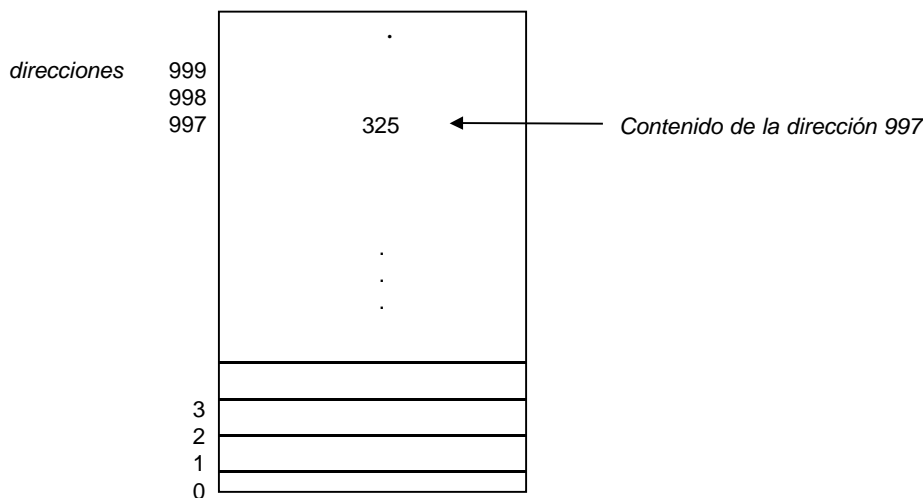


Figura 1.3. Memoria central de una computadora

LA MEMORIA PRINCIPAL

La **memoria central (RAM, Random Access Memory)** o simplemente **memoria** se utiliza para almacenar de modo temporal información, datos y programas. En general, la información almacenada en memoria puede ser de dos tipos: las *instrucciones* de un programa y los *datos* con los que operan las instrucciones. Para que un programa se pueda *ejecutar* (correr, rodar, funcionar,..., en inglés *run*), debe ser situado en la memoria central, en una operación denominada *carga (load)* del programa. Después, cuando se ejecuta (se realiza, funciona) el programa, *cualquier dato a procesar por el programa se debe llevar a la memoria* mediante las instrucciones del programa. En la memoria central, hay también datos diversos y espacio de almacenamiento temporal que necesita el programa cuando se ejecuta con el fin de poder funcionar.

Es un tipo de memoria volátil (su contenido se pierde cuando se apaga la computadora); esta memoria es, en realidad, la que se suele conocer como memoria principal o de trabajo. La memoria **ROM**, es una memoria que almacena información de modo permanente en la que no se puede escribir (viene pregrabada por el fabricante) ya que es una **memoria de sólo lectura**.

Con el objetivo de que el procesador pueda obtener los datos de la memoria central más rápidamente, la mayoría de los procesadores actuales (muy rápidos) utilizan con frecuencia una **memoria** denominada **cache** que sirve para almacenamiento intermedio de datos entre el procesador y la memoria principal.

LA UNIDAD CENTRAL DE PROCESO (UCP)

La *Unidad Central de Proceso*, **UCP (Central Processing Unit, CPU**, en inglés), dirige y controla el proceso de información realizado por la computadora. La UCP procesa o manipula la información almacenada en memoria y consta de dos componentes: *unidad de control (UC)* y *unidad aritmético-lógica (UAL)*. La **unidad de control (Control Unit, CU)** coordina las actividades de la computadora y determina qué operaciones se deben realizar y en qué orden; asimismo controla y sincroniza todo el proceso de la computadora. La **unidad aritmético-lógica (Arithmetic-Logic Unit, ALU)** realiza operaciones aritméticas y lógicas, tales como suma, resta, multiplicación, división y comparaciones. Las series de operaciones requeridas para procesar una instrucción de máquina se llaman **ciclo de la máquina**. Los ciclos de máquina se suelen medir en nanosegundos o picosegundos.

El **procesador o microprocesador** es un *chip (circuito integrado)* que controla y realiza las funciones y operaciones. En realidad el microprocesador representa a la Unidad Central de Proceso. Todas las UCP tienen una velocidad de trabajo, regulada por un pequeño *crystal de cuarzo*, y que se conoce como *frecuencia de reloj*. El número de ciclos de reloj por segundo se mide en hertzios. La velocidad de los microprocesadores se mide en MHz o en GHz.

DISPOSITIVOS DE ALMACENAMIENTO SECUNDARIO (ALMACENAMIENTO MASIVO)

La memoria secundaria, mediante los dispositivos de almacenamiento secundario, proporciona capacidad de almacenamiento fuera de la UCP y del almacenamiento o memoria principal. El almacenamiento secundario es no volátil. Las unidades (*drives*,

en inglés), periféricos o dispositivos de almacenamiento secundario son dispositivos que actúan como medio de soporte para almacenar los datos –temporal o permanentemente– que ha de manipular la CPU durante el proceso en curso y que no puede contener la memoria principal.

Las tecnologías de almacenamiento secundario más importantes son discos magnéticos, discos ópticos y cintas magnéticas.

La información almacenada en la memoria secundaria se conserva en unidades de almacenamiento denominadas **archivos** (**ficheros**, *files* en inglés) que pueden ser tan grandes como se desee. Los resultados de los programas se pueden guardar como *archivos de datos* y los programas que se escriben se guardan como *archivos de programas*, ambos en la memoria auxiliar.

Los discos son dispositivos formados por componentes electromagnéticos que permiten un acceso rápido a bloques físicos de datos. La información se registra en la superficie del disco y se accede a ella por medio de cabezas de *lectura/escritura* que se mueven sobre la superficie. Los discos magnéticos se clasifican en **disquetes** (*floppy disk*) y **discos duros** (*hard disk*).

Los **discos ópticos** difieren de los tradicionales discos duros o discos magnéticos en que los primeros utilizan un haz de láser para grabar la información. Son dispositivos de almacenamiento que utilizan la misma tecnología que los dispositivos compactos de audio para almacenar información digital. Los dos grandes modelos existentes en la actualidad son los discos compactos (CD) y los discos versátiles digitales (DVD).

El **CD-ROM** (*Compact Disk-Read Only Memory*, *Disco compacto - Memoria de solo lectura*) es el medio ideal para almacenar información de forma masiva que no necesita ser actualizada con frecuencia (dibujos, fotografías, enciclopedias,...). La llegada de estos discos al mercado hizo posible el desarrollo de la *multimedia*, es decir, la capacidad de integrar medios de todo tipo (texto, imágenes, sonido e imágenes).

El **DVD** (*Digital Versatil Disk*): **Videodisco digital** (**DVD-+R**, **DVD-+RW**, **DVD-RAM**) nació en 1995, gracias a un acuerdo entre los grandes fabricantes de electrónica de consumo, estudios de cine y de música (Toshiba, Philips, Hitachi., JVC,...). Son dispositivos de alta capacidad de almacenamiento, interactivos y con total compatibilidad con los medios existentes. Es capaz de almacenar hasta 26 CD con una calidad muy alta y con una capacidad que varía, desde los 4.7 GB del tipo de una cara y una capa hasta los 17 GB de la de dos caras y dos capas.

Las **cintas magnéticas** son los primeros dispositivos de almacenamiento de datos que se utilizaron y por ello, hasta hace poco tiempo – y aún hoy– han sido los más empleados para almacenar copias de seguridad. Poseen una gran capacidad de almacenamiento pero tienen un gran inconveniente, son *dispositivos de almacenamiento de acceso secuencial*. Por esta razón, la rapidez de acceso a los datos en las cintas es menor que en los discos.

Las cintas de audio digital (DAT, Digital Audio Tape) son unidades de almacenamiento con capacidad para grabar varios GB de información en un único cartucho. Una memoria flash, también comercializada como un disco, es un pequeño almacén de memoria móvil de pequeño tamaño.

Un **dispositivo de entrada** es cualquier dispositivo que permite que una persona envíe información a la computadora. Los dispositivos de entrada, por excelencia, son un teclado y un ratón. El uso del ratón y de menús facilita dar órdenes al computador y es mucho más sencillo que las tediosas órdenes con combinaciones de teclas que siempre se deben memorizar. Algunos dispositivos de entrada no tan típicos pero cada vez más usuales en las configuraciones de sistemas informáticos son: escáner, lápiz óptico, micrófono y reconocedor de voz.

1.2 Redes

Hoy día los computadores autónomos (*standalone*) prácticamente no se utilizan (excepción hecha del hogar) y están siendo reemplazados hasta en los hogares y en las pequeñas empresas, por redes de computadores. Una red es un conjunto de computadores conectados entre sí para compartir recursos.

Las redes se pueden clasificar en varias categorías siendo las más conocidas las redes de área local (**LAN**, *local area network*) y las redes de área amplia (**WAN**, *wide area network*). Una WAN es una red que enlaza muchos computadores personales y redes de área local en una zona geográfica amplia. La red más conocida y popular en la actualidad es la red Internet que está soportada por la World Wide Web.

El **sistema cliente-servidor** divide el procesamiento de las tareas entre los computadores “cliente” y los computadores “servidor” que a su vez están conectados en red. A cada máquina se le asignan funciones adecuadas a sus características. El **cliente** es el usuario final o punto de entrada a la red y normalmente en un computador personal de escritorio o portátil, o una estación de trabajo. El usuario, normalmente interactúa directamente sólo con la parte cliente del sistema, normalmente, para entrada o recuperación de información y uso de aplicaciones para análisis y cálculos posteriores.

El **servidor** proporciona recursos y servicios a otros computadores de la red (los clientes). El servidor puede ser desde un gran computador a otro computador de escritorio pero especializado para esta finalidad y mucho más potente. Los servidores almacenan y procesan los datos compartidos y también realizan las funciones no visibles, de segundo plano (*back-end*), a los

usuarios, tales como actividades de gestión de red, implementación de bases de datos, etc. Otra forma de sistema distribuido es la computación **P2P** (*peer-to-peer*) que es un sistema que enlaza a los computadores vía Internet o redes privadas de modo que pueden compartir tareas de proceso. El modelo P2P se diferencia del modelo de red cliente/servidor en que la potencia de proceso reside solo en los computadores individuales de modo que trabajan juntos colaborando entre sí, pero sin un servidor o cualquier otro computador los controle.

Una **red de área local (LAN, local area network)** normalmente une a decenas y a veces centenares de computadores en una pequeña empresa u organismo público. Una **red global**, tal como Internet, que se expande a distancias mucho mayores y conecta centenares o millares de máquinas que a su vez se unen a redes más pequeñas a través de computadores pasarela (*gateway*). Un computador pasarela (*gateway*) es un puente entre una red tal como Internet en un lado y una red de área local en el otro lado. La computadora también suele actuar como un **cortafuegos (firewall)** cuyo propósito es mantener las transmisiones ilegales, no deseadas o peligrosas fuera del entorno local.

INTERNET Y LA WORLD WIDE WEB

Internet, conocida también como la Red de Redes, se basa en la tecnología Cliente/Servidor. Las personas que utilizan la Red controlan sus tareas mediante aplicaciones Web tal como software de navegador. Todos los datos incluyendo mensajes de correo-e y las páginas Web se almacenan en servidores. Un cliente (usuario) utiliza Internet para solicitar información de un servidor Web determinado situado en computador lejano.

Las plataformas cliente incluyen PC y otros computadores, pero también un amplio conjunto de dispositivos electrónicos (*handheld*) tales como PDA, teléfonos móviles, consolas de juegos, etc., que acceden a Internet de modo inalámbrico (sin cables) a través de señales radio.

La World Wide Web (**WWW**) o simplemente la Web fue creada en 1989 por Bernards Lee en el CERN (European Laboratory for Particles Physics) aunque su difusión masiva comenzó en 1993 como medio de comunicación universal. La Web es un sistema de estándares aceptados universalmente para almacenamiento, recuperación, formateado y visualización de información, utilizando una arquitectura cliente/servidor. Se puede utilizar la Web para enviar, visualizar, recuperar y buscar información o crear una página Web. La Web combina texto, hipermedia, sonidos y gráficos, utilizando interfaces gráficas de usuario para una visualización fácil.

Para acceder a la Web se necesita un programa denominado navegador Web (*browser*). Se utiliza el navegador para visualizar textos, gráficos y sonidos de un documento Web y activar los enlaces (*links*) o conexiones a otros documentos. Cuando se hace clic (con el ratón) en un enlace a otro documento se produce la transferencia de ese documento situado en otro computador a su propio computador.

La Web se basa en un lenguaje estándar de hipertexto denominado **HTML** (Hypertext Markup Language) que da formatos a documentos e incorpora enlaces dinámicos a otros documentos almacenados en el mismo computador o en computadores remotos.

Otros servicios que proporciona la Web y ya muy populares para su uso en el mundo de la programación son: el correo electrónico y la mensajería instantánea. El correo electrónico (*e-mail*) utiliza protocolos específicos para el intercambio de mensajes: **SMTP** (*Simple Mail Transfer Protocol*), **POP** (*Post Office Protocol*) e **IMAP** (*Internet Message Action Protocol*).

1.3 El software (los programas)

Las operaciones que debe realizar el *hardware* son especificadas por una lista de instrucciones, llamadas programas, o *software*. Un programa de software es un conjunto de **sentencias** o **instrucciones** al computador. El proceso de escritura o codificación de un programa se denomina **programación** y las personas que se especializan en esta actividad se denominan **programadores**. Existen dos tipos importantes de software: software del sistema y software de aplicaciones. Cada tipo realiza una función diferente.

Software del sistema es un conjunto generalizado de programas que gestiona los recursos del computador, tal como el procesador central, enlaces de comunicaciones y dispositivos periféricos. Los programadores que escriben software del sistema se llaman **programadores de sistemas**. **Software de aplicaciones** son el conjunto de programas escritos por empresas o usuarios individuales o en equipo y que instruyen a la computadora para que ejecute una tarea específica. Los programadores que escriben software de aplicaciones se llaman **programadores de aplicaciones**.

SISTEMA OPERATIVO

Cuando un usuario interactúa con un computador, la interacción está controlada por el sistema operativo. Un usuario se comunica con un sistema operativo a través de una interfaz de usuario de ese sistema operativo. Los sistemas operativos modernos

utilizan una interfaz gráfica de usuario, **IGU** (*Graphical User Interface*, GUI) que hace uso masivo de iconos, botones, barras y cuadros de diálogo para realizar tareas que se controlan por el teclado o el ratón (*mouse*) entre otros dispositivos. Normalmente el sistema operativo se almacena de modo permanente en un chip de memoria de sólo lectura (ROM). Otra parte del sistema operativo puede residir en disco que se almacena en memoria RAM en la inicialización del sistema por primera vez en una operación que se llama *carga* del sistema (*booting*)

El sistema operativo dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de cintas y discos. Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos pueden ser: *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios), atendiendo al número de usuarios y *monocarga* (una sola tarea) o *multitarea* (múltiples tareas) según las tareas (procesos) que puede realizar simultáneamente. C corre prácticamente en todos los sistemas operativos, Windows 95, Windows NT/2000, Windows XP, UNIX, Linux,... y en casi todas las computadoras personales actuales PC, Mac, Sun, etc.

TIPOS DE SISTEMAS OPERATIVOS

Las diferentes características especializadas del sistema operativo permiten a los computadores manejar muchas diferentes tareas así como múltiples usuarios de modo simultáneo o en paralelo, bien de modo secuencial. En base a sus características específicas los sistemas operativos se pueden clasificar en varios grupos:

La multiprogramación permite a múltiples programas compartir recursos de un sistema de computadora en cualquier momento. Con multiprogramación, un grupo de programas se ejecutan alternativamente y se alternan en el uso del procesador. Cuando se utiliza un sistema operativo de un único usuario, la multiprogramación toma el nombre de **multitarea**.

Un sistema operativo multiusuario es un sistema operativo que tiene la capacidad de permitir que muchos usuarios compartan simultáneamente los recursos de proceso de la computadora. Dada la alta velocidad de transferencia de las operaciones, la sensación es que todos los usuarios están conectados simultáneamente a la UCP.

Un sistema operativo trabaja en multiproceso cuando puede enlazar a dos o más UCP para trabajar en paralelo en un único sistema de computadora. El sistema operativo puede asignar múltiples UCP para ejecutar diferentes instrucciones del mismo programa o de programas diferentes simultáneamente, dividiendo el trabajo entre las diferentes UCP.

1.4 Lenguajes de programación

Como se ha visto en el apartado anterior, para que un procesador realice un proceso se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de *interpretar* el algoritmo, lo que significa:

- comprender las instrucciones de cada paso,
- realizar las operaciones correspondientes.

Cuando el procesador es una computadora, el algoritmo se ha de expresar en un formato que se denomina *programa*. Un programa se escribe en un *lenguaje de programación*. Los principales tipos de lenguajes utilizados en la actualidad son tres:

- *lenguajes máquina*,
- *lenguaje de bajo nivel (ensamblador)*,
- *lenguajes de alto nivel*.

Los **lenguajes máquina** son aquellos que están escritos en lenguajes directamente inteligibles por la máquina (computadora), ya que sus instrucciones son *cadena binarias*. Las instrucciones en lenguaje máquina dependen del *hardware* de la computadora y, por tanto, diferirán de una computadora a otra. Las *ventajas* de programar en lenguaje máquina son la posibilidad de cargar (transferir un programa a la memoria) sin necesidad de traducción posterior, lo que supone una velocidad de ejecución superior a cualquier otro lenguaje de programación. Los *inconvenientes* - en la actualidad - superan a las ventajas, lo que hace prácticamente no recomendables los lenguajes máquina. Estos inconvenientes son:

- dificultad y lentitud en la codificación,
- poca fiabilidad,
- dificultad grande de verificar y poner a punto los programas,
- los programas sólo son ejecutables en el mismo procesador (UPC, *Unidad Central de Proceso*)

Los **lenguajes de bajo nivel** son más fáciles de utilizar que los lenguajes máquina, pero, al igual, que ellos, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el *ensamblador* (*assembly language*). Las instrucciones

en lenguaje ensamblador son instrucciones conocidas como **nemotécnicos** (*mnemonics*). Por ejemplo, nemotécnicos típicos de operaciones aritméticas son: en inglés, ADD, SUB, DIV, etc.; en español, SUM, RES, DIV, etc.

EJEMPLO 1.2 Programación en lenguaje de bajo nivel

Una instrucción típica de suma sería:

ADD M, N, P

Esta instrucción podría significar «*sumar el número contenido en la posición de memoria M al número almacenado en la posición de memoria N y situar el resultado en la posición de memoria P*». Evidentemente, es mucho más sencillo recordar la instrucción anterior con un nemotécnico que su equivalente en código máquina:

0110 1001 1010 1011

Un programa escrito en lenguaje ensamblador no puede ser ejecutado directamente por la computadora - en esto se diferencia esencialmente del lenguaje máquina -, sino que requiere una fase de *traducción* al lenguaje máquina. El programa original escrito en lenguaje ensamblador se denomina *programa fuente* y el programa traducido en lenguaje máquina se conoce como *programa objeto*, ya directamente inteligible por la computadora. El traductor de programas fuente a objeto es un programa llamado *ensamblador* (*assembler*).

Los lenguajes ensambladores presentan la *ventaja* frente a los lenguajes máquina de su mayor facilidad de codificación y, en general, su velocidad de cálculo. Los *inconvenientes* más notables de los lenguajes ensambladores son:

- Dependencia total de la máquina, lo que impide la transportabilidad de los programas (posibilidad de ejecutar un programa en diferentes máquinas).
- La formación de los programas es más compleja que la correspondiente a los programadores de alto nivel, ya que exige no sólo las técnicas de programación, sino también el conocimiento del interior de la máquina.

Hoy día los lenguajes ensambladores tienen sus aplicaciones muy reducidas en la programación de aplicaciones y se centran en aplicaciones de tiempo real, control de procesos y de dispositivos electrónicos, etc.

Los **lenguajes de alto nivel** son los más utilizados por los programadores. Los programas escritos en lenguaje de alto nivel son *portables* o *transportables*, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras. Los lenguajes de alto nivel presentan las siguientes *ventajas*:

- El tiempo de formación de los programadores es relativamente corto comparado con otros lenguajes.
- La escritura de programas se basa en reglas sintácticas similares a los lenguajes humanos.
- Las modificaciones y puestas a punto de los programas son más fáciles.
- Reducción del coste de los programas.
- Transportabilidad.

Los *inconvenientes* se concretan en:

- Incremento del tiempo de puesta a punto, al necesitarse diferentes traducciones del programa fuente para conseguir el programa definitivo.
- No se aprovechan los recursos internos de la máquina, que se explotan mucho mejor en lenguajes máquina y ensambladores.
- Aumento de la ocupación de memoria.
- El tiempo de ejecución de los programas es mucho mayor.

Los lenguajes de programación de alto nivel existentes hoy son muy numerosos, aunque la práctica demuestra que su uso mayoritario se reduce a

C C++ COBOL FORTRAN Pascal Visual BASIC VB.Net Java C#

El mundo Internet consume gran cantidad de recursos en forma de lenguajes de programación tales como Java, HTML, XML, JavaScript, PHP, etc.

Los **traductores de lenguaje** son programas que traducen a su vez los programas fuente escritos en lenguajes de alto nivel a código máquina. Los traductores se dividen en **compiladores** e **interpretes**

Un **intérprete** es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta. Un **compilador** es un programa que traduce los programas fuente escritos en lenguaje de alto nivel – C, FORTRAN, C++, Java,...- a lenguaje máquina. Los programas escritos en lenguaje de alto nivel se llaman *programas fuente* y el programa traducido *programa objeto* o *código objeto*. El compilador traduce - sentencia a sentencia - el programa fuente. Los lenguajes compiladores típicos son : C, Pascal, FORTRAN y COBOL.

LA COMPILACIÓN Y SUS FASES

La *compilación* es el proceso de traducción de programas fuente a programas objeto. El programa objeto obtenido de la compilación ha sido traducido normalmente a código máquina. Para conseguir el programa máquina real se debe utilizar un programa llamado *montador* o *enlazador* (*linker*). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable (Fig. 1.4)

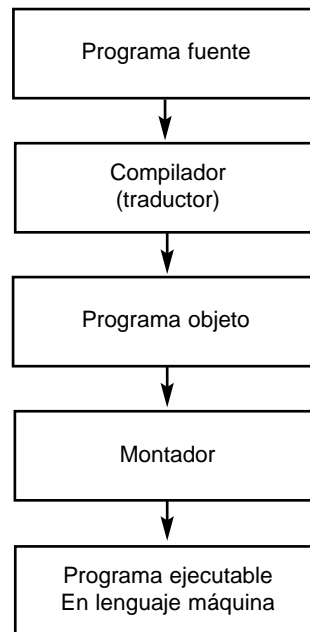


Figura 1.4 Fases de la compilación.

El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador suele tener los siguientes pasos:

- Escritura del *programa fuente* con un *editor*.
- *Compilar* el programa con el compilador C.
- *Verificar y corregir errores de compilación* (listado de errores).
- Obtención del *programa objeto*.
- El enlazador (*linker*) obtiene el *programa ejecutable*.
- Se ejecuta el programa y, si no existen errores, se obtendrá la salida del programa.

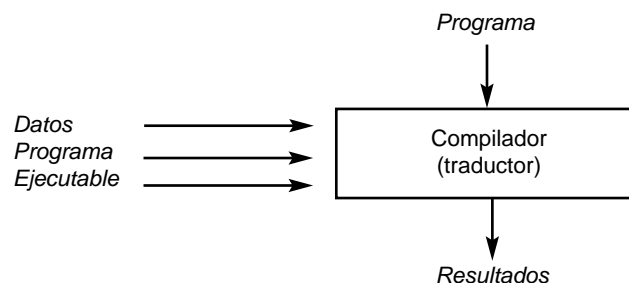


Figura 1.5 Ejecución de un programa.

El proceso de ejecución sería el mostrado en las figuras 1.5 y 1.6.

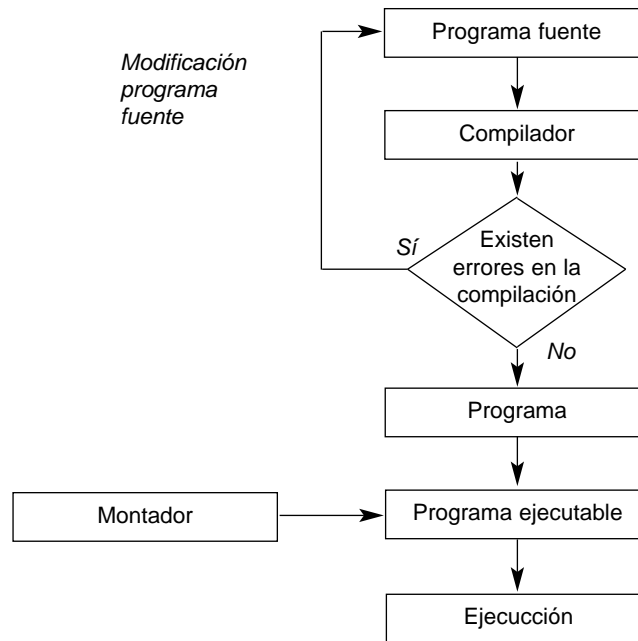


Figura 1.6 Fases de ejecución de un programa.

En el capítulo 2 se describirá en detalle el proceso completo y específico de ejecución de programas en lenguaje C

1.5 El lenguaje C: historia y características

C es el lenguaje de programación de propósito general asociado, de modo universal, al sistema operativo UNIX. Sin embargo, la popularidad, eficacia y potencia de C, se ha producido porque este lenguaje no está prácticamente asociado a ningún sistema operativo, ni a ninguna máquina, en especial. Esta es la razón fundamental, por la cual C, es conocido como el ***lenguaje de programación de sistemas***, por excelencia.

C es una evolución de los lenguajes BCPL –desarrollado por Martin Richards- y B –desarrollado por Ken Thompson en 1970- para el primitivo UNIX de la computadora DEC PDP-7. C nació realmente en 1978, con la publicación de *The C Programming Language*, de Brian Kernighan y Dennis Ritchie (Prentice Hall, 1978). En 1983, el American National Standard Institute (ANSI), una organización internacional de estandarización, creó un comité (el denominado **X3J11**) cuya tarea fundamental consistía en hacer “una definición no ambigua del lenguaje C, e independiente de la máquina”. Con esta definición de C se asegura que cualquier fabricante de software que vende un compilador **ANSI C** incorpora todas las características del lenguaje, especificadas por el estándar. Esto significa también que los programadores que escriban programas en C estándar tendrán la seguridad de que correrán sus modificaciones en cualquier sistema que tenga un compilador C.

C es un *lenguaje de alto nivel*, que permite programar con instrucciones de lenguaje de propósito general. También, C se define como un lenguaje de programación estructurado de propósito general; aunque en su diseño también primó el hecho de fuera especificado como un lenguaje de *programación de sistemas*, y esta característica le proporciona una enorme cantidad de potencia y flexibilidad.

El estándar ANSI C formaliza construcciones no propuestas en la primera versión de C, en especial, asignación de estructuras y enumeraciones. Entre otras aportaciones, se definió esencialmente, una nueva forma de declaración de funciones (prototipos). Pero, es esencialmente la biblioteca estándar de funciones, otra de sus grandes aportaciones.

Hoy, en el siglo XXI, C sigue siendo uno de los lenguajes de programación más utilizados en la industria del *software*, así como en institutos tecnológicos, escuelas de ingeniería y universidades. Prácticamente todos los fabricantes de sistemas operativos, UNIX, Linux, MacOS, Solaris,... soportan diferentes tipos de compiladores de lenguaje C.

VENTAJAS DE C

El lenguaje C tiene una gran cantidad de ventajas sobre otros lenguajes, y son, precisamente, la razón fundamental de que después de casi dos décadas de uso, C siga siendo uno de los lenguajes más populares y utilizados en empresas, organizaciones y fábricas de software de todo el mundo. Algunas ventajas que justifican el uso todavía creciente del lenguaje C en la programación de computadoras son:

- El lenguaje C es potente y flexible.
- C se utiliza por programadores profesionales para desarrollar software en la mayoría de los modernos sistemas de computadora.
- Se puede utilizar C para desarrollar sistemas operativos, compiladores, sistemas de tiempo real y aplicaciones de comunicaciones.
- Un programa en C puede ser escrito para un tipo de computadora y trasladarse a otra computadora con pocas o ninguna modificación (propiedad conocida como portabilidad).

C se caracteriza por su velocidad de ejecución. En los primeros días de la informática, los problemas de tiempo de ejecución se resolvían escribiendo todo o parte de una aplicación en **lenguaje ensamblador** (lenguaje muy cercano al lenguaje máquina).

Debido a que existen muchos programas escritos en C, se han creado numerosas bibliotecas C para programadores profesionales que soportan gran variedad de aplicaciones. Existen bibliotecas del lenguaje C que soportan aplicaciones de bases de datos, gráficos, edición de texto, comunicaciones, etc.

En la actualidad son muchos los fabricantes de compiladores C, y se pueden encontrar en el comercio y de distribución gratuita tanto en empresas de distribución como en Internet para los sistemas operativos Windows, Linux, Unix y Mac, entre otros. Todos los compiladores del lenguaje C++ pueden ejecutar programas escritos en lenguaje C, preferentemente si cumplen el estándar ANSI C.¹

REFERENCIAS BIBLIOGRÁFICAS y LECTURAS SUPLEMENTARIAS

JOYANES AGUILAR, Luis (2003). *Fundamentos de programación. Algoritmos, Estructuras de datos y Objetos*, 3ª edición, Madrid: McGraw-Hill.

Libro de referencia para el aprendizaje de la programación con un lenguaje algorítmico. Libro complementario de esta obra y que ha cumplido ya quince años desde la publicación de su primera edición

KARBO, Michael B. (2004) *Arquitectura del PC. Teoría y Práctica*, 2ª edición. Barcelona: PC-Cuadernos Técnicos, nº 17, KnowWare E.U.R.L (www.pc-cuadernos.com).

Excelente manual de *hardware*. Eminentemente práctico con una gran cantidad de información. Al formar parte de una colección editada de modo periódico, tiene un precio muy económico. Recomendable por su calidad y bajo coste.

LAUDON, Kennet C. y LAUDON, Jane P. (2003). *Essentials of Management Information Systems*. Fifth edition. Upper Saddle River: New Jersey: Prentice Hall,

Magnífico libro sobre sistemas de información. Escrito con sencillez pero con un gran rigor técnico. Está acompañado de una gran cantidad de gráficos y figuras ilustrativas. Actualizado totalmente a los modernos sistemas de información.

Recomendable para continuar formándose en informática fundamental y de modo complementario a la lectura de esta obra.

LÓPEZ CRUZ, Pedro A. (2004). *Hardware y componentes*. Madrid: Anaya.

Completo y actualizado libro del hardware de un computador. Contiene las características técnicas junto con sus correspondientes explicaciones no solo del computador como máquina sino de todos sus componentes tanto internos como externos. Es una excelente referencia para conocer más sobre todos los dispositivos *hardware* modernos de un computador.

¹ Opciones gratuitas buenas puede encontrar en el sitio del fabricante de software Borland. También puede encontrar y descargar un compilador excelente Dev-C++ en software libre que puede compilar código C y también código C++, en www.bloodshed.net y en www.download.com puede así mismo encontrar diferentes compiladores totalmente gratuitos. Otros numerosos sitios puede encontrar en software gratuito en numerosos sitios de la red. Los fabricantes de software y de computadoras (IBM, Microsoft, HP,...) ofrecen versiones a sus clientes aunque normalmente no son gratuitos

MAÑAS, José Antonio. (2004). *Mundo IP: Introducción a los secretos de Internet y las Redes de Datos*. Madrid. Ediciones Nowtilus.

Libro completo sobre Redes de Datos e Internet. Muy docente. Escrito por un catedrático de la Universidad Politécnica de Madrid, su estilo es muy agradable, sencillo, sin por ello dejar el rigor científico y técnico. Recomendable para el lector que desee conocer el mundo de Internet y de las Redes de Datos, por otra parte necesarios para la formación de todo programador en C. Conveniente su lectura en paralelo con esta obra o a su terminación.

EJERCICIOS DE REPASO

1. La siguiente no es una ventaja del Lenguaje C
 - Se pueden escribir sistemas operativos y programas importantes para el sistema.
 - Está en lenguaje binario.
 - Es adecuado para escribir programas portables entre máquinas diferentes.
2. Las fases de ejecución de un programa en C son:
 - Análisis, Diseño e Implementación.
 - Compilación. Enlazado y Ejecución.
 - Depuración, Compilación y Verificación.
3. La Web utiliza sobre todo como lenguaje de programación
 - Los lenguajes C y C++.
 - Lenguaje HTML.
 - Lenguajes máquina.
4. Si un sistema operativo permite trabajar a la vez a varias personas se diría que es:
 - Un sistema multiproceso.
 - Un sistema multitarea.
 - Un sistema multiusuario.
5. Un cortafuegos es un componente importante de:
 - Un sistema conectado a Internet.
 - La Unidad Central de Procesamiento.
 - Los sistemas de almacenamiento secundario.
6. ¿Que parte de una computadora está dividida en palabras?
 - La memoria del sistema.
 - Los archivos que están en un disco USB.
 - Los programas ejecutables.
7. Los siguientes, son protocolos de Internet
 - IMAP, SMTP, HTTP.
 - CPU, ALU, USB.
 - LAN, WAN, DEC.
8. ¿En cuál de estos tipos de aplicación estaría justificado usar Lenguaje Ensamblador para programar:
 - Compiladores.
 - Aplicaciones de Tiempo Real.
 - Páginas Web.
9. Si se quisiese buscar alguna ventaja para elegir un lenguaje máquina para programar sería:
 - Por su facilidad de uso y depuración.
 - Por su rapidez de codificación.
 - Por su rapidez de ejecución.
10. El lenguaje C fue creado por:
 - El gobierno de Estados Unidos.
 - Una comisión de la organización ANSI.
 - Brian Kernighan y Dennis Ritchie.

Fundamentos de programación

Este capítulo le introduce a la metodología a seguir para la resolución de problemas con computadoras y con un lenguaje de programación tal como C.

La resolución de un problema con una computadora se hace escribiendo un programa, que exige al menos los siguientes pasos:

1. Definición o análisis del problema.
2. Diseño del algoritmo.
3. Transformación del algoritmo en un programa.
4. Ejecución y validación del programa.

Uno de los objetivos fundamentales de este libro es el aprendizaje y diseño de algoritmos. Este capítulo introduce al lector en el concepto de algoritmo y de programa, así como en las herramientas que permiten “dialogar” al usuario con la máquina: los lenguajes de programación.

2.1 Fases en la resolución de problemas

El proceso de resolución de un problema abarca desde la descripción inicial del problema hasta el desarrollo de un programa de computadora que lo resuelva. El proceso de diseño de programas tiene una serie de fases que generalmente deben seguir todos los programadores. Estas fases son: *análisis del problema*; *diseño del algoritmo*; *codificación*; *compilación y ejecución*, *verificación*; *depuración*; *mantenimiento y documentación*.

Las dos primeras fases conducen a un diseño detallado en forma de algoritmo. En la tercera fase se implementa el algoritmo en código escrito en un lenguaje de programación. En la fase de *compilación y ejecución* se traduce y ejecuta el programa. En las fases de *verificación y depuración* el programador busca errores de las etapas anteriores y los elimina.

Es importante definir el concepto de **algoritmo**. Un algoritmo es un conjunto finito de reglas que proponen una serie de operaciones que sirven para resolver un determinado problema y que cumple las siguientes características:

- *Finito*. Debe acabar siempre tras un número finito de pasos, si bien este número de pasos puede ser arbitrariamente grande.
- *“Definibilidad”*. Cada paso del algoritmo debe definirse de modo preciso. Las acciones del algoritmo deben estar expresadas sin ambigüedad.
- *Efectividad*. Las operaciones del algoritmo deben ser básicas, estar expresadas de modo exacto y deben ejecutarse en un tiempo finito.

- *Entrada*. Todo algoritmo debe tener cero o más datos de entrada.
- *Salida*. Todo algoritmo debe tener cero o más datos de salida.

Son ejemplos de algoritmos básicos: calcular el máximo común divisor de dos números; decidir si un número es primo; calcular el mayor de una secuencia de números; etc.

2.1.1 ANÁLISIS DEL PROBLEMA

Consiste en definir el problema y especificar claramente aquello que es necesario para su resolución. Para hacer el análisis hay que responder a las siguientes preguntas: ¿qué entrada tiene el problema?; ¿cuál es la salida deseada?; ¿qué método produce la salida deseada a partir de los datos de entrada? Normalmente la definición del problema comienza analizando los requisitos del usuario; pero estos requisitos con frecuencia suelen ser imprecisos y difíciles de escribir. La fase de especificación requiere normalmente una gran comunicación entre los programadores y los futuros usuarios del sistema.

2.1.2 DISEÑO DEL ALGORITMO

En esta etapa hay que indicar *cómo* hace el algoritmo la tarea solicitada, y eso se traduce en la construcción de un algoritmo. Los métodos más eficaces se basan en la técnica *divide y vencerás*. El problema se divide en subproblemas, y a continuación se divide cada uno de estos subproblemas en otros hasta que pueda ser implementada una solución. Esta fase de diseño es bastante larga, pero hay que tener en cuenta que el gasto de tiempo en la fase de diseño será ahorro de tiempo cuando se escriba y depure el programa. El resultado final del diseño es una solución que debe ser fácil de traducir a estructuras de datos y estructuras de control de un lenguaje de programación específico.

HERRAMIENTAS DE PROGRAMACIÓN

Las dos herramientas más comúnmente utilizadas para diseñar algoritmos son: *diagramas de flujo* y *pseudocódigos*.

- **Diagrama de flujo (*flowchart*)**. Es una representación gráfica de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (**ANSI**).
- **Pseudocódigo**. Es una herramienta de programación en la cual las instrucciones escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia el *pseudocódigo* (también **seudocódigo**) se puede definir como *un lenguaje de especificación de algoritmos*. Aunque no existen reglas para la escritura del pseudocódigo en español, se ha recogido una notación estándar que se utilizará en este libro y que ya es muy empleada en muchos libros de programación.

2.1.3 CODIFICACIÓN DE UN PROGRAMA

Consiste en escribir en un lenguaje de programación el algoritmo creado en la fase de diseño, debiendo seguirse las siguientes reglas:

- Si un problema se ha dividido en subproblemas los algoritmos que resuelven cada subproblema deben ser codificados y probados independientemente.
- Deben usarse como identificadores términos significativos usando nombres para los datos, y verbos para los subprogramas.
- Ha de tenerse especial cuidado en la comunicación de los distintos subprogramas, siendo recomendable que esta comunicación se realice siempre mediante los parámetros.
- Sólo deben usarse variables globales si son datos inherentes e importantes del programa.
- El sangrado (indentación) así como los buenos comentarios facilitan la posterior lectura del código.

2.1.4 COMPILACIÓN Y EJECUCIÓN DE UN PROGRAMA

Una vez que el algoritmo se ha convertido en programa fuente, es preciso introducirlo mediante un procesador de texto en la memoria de la computadora, para que mediante un compilador pueda ser traducido a lenguaje máquina. Si tras la compilación se presentan errores (*errores de compilación*) es necesario volver a editar el programa y corregirlos. Una vez corregidos los errores hay que ejecutar el programa, obteniéndose la salida de resultados, siempre que no existan errores (*errores de ejecución*).

2.1.5 VERIFICACIÓN Y DEPURACIÓN

La **depuración** de un programa es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores. Para ello hay que eliminar los errores de ejecución y los errores lógicos. Esta eliminación de errores se efectúa proporcionando al programa datos de entrada válidos que conducen a una solución conocida. También deben incluirse datos no válidos para comprobar la capacidad de detección de errores del programa.

Si bien el método de depuración es muy usado y proporciona buenos resultados; si se quiere estar seguros de que un programa funciona correctamente, hay que *probar* todos los posibles datos de entrada o una muestra suficientemente significativa, o bien *verificar* el programa, operación consistente en demostrar formalmente que el programa funciona correctamente.

2.1.6 DOCUMENTACIÓN Y MANTENIMIENTO

La documentación de un programa consiste en la descripción de cada uno de los pasos que hay que realizar para resolver el problema. La documentación de un programa puede ser *interna* (contenida en las líneas de comentario) o *externa* (contenida en análisis, diagramas de flujo, pseudocódigos, manuales de usuario con instrucciones de ejecución del programa etc.).

MANUAL DE USUARIO

Este manual es un documento comercial importante convierte al programa en más accesible y asequible al usuario. Es frecuente que este manual se edite como libro, aunque también suele incluirse en el propio programa en cuyo caso se denomina *manual de ayuda en línea*. Debe abarcar al menos los siguientes puntos:

- Órdenes necesarias para cargar el programa en memoria desde el almacenamiento secundario (disco) y para arrancar su funcionamiento.
- Nombres de los archivos externos a los que accede el programa.
- Formato de todos los mensajes de error o informes.
- Opciones en el funcionamiento del programa.
- Descripción detallada de la función realizada por el programa.
- Descripción detallada, preferiblemente con ejemplos, de cualquier salida producida por el programa.
- Instrucciones para la instalación del programa.

El mantenimiento del programa consiste en corregir posibles errores futuros de ejecución del programa, y en mejorar el programa añadiendo nuevas características o modificando las ya existentes debido a la necesidad de ejecutarlo en un nuevo entorno, la aparición de nuevo *hardware* o el cambio de las necesidades del usuario. Después de cada cambio, la documentación debe ser actualizada.

MANUAL DE MANTENIMIENTO

El manual de mantenimiento es la documentación requerida para mantener un programa durante su ciclo de vida. Se divide en dos categorías: documentación interna y documentación externa.

La **documentación interna** incluye:

- *Cabecera de programa* (con comentarios que reflejen el nombre del programador, fecha, versión, breve descripción del programa).
- Nombres significativos para describir identificadores.
- *Comentarios significativos*, encerrados entre llaves { } o bien paréntesis/asteriscos (* *), relativos a: misión de los módulos de que consta el programa; especificación de precondiciones y postcondiciones; explicación de partes confusas del algoritmo o descripción clara y precisa de los modelos de datos fundamentales y las estructuras de datos seleccionadas para su representación.
- Claridad de estilo y formato: una sentencia por línea, *indentación* (sangrado), líneas en blanco para separar módulos (procedimientos, funciones, unidades, etc.).

La **documentación externa** es ajena al programa fuente y se suele incluir en un manual que acompaña al programa. La documentación externa debe incluir:

- Listado actual del programa fuente, mapas de memoria, referencias cruzadas, etc.

- Especificación del programa: documento que define el propósito y modo de funcionamiento del programa.
- Diagrama de estructura que representa la organización jerárquica de los módulos que comprende el programa.
- Explicaciones de fórmulas complejas.
- Especificación de los datos a procesar: archivos externos incluyendo el formato de las estructuras de los registros, campos etc.
- Formatos de pantallas utilizados para interactuar con los usuarios.
- Cualquier indicación especial que pueda servir a los programadores que deben mantener el programa.

2.2 Programación estructurada

La **programación estructurada** consiste en escribir un programa de acuerdo con las siguientes reglas: el programa tiene un diseño modular; los módulos son diseñados descendientemente; cada módulo de programa (**subprograma**) se codifica usando las tres estructuras de control (*secuencia, selección e iteración*). Una definición más formal de programación estructurada es el conjunto de técnicas que incorporan: *recursos abstractos, diseño descendente y estructuras básicas de control*.

2.2.1 RECURSOS ABSTRACTOS

Descomponer un programa en términos de recursos abstractos consiste en descomponer acciones complejas en términos de acciones más simples capaces de ser ejecutadas en una computadora. La modularidad y la abstracción procedimental son complementarias, pudiéndose cambiar el algoritmo de un módulo sin afectar al resto de la solución de un problema. Una **abstracción procedimental** separa el propósito de un subprograma de su implementación. Una vez que el subprograma se haya codificado se puede usar sin necesidad de conocer su cuerpo y basta con su nombre y una descripción de sus parámetros.

2.2.2 DISEÑO DESCENDENTE (TOP DOWN)

El **diseño descendente** se encarga de resolver un problema realizando una descomposición en otros más sencillos, mediante el método de refinamiento por pasos. Se descompone el problema en etapas o estructuras jerárquicas, de forma que se puede considerar cada estructura desde los puntos de vista: *¿qué hace? ¿cómo lo hace?*. El resultado de esta jerarquía de módulos es que cada módulo se refina por los de nivel más bajo que resuelven problemas más pequeños y contienen más detalles sobre los mismos.

2.2.3 ESTRUCTURAS DE CONTROL

Las **estructuras de control** sirven para especificar el orden en que se ejecutarán las distintas instrucciones de un algoritmo. Este orden de ejecución determina el *flujo de control* del programa. Las tres estructuras básicas de control son: *secuencia, selección, iteración*. La programación estructurada hace los programas más fáciles de escribir, verificar, leer y mantener, utilizando un número limitado de estructuras de control que minimizan la complejidad de los problemas.

2.2.4 TEOREMA DE LA PROGRAMACIÓN ESTRUCTURADA

El teorema de la programación estructurada de Böhm y Jacopini (1966) dice que todo **programa propio** puede ser escrito usando solamente tres tipos de estructuras básicas de control que son: *secuencia, selección, iteración*. Un programa se dice que es propio si: tiene un solo punto de entrada y otro de salida; existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas las partes del programa; todas las instrucciones son ejecutables y no existen bucles infinitos.

2.3 Métodos formales de verificación de programas

Verificar un programa consiste en demostrar que el programa funciona correctamente. Para realizar esta verificación, hay que usar fórmulas matemáticas que deben expresar la semántica del programa, aportadas por la lógica de primer orden. La verificación formal es, en general, compleja y requiere conocimientos matemáticos avanzados, que comienzan con las **precondiciones** que deben cumplir los datos de entrada para llegar a las **postcondiciones** que deben cumplir los datos de salida, mediante los axiomas y las reglas de la lógica de primer orden. Para realizar la verificación formal de un programa hay que usar el sistema formal de Hoare (conjunto finito de axiomas y reglas de inferencia que sirven para razonar sobre la corrección parcial de un programa), y demostrar que el programa siempre termina mediante la obtención de expresiones cota. En este libro sólo se

consideran los siguientes conceptos clave: *asertos*, *precondiciones*, *postcondiciones* e *invariantes*, que ayudan a documentar corregir y clarificar el diseño de módulos y de programas.

Aserciones. Un aserto es una frase que describe la semántica de las variables y datos en un punto de un algoritmo. Estos asertos se escriben como comentarios y forman parte de la documentación del programa. Para expresar formalmente estos asertos se usan fórmulas de la lógica de primer orden. La lógica de primer orden emplea para su sintaxis además de los operadores lógicos (*or*, *not*, *and*...) los cuantificadores *para todo* (\forall) y *existe* (\exists).

Precondiciones y postcondiciones. Las precondiciones y postcondiciones son afirmaciones sencillas sobre condiciones al principio y al final de los módulos. Una *precondición* de un procedimiento es una afirmación lógica sobre sus parámetros de entrada; esta afirmación debe ser *verdadera* cuando se llama al procedimiento, para que éste comience a ejecutarse. Una *postcondición* de un procedimiento es una afirmación lógica que describe el cambio en el *estado del programa* producido por la ejecución del procedimiento; la postcondición describe el efecto de llamar al procedimiento. En otras palabras, la postcondición indicada será verdadera después que se ejecute el procedimiento.

Precondición. Predicado lógico que debe cumplirse al comenzar la ejecución de un módulo.

Postcondición. Predicado lógico que debe cumplirse al acabar la ejecución de un determinado módulo, siempre que se haya cumplido previamente la precondición.

Invariantes de bucles. Un invariante es una condición que es verdadera tanto antes como después de cada iteración (vuelta) y que describe la semántica del bucle. Se utiliza para documentar el bucle y sobre todo para determinar la corrección del mismo. Los siguientes cuatro puntos han de ser verdaderos:

- El invariante debe ser verdadero antes de que comience la ejecución por primera vez del bucle.
- Una ejecución del bucle debe mantener el invariante. Esto es, si el invariante es verdadero antes de cualquier iteración del bucle, entonces se debe demostrar que es verdadero después de la iteración.
- El invariante debe capturar la exactitud del algoritmo, demostrando que, si es verdadero cuando termina el bucle, el algoritmo es correcto.
- El bucle debe terminar después de un número finito de iteraciones.

La identificación de invariantes de bucles, ayuda a escribir bucles correctos. Se representa el invariante como un comentario que precede a cada bucle.

2.4 Factores de calidad del *software*

La construcción de software de calidad debe cumplir las siguientes características:

- **Eficiencia:** La eficiencia de un *software* es su capacidad para hacer un buen uso de los recursos de la computadora. Un sistema eficiente es aquel que usa pocos recursos de espacio y de tiempo.
- **Transportabilidad (*portabilidad*):** La *transportabilidad* o *portabilidad* es la facilidad con la que un *software* puede ser transportado sobre diferentes sistemas físicos o lógicos.
- **Fácil de usar:** Un *software* es fácil de utilizar cuando el usuario puede comunicarse con él de manera cómoda.
- **Compatibilidad:** Facilidad de los productos para ser combinados con otros y usados en diferentes plataformas *hardware* o *software*.
- **Corrección:** Capacidad de los productos *software* de realizar exactamente las tareas definidas por su especificación.
- **Extensibilidad:** Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esta característica, diseño simple y descentralización.
- **Robustez:** Capacidad de los productos *software* de funcionar, incluso, en situaciones anormales.
- **Verificabilidad:** La *verificabilidad*, facilidad de verificación, de un *software*, es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.
- **Reutilización:** Capacidad de los productos de ser *reutilizados*, en su totalidad o en parte, en nuevas aplicaciones.
- **Integridad:** La integridad es la capacidad de un software a proteger sus propios componentes contra los procesos que no tengan el derecho de acceder.

PROBLEMAS RESUELTOS¹

- 2.1. *Escriba el pseudocódigo de un algoritmo que lea tres números y si el primero es positivo calcule el producto de los otros dos, y en otro caso calcule la suma.*

Análisis del problema

Se usan tres variables enteras Numero1, Numero2, Numero3, en las que se leen los datos, y otras dos variables Producto y Suma en las que calcularemos o bien el producto o bien la suma.

Algoritmo

```

Algoritmo Producto_o_Suma
Variables
Entero Numero1, Numero2, Numero3, Producto, Suma
Inicio
  Leer(Numero1, Numero2, Numero3)
  Si (Numero1 > 0) entonces
    Producto ← Numero2 * Numero3
    Escribe('El producto de los dos últimos números es', Producto)
  Sino
    Suma ← Numero2 + Numero3
    Escribe('La suma de los dos últimos números es ', Suma)
  Fin si
Fin

```

- 2.2. *Escribir el pseudocódigo de un algoritmo que sume los 50 primeros números naturales.*

Análisis del problema

Se usa una variable Contador que cuenta los 50 primeros números naturales y una variable suma, para almacenar las sucesivas sumas, 1, 1+2, 1+2+3,.....

Algoritmo

```

Algoritmo cincuenta
Variables
Entero Contador, Suma
Inicio
  Contador ← 0
  Suma ← 0
  Mientras Contador <= 50 hacer
    Suma ← Suma + Contador
    Contador ← Contador + 1
  Fin mientras
  Escribe( Suma)
fin

```

- 2.3. *Escribir el pseudocódigo de un programa que lea 100 número enteros y calcule el mayor.*

¹ Estos problemas tiene como objetivo fundamental familiarizar al lector con la escritura de algoritmos y programas; le aconsejamos compile y ejecute los mismos en una computadora observando sus resultados. A medida que avance en el estudio de capítulos posteriores entenderá su lógica y modo de funcionamiento, y le sugerimos los revise en ese momento.

Análisis del problema

Se usa una variable entera `Numero` que se encarga de leer los números. Otra variable entera `Mayor` contiene en cada momento el mayor de los números leídos hasta el momento. Se programa un bucle que lee los números y en cada iteración almacena en la variable `Mayor` el número leído hasta el momento que sea el mayor de todos. En una variable `Contador` se lleva la cuenta de los números leídos

Algoritmo

```

Algoritmo ElMayor
Variables
Entero  Numero, Mayor, Contador
Inicio
  Leer(Numero)
  Mayor ← Numero
  Contador ← 1
  Mientras Contador < 50 hacer
    Contador ← Contador + 1
    Leer(Numero)
    Si(Mayor < Numero) entonces
      Mayor ← Numero
    Fin si
  Fin mientras
  Escribe(Mayor)
Fin

```

- 2.4.** *Escribir y compilar un programa C que lea 5 números enteros desde el teclado y calcule su suma. Expresa el invariante del bucle como una aserción hecha en lenguaje natural.*

Análisis del problema

Para escribir el programa, basta con definir una constante `n` que tome el valor 5, y mediante un bucle controlado por el contador `c`, ir leyendo números del teclado en la variable `Numero` y sumarlos en un acumulador `Suma`. Por lo tanto, el invariante del bucle debe decir si se han leído `c` números, siendo `c` menor o igual que `n` y en `Suma` se han acumulado los `c` números leídos.

Codificación

```

#include <stdio.h>
#define n 5
void main ( )
{
  int c, Numero, Suma;
  c = 0;
  Suma=0
  while( c < n )
  {
    c = c + 1;
    scanf("%d",&Numero);
    Suma = Suma + Numero;
    /* invariante= se han leído c números siendo c menor o igual
       que n y en Suma se han acumulado los c números leídos*/
  }
}

```

```
    printf("su suma es %d\n", Suma);
}
```

2.5. ¿Cuál es el invariante del bucle siguiente escrito en C?

```
Indice = 1; Suma = A[0];
while (Indice < N - 1)
{
    Indice = Indice + 1;
    Suma = Suma + A[Indice]
}
```

Análisis del problema

El invariante del bucle debe ser un predicado que se cumpla antes de la ejecución del bucle, al final de la ejecución de cada iteración del bucle y por supuesto en la terminación. Se supone que A es un vector cuyos valores se puedan sumar, y que los índices varían en el rango 0 hasta el N-1. En este caso el invariante del bucle debe expresar que en el acumulador suma se han sumado los elementos del array hasta la posición *Indice*.

$$INV \equiv \left(Suma = \sum_{k=0}^{Indice} A(k) \right) \wedge (Indice \leq N - 1)$$

Solución

```
Indice = 0;
Suma = A[0];
while (Indice < N - 1)
{
    Indice = Indice + 1;
    Suma = Suma + A[Indice]
```

$$INV \equiv \left(Suma = \sum_{k=0}^{Indice} A(k) \right) \wedge (Indice \leq N - 1)$$

```
}
```

```
    INV ^ (Indice ≥ N - 1)
```

2.6. Escriba el invariante del siguiente bucle escrito en C. Suponga que $n \geq 0$.

```
Indice = 0;
Maximo = A[0]
while (Indice != ( n - 1))
{
    Indice = Indice + 1;
    If( Máximo < A[indice])
        Maximo = A[Indice]
}
```

Análisis del problema

El invariante debe expresar que *Máximo* contiene el elemento mayor del array desde las posiciones 0 hasta la posición *Indice* y que además se encuentra en esas posiciones.

Solución

```
Indice = 0; Maximo = A[0];
```

$$INV \equiv (0 \leq \text{Indice} \leq n-1) \wedge (\forall k(0 \leq k \leq \text{Indice} \rightarrow \text{Máximo} \geq A(k)) \wedge \exists k(0 \leq k \leq \text{Indice} \wedge \text{Máximo} = A(k))) \wedge (\text{Indice} \leq n-1)$$

```
while (Indice != (n-1))
{
    Indice = Indice + 1;
    if (Maximo < A[Indice])
        Maximo = A[Indice];
```

$$INV \equiv (0 \leq \text{Indice} \leq n-1) \wedge (\forall k(0 \leq k \leq \text{Indice} \rightarrow \text{Máximo} \geq A(k)) \wedge \exists k(0 \leq k \leq \text{Indice} \wedge \text{Máximo} = A(k))) \wedge (\text{Indice} \leq n-1)$$

```
}
```

$$INV \wedge (\text{Indice} = n-1)$$

- 2.7.** *Escriba un programa en C que calcule la parte entera de la raíz cuadrada de un número positivo n usando solamente sumas de número naturales y comparaciones entre ellos. Expresé fórmulas de la lógica de primer orden que indiquen la semántica del programa en cada uno de sus puntos.*

Análisis del problema

El programa que se desarrolla está basado en la propiedad del cuadrado de una suma $(x+1)^2 = x^2 + 2x + 1$. Con la propiedad anterior y haciendo uso de un algoritmo voraz (variable x) de izquierda a derecha que avance con paso uno, y almacenando en una variable y los valores de $2x + 1$, se puede obtener el valor de $(x+1)^2$ si se tiene previamente almacenado el valor de x^2 .

Sea x un contador de números naturales. Sea y un contador de impares de números naturales ($y = 2x+1$) y z el cuadrado de $x+1$, ($z = (x+1)^2$). De acuerdo con estas propiedades si se inicializan convenientemente las variables x , y , z , el siguiente bucle calcula la parte entera de la raíz cuadrada del número n .

```
while (z <= n)
{
    x = x + 1;
    y = y + 2;
    z = z + y;
}
```

Con lo anteriormente dicho, la codificación del algoritmo pasa por leer el valor de n positivo, e inicializar convenientemente las variables x , y , z a los valores 0, 1, 1, respectivamente.

Solución

```
#include <stdio.h>
int main()
{
    int x, y, z, n;
    printf("introduzca valor de n \n");
    scanf("%d", &n);
    x = 0;
    y = 1;
    z = 1;
    while (z <= n)
```

```

{
    x = x + 1;
    y = y + 2;
    z = z + y;
}
printf("%d %d %d\n", x, y, z);
}

```

Las fórmulas que siguen expresan la semántica del programa hecha con la sintaxis de la lógica de primer orden.

```

#include <stdio.h>
int main()
{
    int x, y, z, n;
    printf("introduzca valor de n\n");
    scanf("%d", &n);
    x = 0; y = 1; z = 1;

```

$$0 \leq n \wedge (x = 0) \wedge (y = 1) \wedge (z = 1) \wedge (0 \leq n) \rightarrow (z = (x + 1)^2) \wedge (y = 2x + 1) \wedge (0 \leq n) \wedge (x^2 \leq n)$$

```

while (z <= n)
{

```

$$INV \wedge (z \leq n) \rightarrow (z + y + 2 = (x + 1 + 1)^2) \wedge (y + 2 = 2(x + 1) + 1) \wedge (0 \leq n) \wedge ((x + 1)^2 \leq n)$$

```

x = x + 1;

```

$$(z + y + 2 = (x + 1)^2) \wedge (y + 2 = 2x + 1) \wedge (0 \leq n) \wedge (x^2 \leq n)$$

```

y = y + 2;

```

$$(z + y = (x + 1)^2) \wedge (y = 2x + 1) \wedge (0 \leq n) \wedge (x^2 \leq n)$$

```

z = z + y;

```

$$INV \equiv (z = (x + 1)^2) \wedge (y = 2x + 1) \wedge (0 \leq n) \wedge (x^2 \leq n)$$

```

}

```

```

printf("%d %d %d\n", x, y, z);
}

```

$$(z = (x + 1)^2) \wedge (y = 2x + 1) \wedge (0 \leq n) \wedge (x^2 \leq n) \wedge (z > n) \rightarrow (0 \leq n) \wedge x = [n]$$

- 2.8.** Escriba un programa en C que calcule el producto de dos números naturales usando sólo sumas. Exprese fórmulas de la lógica de primer orden que indiquen la semántica del programa en cada uno de sus puntos.

Análisis del problema

Para resolver el problema se recuerda que para multiplicar dos números positivos basta con sumar tantas veces uno de ellos a una variable como unidades indique el otro. De esta manera si se usa un bucle voraz descendente, y si en las variables x , e y p contienen los datos, y en p se quiere obtener el producto, se tiene de manera obvia el siguiente bucle.

```

while (x != 0)
{

```

```

    x = x - 1;
    p = p + y;
}

```

La inicialización de las variables x , e y , será respectivamente a los dos datos iniciales y , en este problema, p a cero.

Codificación

```

#include <stdio.h>
int main()
{
    int a, b, x, y, p;
    printf(" dame a >0 y b >0 "); scanf("%d %d",&a,&b);
    x = a; y = b; p = 0;
    while (x != 0)
    {
        x = x - 1;
        p = p + y;
    }
    printf("%d %d %d\n", a, b, p);
}

```

Las fórmulas que siguen expresan la semántica del programa hecha con la sintaxis de la lógica de primer orden.

```

#include <stdio.h>
int main()
{
    int a, b, x, y, p;
    printf(" dame a >0 y b >0 "); scanf("%d %d",&a,&b);

```

$$(a \geq 0) \wedge (b \geq 0)$$

```

x = a;
y = b;

```

$$(x = a) \wedge (x \geq 0) \wedge (b \geq 0) \\ (p + x * y = a * b) \wedge (a \geq 0)$$

```

while (x != 0)
{

```

$$INV \wedge (x \neq 0) \rightarrow (p + y + (x - 1) * y = a * b) \wedge (x > 0)$$

```

    x = x - 1;

```

$$(p + y + x * y = a * b) \wedge (x \geq 0)$$

```

    p = p + y ;

```

$$INV \equiv (p + x * y = a * b) \wedge (x \geq 0)$$

```

}

```

$$(p = a * b)$$

```

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

```

PROBLEMAS PROPUESTOS

- 2.1. Escriba el pseudocódigo de un algoritmo para:
 - Sumar dos números enteros.
 - Restar dos números enteros.
 - Multiplicar dos números enteros.
 - Dividir dos números enteros.
- 2.2. Escriba el pseudocódigo de un algoritmo que lea la base y la altura de un triángulo y calcule su área.
- 2.3. El máximo común divisor de dos números enteros positivos es aquel número entero que divide a los dos números y es el mayor de todos. Escriba un algoritmo que calcule el máximo común divisor de dos números. Exprese en lenguaje natural el invariante del bucle.
- 2.4. Escriba un algoritmo que lea tres números enteros y decida si uno de ellos coincide con la suma de los otros dos.
- 2.5. Diseñar un algoritmo que lea e imprima una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe imprimir. Visualizar el número de valores leídos.
- 2.6. Diseñar un algoritmo que imprima y sume la serie de números 3, 6, 9,..... 99.
- 2.7. Diseñe un algoritmo que calcule la suma de los enteros 1, 2,.....,30. Exprese el invariante del bucle mediante especificación informal y formal.
- 2.8. Escriba un programa C que presente en pantalla todas las potencias enteras de 2 que sean menores o iguales que 100. Exprese el invariante del bucle formalmente.
- 2.9. Escriba un algoritmo que sume los números pares comprendidos entre 20 y 100 ambos inclusive.
- 2.10. Verifique el algoritmo del ejercicio 2.9

El lenguaje C: elementos básicos

Una vez que se le ha enseñado a crear sus propios programas, se analizan los fundamentos del lenguaje de programación C. Este capítulo comienza con un repaso de los conceptos teóricos y prácticos relativos a la estructura de un programa enunciados en el capítulo anterior, dada su gran importancia en el desarrollo de aplicaciones, incluyendo además los siguientes temas:

- creación de un programa;
- elementos básicos que componen un programa;
- tipos de datos en C y cómo se declaran;
- concepto de constantes y su declaración;
- concepto y declaración de variables;
- tiempo de vida o duración de variables;
- operaciones básicas de entrada/salida.

3.1 Estructura general de un programa en C

Una función en C es un grupo de instrucciones que realizan una o más acciones. Un programa C puede incluir: directivas de preprocesador; declaraciones globales; la función `main()`; funciones definidas por el usuario; comentarios del programa.

3.1.1 DIRECTIVAS DEL PREPROCESADOR

El **preprocesador** consta de *directivas* que son instrucciones al compilador. Todas las directivas del preprocesador comienzan con el signo de libro o “almohadilla” (`#`) y no terminan en punto y coma ya que no son instrucciones del lenguaje C. La directiva `#include` indica al compilador que lea el archivo fuente (archivo cabecera o de inclusión) que viene a continuación de ella y su contenido lo inserte en la posición donde se encuentra dicha directiva. Estas instrucciones son de la forma `#include <nombrearch.h>` o bien `#include "nombrearch.h"`. La directiva `#define` indica al preprocesador que defina un ítem de datos u operación para el programa C. Por ejemplo, la directiva `#define TAM 10` sustituirá el valor 10 cada vez que `TAM` aparezca en el programa.

3.1.2 DECLARACIONES GLOBALES

Las *declaraciones globales* indican al usuario que las constantes o variables así declaradas son comunes a todas las funciones de su programa. Se sitúan antes de la función `main()`. La zona de declaraciones globales puede incluir declaraciones de variables además de declaraciones de prototipos de función.

EJEMPLO 3.1 *Se realizan declaraciones de ámbito global.*

```
#include <stdio.h>

/* Definición de macros */

#define MICONST1 0.50
#define MICONST2 0.75

/* Declaraciones globales */

int Calificaciones;
int ejemplo (int x);
int main()
{
    ...
}
```

3.1.3 FUNCIÓN MAIN()

Cada programa C contiene una función `main()` que es un punto inicial de entrada al programa. Su estructura es:

```
int main()
{
    ...
    bloque de sentencias
}
```

Además de la función `main()`, un programa C consta de una colección de subprogramas que en C siempre son funciones. Las sentencias de C situadas en el cuerpo de la función `main()`, o de cualquier otra función, deben terminar en punto y coma.

3.1.4 FUNCIONES DEFINIDAS POR EL USUARIO

C proporciona funciones *predefinidas* (denominadas *funciones de biblioteca*) y *definidas por el usuario*. Se invocan por su nombre y los parámetros opcionales que incluye. Después de que la función sea llamada, el código asociado con la función se ejecuta y, a continuación, se retorna a la función llamadora. En C, las funciones definidas por el usuario requieren una *declaración* o *prototipo* en el programa, que indica al compilador el nombre por el cual ésta será invocada, el tipo y el número y tipo de sus argumentos. Las funciones de biblioteca requieren que se incluya el archivo donde está su declaración.

EJEMPLO 3.2 *Programa típico con una función `main()` y declaración de codificación de una función `prueba()`.*

```
#include <stdio.h>
void prueba();
int main()
{
    prueba();
    return 0;
}

void prueba()
{
    printf ( "Mis primeros pasos \n");
}
```

Un *comentario* es cualquier información que se añade a su archivo fuente. Los comentarios en C estándar comienzan con la secuencia `/*` y terminan con la secuencia `*/` (los compiladores C++ admiten también el tipo de comentario que empieza por `//`)

3.2 Los elementos de un programa C

Los elementos básicos de un programa C son : identificadores; palabras reservadas; comentarios; signos de puntuación; separadores y archivos cabecera.

Identificador. Un *identificador* es una secuencia de caracteres, letras, dígitos y subrayados. El primer carácter debe ser una letra (no un subrayado). Las letras mayúsculas y minúsculas son diferentes. Pueden tener cualquier longitud, pero el compilador ignora a partir del carácter 32. No pueden ser palabras reservadas.

Palabras reservadas. Una *palabra reservada*, tal como void, es una característica del lenguaje C asociada con algún significado especial. Una palabra reservada no se puede utilizar como nombre de identificador, objeto o función. Ejemplos de palabras reservadas son: asm, auto, break, case, char, const, continue, default, etc.

Comentarios. Los *comentarios* se encierran entre /* y */ pueden extenderse a lo largo de varias líneas. Los comentarios son ignorados por el compilador.

Signos de puntuación y separadores. Todas las sentencias de C deben terminar con un punto y coma. Los separadores son espacios en blanco, tabulaciones, retornos de carro y avances de línea.

Otros signos de puntuación son:

! % ^ & * () - + = { } ~
[] \ ; ' : < > ? , . / "

Archivos de cabecera. Un *archivo de cabecera* es un archivo especial que contiene las declaraciones de objetos y funciones de la biblioteca que son añadidos en el lugar donde se insertan. Un archivo cabecera se inserta con la directiva #include.

3.3 Tipos de datos en C

Los tres tipos de datos básicos de C son: *enteros*; *números de coma flotante (reales)* y *caracteres*. La Tabla 3.1 recoge los principales tipos de datos básicos, sus tamaños en bytes y el rango de valores que puede almacenar.

Tabla 3.1 Tipos de datos simples de C.

Tipo	Ejemplo	Tamaño en bytes	Rango Mínimo..Máximo
char	'C'	1	0..255
short	-15	2	-128..127
int	1024	2	-32768..32767
unsigned int	42325	2	0..65535
long	262144	4	-2147483648..2147483637
float	10.5	4	3.4*(10 ⁻³⁸)..3.4*(10 ³⁸)
double	0.00045	8	1.7*(10 ⁻³⁰⁸)..1.7*(10 ³⁰⁸)
long double	1e-8	8	igual que double

3.3.1 ENTEROS (INT)

Los tipos enteros se almacenan internamente en 2 bytes de memoria. La Tabla 3.2 resume los tres tipos enteros básicos, junto con el rango de valores y el tamaño usual en bytes (depende de cada compilador C).

Tabla 3.2 Tipos de datos enteros.

Tipo C	Rango de valores	Uso recomendado
int	-32.768 .. +32.767	Aritmética de enteros, bucles for, conteo.
unsigned int	0 .. 65.535	Conteo, bucles for, índices.
short int	-32.768 .. +32.767	Aritmética de enteros, bucles for, conteo.

DECLARACIÓN DE VARIABLES

La forma más simple de una declaración de variable en C es declarar el tipo de dato y a continuación el nombre de la variable, seguida, opcionalmente de su valor inicial *<tipo de dato> <nombre de variable> = <valor inicial>*. Se pueden también declarar múltiples variables en la misma línea:

```
<tipo_de_dato> <nom_var1>, <nom_var2> ... <nom-varn>
```

C permite escribir constantes enteras en *octal* (base 8) o *hexadecimal* (base 16). La Tabla 3.3 muestra ejemplos de constantes enteras representadas en sus notaciones decimal, hexadecimal y octal.

Tabla 3.3 Constantes enteras en tres bases diferentes.

Base 10 Decimal	Base 16 Hexadecimal (Hex)	Base 8 Octal
8	0x08	010
10	0x0A	012
16	0x10	020
65536	0x10000	0200000
24	0x18	030
17	0x11	021

Si el rango de los tipos enteros básicos no es suficientemente grande para sus necesidades, se consideran tipos enteros largos. La Tabla 3.4 muestra los dos tipos de datos enteros largos. Ambos tipos requieren 4 bytes de memoria (32 bits) de almacenamiento.

Tabla 3.4 Tipos de datos enteros largos.

Tipo C	Rango de valores
long	-2147483648 .. 2147483647
unsigned long	0 .. +4294967295

3.3.2 TIPOS DE COMA FLOTANTE (FLOAT/DOUBLE)

Los tipos de datos de coma (*punto*) flotante representan números reales que contienen una coma (un punto) decimal, tal como 3.14159, o números muy grandes, tales como 1.85×10^{15} . La declaración de las variables de coma flotante es igual que la de variables enteras. C soporta tres formatos de coma flotante. El tipo `float` requiere 4 bytes de memoria, `double` requiere 8 bytes y `long double` requiere 10 bytes. La Tabla 3.5 muestra los tipos de datos en coma flotante.

Tabla 3.5 Tipos de datos en coma flotante.

Tipo C	Rango de valores	Precisión
float	3.4×10^{-38} .. 3.4×10^{38}	7 dígitos
double	1.7×10^{-308} .. 1.7×10^{308}	15 dígitos
long double	3.4×10^{-4932} .. 1.1×10^{4932}	19 dígitos

3.3.3 CARACTERES (CHAR)

C procesa datos carácter (tales como texto) utilizando el tipo de dato `char`. Este tipo representa valores enteros en el rango -128 a +127. El lenguaje C proporciona el tipo `unsigned char` para representar valores de 0 a 255 y así representar todos los caracteres ASCII. Los caracteres se almacenan internamente como números, y por tanto se pueden realizar operaciones aritméticas con datos tipo `char`.

EJEMPLO 3.4 Definir e inicializar una variable de tipo `char`, a continuación convertir a mayúscula.

```
char car = 'b';
car = car - 32;
```

El ejemplo convierte b (código ASCII 98) a B (código ASCII 66).

3.4 El tipo de dato lógico

Los compiladores de C no incorporan el tipo de dato *lógico*. C usa el tipo `int` para simular el tipo lógico interpretando todo valor distinto de 0 como "verdadero" y el valor 0 como "falso". Una expresión lógica que se evalúa a 0 se considera falsa; una expresión lógica que se evalúa a 1 (o valor entero distinto de 0) se considera verdadera.

3.5 Constantes

Una constante es un objeto cuyo valor no puede cambiar a lo largo de la ejecución de un programa.

Constantes literales. Las constantes literales o constantes, en general, se clasifican en cuatro grupos, cada uno de los cuales puede ser de cualquiera de los tipos: constantes enteras; constantes reales; constantes de caracteres; constantes de cadena.

Constantes enteras. Son una sucesión de dígitos precedidos o no por el signo + o – dentro de un rango determinado. Por ejemplo, 234, y -456.

Constantes reales. Son una sucesión de dígitos con un punto delante, al final o en medio y seguidos opcionalmente de un exponente: Por ejemplo, 82.347, .63, 83., 47e-4,.25E7 y 61.e+4.

Constantes carácter. Una constante carácter (`char`) es un carácter del código ASCII encerrado entre apóstrofes. Por ejemplo, 'A', 'b', 'c'.

Constantes cadena. Una constante cadena es una secuencia de caracteres encerrados entre dobles comillas. Por ejemplo, "123", "12 de octubre 1492", "esto es una cadena". En memoria, las cadenas se representan por una serie de caracteres ASCII más un 0 o nulo que es definido en C mediante la constante `NULL`.

Constantes definidas (simbólicas). Las constantes pueden recibir nombres simbólicos mediante la directiva `#define`.

EJEMPLO 3.5 Se ponen nombres simbólicos a constantes de interés.

```
#define NUEVALINEA '\n'
#define PI 3.1415929 /* valor de la constante Pi */
```

Constantes numeradas. Las constantes enumeradas permiten crear listas de elementos afines. Por ejemplo:

```
enum dias {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

Al procesar esta sentencia el compilador *enumera* los identificadores comenzando por 0. Después de declarar un tipo de dato enumerado, se pueden crear variables de ese tipo, como con cualquier otro tipo de datos.

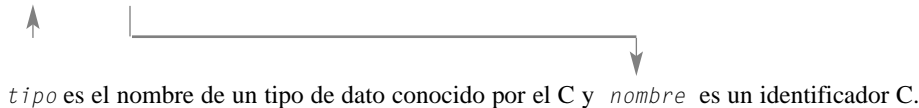
Constantes declaradas `const` y `volatile`. El cualificador `const` permite dar nombres simbólicos a constantes. Su valor no puede ser modificado por el programa. Su formato es: `const tipo nombre = valor`; La palabra reservada `volatile` actúa como `const`, pero su valor puede ser modificado no sólo por el propio programa, sino también por el *hardware* o por el *software* del sistema.

3.6 Variables

En C una *variable* es una posición con nombre (identificador) de memoria donde se almacena un valor de un tipo de dato. Su valor puede cambiar a lo largo de la ejecución del programa.

Delaración. Una *declaración* de una variable es una sentencia que proporciona información de la variable al compilador C. Es preciso *declarar* las variables antes de utilizarlas. Su sintaxis es:

```
tipo nombre;
```



`tipo` es el nombre de un tipo de dato conocido por el C y `nombre` es un identificador C.

3.7 Entradas y salidas

La biblioteca C proporciona facilidades para entrada y salida. Los programas que utilizan estas facilidades requieren incluir el archivo de cabecera `stdio.h`.

Salida. La función `printf()` visualiza en la pantalla datos del programa, transforma los datos, que están en representación binaria, a ASCII según los códigos transmitidos. El formato general que tiene la función es:

```
printf(cadena_de_control, dato1, dato2, ..., dato);
```

<code>cadena_de_control</code>	contiene los tipos de los datos y forma de mostrarlos.
<code>dato1, dato2 ...</code>	variables, constantes, o en general expresiones de salida.

Los códigos de formato más utilizados y su significado:

<code>%d</code>	El dato se convierte a entero decimal.
<code>%o</code>	El dato entero se convierte a octal.
<code>%x</code>	El dato entero se convierte a hexadecimal.
<code>%u</code>	El dato entero se convierte a entero sin signo.
<code>%c</code>	El dato se considera de tipo carácter.
<code>%e</code>	El dato se considera de tipo <code>float</code> o <code>double</code> se convierte a <i>notación científica</i> .
<code>%f</code>	El dato se considera de tipo <code>float</code> o <code>double</code> se convierte a <i>notación decimal</i> .
<code>%g</code>	El dato se considera de tipo <code>float</code> o <code>double</code> se convierte al código <code>%e</code> o <code>%f</code> , dependiendo de la representación mas corta.
<code>%s</code>	El dato ha de ser una cadena de caracteres.

Entrada. La función mas utilizada para la entrada de datos a través del teclado es `scanf()`. Su formato es:

```
scanf (cadena_de_control, var1, var2, var3, ..., varn);
```

<code>cadena_de_control</code>	contiene los tipos de los datos y si se desea su anchura.
<code>var1, var2 ...</code>	variables del tipo de los códigos de control.

Los códigos de formato más comunes son los ya indicados en la salida.

Salida de cadenas de caracteres. Con la función `printf()` se puede dar salida a cualquier dato, asociándolo el código que le corresponde. En particular, para dar salida a una cadena de caracteres se utiliza el código `%s`. Para salida de cadenas, la biblioteca C proporciona la función específica `puts()`; tiene un solo argumento, que es una cadena de caracteres; escribe la cadena en la salida estándar (pantalla) y añade el fin de línea.

Entrada de cadenas de caracteres. La entrada de una cadena de caracteres se hace con la función mas general `scanf()` y el código `%s`. `scanf()` con el código `%s` capta palabras, el criterio de terminación es el encontrarse un blanco, o bien fin de línea. La biblioteca de C dispone de una función específica para leer una cadena de caracteres. Es la función `gets()`, que lee del dispositivo estándar de entrada una cadena de caracteres. Termina la captación con un retorno de carro. `gets(variable_cadena);`

PROBLEMAS RESUELTOS

3.1. ¿Cual es la salida del siguiente programa?.

```
#include <stdio.h>
#define prueba "esto es una prueba"
int main()
{
    char cadena[21]="sale la cadena.";
    puts(prueba);
    puts("Escribimos de nuevo.");
    puts(cadena);
    puts(&cadena[8]);
    return 0;
}
```

Solución

```
esto es una prueba
Escribimos de nuevo.
sale la cadena.
cadena.
```

3.2. Codifique un programa en C que escriba en dos líneas distintas las frases
Bienvenido a la programación en C
Pronto comenzaremos a programar en C.

Codificación

```
#include <stdio.h>
int main()
{
    printf("Bienvenido a la programación en C\n");
    printf(" Pronto comenzaremos a programar en C\n");
    return 0;
}
```

- 3.3. Codifique un programa en C que copie en un array de caracteres la frase *es un nuevo ejemplo en C* y lo escriba en la pantalla.

Codificación

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

int main()
{
    char ejemplo[50];
    strcpy (ejemplo, " Es un nuevo ejemplo de programa en C\n");
    printf( ejemplo);
    return 0;
}
```

- 3.4. ¿Cual es la salida del siguiente programa?.

```
#include <stdio.h>
#define Constante "de declaracion de constante."
int main(void)
{
    char Salida[21]="Esto es un ejemplo" ;
    puts(Salida);
    puts(Constante);
    puts("Salta dos lineas\n");
    puts("y tambien un");
    puts(&Salida[11]);
    puts("de uso de la funcion puts.");
    return 0;
}
```

Solución

Esto es un ejemplo
de declaracion de constante.
Salta dos lineas

y tambien un
ejemplo
de uso de la funcion puts.

- 3.5. ¿Cuál es la salida del siguiente programa?.

```
#include <stdio.h>
int main()
{
    char pax[] = "Juan Sin Miedo";
    printf( "%s %s\n",pax,&pax[4]);
    puts(pax);
    puts(&pax[4]);
    return 0;
}
```

Solución

```
Juan Sin Miedo Sin Miedo
Juan Sin Miedo
Sin Miedo
```

- 3.6.** *Escriba y ejecute un programa que escriba su nombre y dirección.*

Codificación

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

{
    printf( " Lucas Sánchez García\n");
    printf( " Calle Marquillos de Mazarambroz, 2\n");
    printf( " Mazarambroz, TOLEDO\n");
    printf( " Castilla la Mancha, ESPAÑA\n");
    return 0;
}
```

- 3.7.** *Escribir y ejecutar un programa que escriba una página de texto con no más de 50 caracteres por línea.*

Codificación (Consultar en la página web del libro)

- 3.8.** *Depurar el siguiente programa.*

```
#include <stdio.h>
void main()
{
    printf("El lenguaje de programación C)
```

Solución

A la codificación anterior le falta en la orden `printf`, terminar con `\n`. Además el programa debe terminar con `}`.

El programa depurado es el siguiente:

```
#include <stdio.h>
void main()
{
    printf(" El lenguaje de programación C\n");
}
```

- 3.9.** *Escriba un programa que escriba la letra B con asteriscos.*

Codificación

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



```

printf("*      *\n");
printf("*      *\n");
printf("*****\n");
printf("*      *\n");
printf("*      *\n");
printf("*      *\n");
printf("*****\n");
}

```

3.10. *Escriba un programa C que lea las iniciales de su nombre y primer apellido y las escriba en pantalla seguidas de un punto.*

Análisis del problema

Se declaran dos variables de tipo `char` una para leer la inicial del nombre y otra para leer la inicial del primer apellido.

Codificación

```

#include <stdio.h>
int main()
{
    char n, a;
    printf("Introduzca la inicial de su nombre y su apellido: ");
    scanf("%c %c",&n,&a);
    printf("Hola, %c . %c .\n",n,a);
    return 0;
}

```

3.11. *Escriba un programa que lea una variable entera y cuatro reales y las escriba en pantalla.*

Codificación

```

#include <stdio.h>
int main()
{
    int v1 ;
    float v2,precio, b,h;
    printf("Introduzca v1 y v2: ");
    scanf("%d %f",&v1,&v2);           /*lectura valores v1 y v2 */
    printf("valores leídos: %d %f\n", v1,v2);
    printf("Precio de venta al público\n");
    scanf("%f",&precio);              /*lectura de precio */
    printf("Precio de venta %f\n", precio);
    printf("Base y altura: ");        /*lectura de base y altura */
    scanf("%f %f\n",&b,&h);
    printf("Base y altura %f %f\n", b, h);
    return 0;
}

```

3.12. *Escriba un programa que lea la base y la altura de un cilindro y las presente en pantalla.*

Codificación

```

#include <stdio.h>

```

```
int main()
{
    float base, altura;
    printf("Introduzca base: ");
    scanf("%f",&base);
    printf("Introduzca altura\n");
    scanf("%f",&altura);
    printf("Base leída %f\n", base);
    printf("Altura leída %f ", altura);
    return 0;
}
```

PROBLEMAS PROPUESTOS

- 3.1. Escribir y depurar un programa que visualice la letra A con asteriscos.
- 3.2. Escribir un programa que lea un texto de cinco líneas y lo presente en pantalla.
- 3.3. Escribir un programa que lee 5 número enteros y tres números reales y los escriba.
- 3.4. Escribir y ejecutar un programa que lea su nombre y dirección y lo presente.
- 3.5. ¿Cuál es la salida del siguiente programa?

```
#include <stdio.h>
int main()
{
    char p[] = "Esto es una prueba";
    printf("%s %s\n",p, &p[2]);
    puts(p);
    puts(&p[2]);
    return 0;
}
```

- 3.6. Depurar el siguiente programa

```
#include <stdio.h>
void main()
{
    printf("Esto es un ejemplo);
}
```

- 3.7. Escribir un programa que presente en pantalla los 5 primeros números impares.
- 3.8. Escribir un programa que lea la base y la altura de un trapecio y calcule su área y la presente en pantalla.
- 3.9. Escribir un programa que calcule lea el radio de una circunferencia y calcule su perímetro.
- 3.10. Realizar un programa que lea tres números reales y escriba su suma y su producto.

Operadores y expresiones

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Este capítulo muestra como C hace uso de los operadores y expresiones para la resolución de operaciones. Los operadores fundamentales que se analizan en el capítulo son:

- aritméticos, lógicos y relacionales;
- de manipulación de bits;
- condicionales;
- especiales.

Además se analizarán las conversiones de tipos de datos y las reglas que seguirá el compilador cuando concurren en una misma expresión diferentes tipos de operadores. Estas reglas se conocen como *prioridad* y *asociatividad*.

4.1 Operadores y expresiones

Los programas C constan de datos, sentencias de programas y expresiones. Una *expresión* es, una sucesión de operadores y operandos debidamente relacionados para formar expresiones matemáticas que especifican un cálculo.

4.2 El operador de asignación

El operador de asignación tiene la siguiente sintaxis:

```
variable = expresión
```

donde *variable* es un identificador válido de C declarado como *variable*. El operador = asigna el valor de la expresión derecha a la variable situada a su izquierda. Este operador es asociativo por la derecha, eso permite realizar asignaciones múltiples. Así, `a = b = c = 10;` equivale a que las variables *a*, *b* y *c* se asigna el valor 10. Esta propiedad permite inicializar varias variables con una sola sentencia. Además del operador de asignación =, C proporciona cinco operadores de asignación adicionales dados en la Tabla 4.1.

Tabla 4.1 Operadores de asignación de C.

Símbolo	Uso	Descripción
=	$a = b$	Asigna el valor de b a a .
*=	$a *= b$	Multiplica a por b y asigna el resultado a la variable a .
/=	$a /= b$	Divide a entre b y asigna el resultado a la variable a .
%=	$a \% = b$	Fija a al resto de a/b .
+=	$a += b$	Suma b y a y lo asigna a la variable a .
-=	$a -= b$	Resta b de a y asigna el resultado a la variable a .

EJEMPLO 4.1 El siguiente fragmento de programa asigna a las variables de la izquierda los valores de la derecha.

```
codigo = 3467
fahrenheit = 123.456;
coordX = 525;
coordY = 725;
```

4.3 Operadores aritméticos

Los operadores aritméticos sirven para realizar operaciones aritméticas básicas. Estos operadores vienen recogidos en la Tabla 4.2 y su prioridad y asociatividad en la Tabla 4.3

Tabla 4.2 Operadores aritméticos.

Operador	Tipos enteros	Tipos reales	Ejemplo
+	Suma	Suma	$x + y$
-	Resta	Resta	$b - c$
*	Producto	Producto	$x * y$
/	División entera: cociente	División en coma flotante	$b / 5$
%	División entera: resto		$b \% 5$

Los paréntesis se pueden utilizar para cambiar el orden usual de evaluación de una expresión determinada por su *prioridad* y *asociatividad*.

Tabla 4.3 Prioridad y asociatividad.

Prioridad (mayor a menor)	Asociatividad
+, - (<i>unitarios</i>)	izquierda-derecha (\rightarrow)
*, /, %	izquierda-derecha (\rightarrow)
+, -	izquierda-derecha (\rightarrow)

EJEMPLO 4.2 ¿Cuál es el resultado de la expresión: $7 * 10 - 5 \% 3 * 4 + 9$?

Los sucesivos pasos en el cálculo son los siguientes:

```
70 - 5% 3 * 4 + 9
70 - 2 * 4 + 9
70 - 8 + 9
62 + 9
71
```

4.4 Operadores de incrementación y decrementación

De las características que incorpora C, una de las más útiles son los operadores de incremento ++ y decremento -- dados en la Tabla 4.4. Estos operadores tienen la propiedad de que pueden utilizarse como sufijo o prefijo. El resultado de la expresión puede ser distinto, dependiendo del contexto.

```
m = n++;
m = ++n;.
```

En el primer caso, se realiza primeramente la asignación y después se incrementa en una unidad. En el segundo caso, se incrementa *n* en una unidad y posteriormente se realiza la asignación.

Tabla 4.4 Operadores de incrementación (++) y decrementación (--).

Incrementación	Decrementación
++n, n++	--n n--
n += 1	n -= 1
N = n + 1	n = n - 1

EJEMPLO 4.3 *Diferencias entre operadores de preincremento y postincremento.*

```
#include <stdio.h>

/* Test de operadores ++ y -- */

void main()
{
    int m = 99, n;
    n = ++m;
    printf("m = %d, n = %d\n",m,n);
    n = m++;
    printf("m = %d, n = %d\n",m,n);
    printf("m = %d \n",m++);
    printf("m = %d \n",++m);
}
```

EJECUCIÓN

```
m = 100, n = 100
m = 101, n = 100
m = 101
m = 103
```

4.5 Operadores relacionales

C no tiene tipos de datos lógicos. En su lugar se utiliza el tipo `int` para este propósito, con el valor entero 0 que representa a *falso* y distinto de cero a *verdadero*. Operadores tales como `>=` y `==` que comprueban una relación entre dos operandos se llaman **operadores relacionales** y se utilizan en expresiones de la forma:

expresión₁ *operador_relacional* *expresión₂*

expresión₁ y *expresión₂*
operador_relacional

expresiones compatibles C
un operador de la tabla siguiente

La Tabla 4.5 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: `char`, `int`, `float`, `double`, etc.

Tabla 4.5 Operadores relacionales de C.

Operador	Significado	Ejemplo
<code>==</code>	<i>Igual a</i>	<code>a == b</code>
<code>!=</code>	<i>No igual a</i>	<code>a != b</code>
<code>></code>	<i>Mayor que</i>	<code>a > b</code>
<code><</code>	<i>Menor que</i>	<code>a < b</code>
<code>>=</code>	<i>Mayor o igual que</i>	<code>a >= b</code>
<code><=</code>	<i>Menor o igual que</i>	<code>a <= b</code>

4.6 Operadores lógicos

Los *operadores lógicos* se utilizan con expresiones para devolver un valor *verdadero* (cualquier entero distinto de cero) o un valor *falso* (0). Los operadores lógicos de C son: `not (!)`, `and (&&)` y `or(||)`. El operador lógico `!` (*not*, *no*) produce *falso* (cero) si su operando es *verdadero* (distinto de cero) y viceversa. El operador lógico `&&` (*and*, *y*) produce *verdadero sólo* si ambos operandos son *verdadero* (no cero); si cualquiera de los operandos es *falso* produce *falso*. El operador lógico `||` (*or*, *o*) produce *verdadero* si cualquiera de los operandos es *verdadero* (distinto de cero) y produce *falso* sólo si ambos operandos son *falsos*. El operador `!` tiene prioridad más alta que `&&`, que a su vez tiene mayor prioridad que `||`. La asociatividad es de izquierda a derecha.

Evaluación en cortocircuito. En C los operandos de la izquierda de `&&` y `||` se evalúan siempre en primer lugar; si el valor del operando de la izquierda determina de forma inequívoca el valor de la expresión, el operando derecho no se evalúa. Esto significa que si el operando de la izquierda de `&&` es falso o el de `||` es verdadero, el operando de la derecha no se evalúa. Esta propiedad se denomina *evaluación en cortocircuito*.

4.7 Operadores de manipulación de bits

Los operadores de manipulación o tratamiento de bits (*bitwise*) ejecutan operaciones lógicas sobre cada uno de los bits de los operandos. Cada operador de manipulación de bits realiza una operación lógica bit a bit sobre datos internos. Los operadores de manipulación de bits se aplican sólo a variables y constantes `char`, `int` y `long`, y no a datos en coma flotante. La Tabla 4.6 recoge los operadores lógicos bit a bit.

Tabla 4.6 Operadores lógicos bit a bit.

Operador	Operación
<code>&</code>	y (and) <i>lógica bit a bit.</i>
<code> </code>	o (or) <i>lógica (inclusiva) bit a bit.</i>
<code>^</code>	o (xor) <i>lógica (exclusiva) bit a bit (or exclusive, xor).</i>
<code>~</code>	<i>Complemento a uno (inversión de todos los bits).</i>
<code><<</code>	<i>Desplazamiento de bits a izquierda.</i>
<code>>></code>	<i>Desplazamiento de bits a derecha.</i>

4.7.1 OPERADORES DE ASIGNACIÓN ADICIONALES

Los operadores de asignación abreviados están disponibles para operadores de manipulación de bits. Estos operadores vienen recogidos en la tabla 4.7

Tabla 4.7 Operadores de asignación adicionales.

Símbolo	Uso	Descripción
<<=	a <<= b	Desplaza a a la izquierda b bits y asigna el resultado a a.
>>=	a >>= b	Desplaza a a la derecha b bits y asigna el resultado a a.
&=	a &= b	Asigna a a el valor a&b.
^=	a ^= b	Establece a a a^b.
=	a = b	Establece a a a b.

4.7.2 OPERADORES DE DESPLAZAMIENTO DE BITS (>>, <<)

Efectúa un desplazamiento a la derecha (>>) o a la izquierda (<<) de n posiciones de los bits del operando, siendo n un número entero. Los formatos de los operadores de desplazamiento son:

```
1. valor << numero_de_bits;
2. valor >> numero_de_bits;
```

El valor puede ser una variable entera o carácter, o una constante. El número_de_bits determina cuántos bits se desplazaran.

4.7.3 OPERADORES DE DIRECCIONES

Los operadores recogidos en la Tabla 4.7 permiten manipular las direcciones de las variables y objetos en general.

Tabla 4.7 Operadores de direcciones.

Operador	Acción
*	Lee o modifica el valor apuntado por la expresión. Se corresponde con un puntero y el resultado es del tipo apuntado.
&	Devuelve un puntero al objeto utilizado como operando, que debe ser un lvalue (variable dotada de una dirección de memoria). El resultado es un puntero de tipo idéntico al del operando.
.	Permite acceder a un miembro de un objeto agregado (unión, estructura).
->	Accede a un miembro de un objeto agregado (unión, estructura) apuntado por el operando de la izquierda.

4.8 Operador condicional

El operador condicional ?, es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene asociatividad a derechas. El formato del operador condicional es:

```
expresion_C ? expresion_v : expresion_f;
```

Se evalúa expresion_C y su valor (cero es falso, distinto de cero es verdadero) determina cuál es la expresión a ejecutar; si la condición es verdadera se ejecuta expresion_v y si es falsa se ejecuta expresion_f . La precedencia de ? es menor que la de cualquier otro operando tratado hasta ese momento. Su asociatividad es a derecha.

EJEMPLO 4.4 Usos del operador condicional `?` :

```

n >= 0 ? 1 : -1           /* 1 si n es positivo, -1 si es negativo */
m >= n ? m : n           /* devuelve el mayor valor de m y n */

```

4.9 Operador coma ,

El *operador coma* permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha. La expresión más a la derecha determina el resultado global. El uso del operador coma es como sigue:

```
expresión1, expresión2, expresión3, ..., expresión
```

Cada expresión se evalúa comenzando desde la izquierda y continuando hacia la derecha.

EJEMPLO 4.5 Se concatenan expresiones con el operador coma `(,)`.

```

int i, j, resultado;
int i;
resultado = j = 10, i = j, ++i;

```

En primer lugar, a *j* se asigna el valor 10, a continuación a *i* se asigna el valor de *j*. Por último, *i* se incrementa a 11. El valor de esta expresión y, por tanto el valor asignado a *resultado* es 11.

4.10 Operadores especiales `()`, `[]`

C admite algunos operadores especiales que sirven para propósitos diferentes. Cabe destacar `()`, `[]`.

El operador `()` es el operador de llamada a funciones. Sirve para encerrar los argumentos de una función, efectuar conversiones explícitas de tipo, indicar en el seno de una declaración que un identificador corresponde a una función, resolver los conflictos de prioridad entre operadores.

El operador `[]` sirve para dimensionar los arrays y designar un elemento de un array o una matriz.

4.11 El operador `sizeof`

C proporciona el operador `sizeof`, que toma un argumento, bien un tipo de dato o bien el nombre de una variable (escalar, array, registro, etc.), y obtiene como resultado el número de bytes que ocupa. El formato del operador es:

```

sizeof(nombre_variable)
sizeof(tipo_dato)
sizeof(expresión)

```

4.12 Conversiones de tipos

Las conversiones de tipos pueden ser *implícitas* (ejecutadas automáticamente) o *explícitas* (solicitadas específicamente por el programador).

Conversión implícita. C hace muchas conversiones de tipos automáticamente. Convierte valores cuando se asigna un valor de un tipo a una variable de otro tipo; C convierte valores cuando se combinan tipos mixtos en expresiones; C convierte valores cuando se pasan argumentos a funciones. Los tipos fundamentales (básicos) pueden ser mezclados libremente en asignaciones y expresiones. Las conversiones se ejecutan automáticamente: los operandos de tipo más bajo se convierten a los de tipo más alto de acuerdo con las siguientes reglas: si cualquier operando es de tipo `char`, `short` o *enumerado* se convierte en tipo `int`; si los operandos tienen diferentes tipos, la siguientes lista determina a qué operación convertirá. Esta operación se llama *promoción integral*.

int, unsigned int, long, unsigned long , float, double

El tipo que viene primero, en esta lista, se convierte en el que viene segundo.

Conversiones explícitas. C fuerza la conversión explícita de tipos mediante el operador de *molde* (*cast*). El operador *molde* tiene el formato: *(tiponombre)valor*. Convierte *valor* a *tiponombre*. El operador *molde* (*tipo*) tiene la misma prioridad que otros operadores unitarios tales como +, - y !.

4.13 Prioridad y asociatividad

La prioridad o precedencia de operadores determina el orden en el que se aplican los operadores a un valor. Los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente:

- Si dos operadores se aplican al mismo operando, el operador con mayor prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad.
- La asociatividad izquierda-derecha significa aplicar el operador más a la izquierda primero, y en la asociatividad derecha-izquierda se aplica primero el operador más a la derecha.
- Los paréntesis tienen la máxima prioridad.

La prioridad de los operadores viene indicada en la Tabla 4.8

Tabla 4.8 Prioridad de los operadores.

Prioridad	Operadores	Asociatividad
1	_ -> [] ()	I – D
2	++ -- ~ ! - + & * sizeof	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		I – D
14	?: (expresión condicional)	D – I
15	= *= /= %= += -= <<= >>= &= = ^=	D – I
16	, (operador coma)	I – D

I - D : Izquierda – Derecha.
D - I : Derecha – Izquierda.

PROBLEMAS RESUELTOS

4.1. Determinar el valor de las siguientes expresiones aritméticas.

$$\begin{array}{ll} 15 / 12 & 15 \% 12 \\ 24 / 12 & 24 \% 12 \\ 123 / 100 & 123 \% 100 \\ 200 / 10 & 200 \% 100 \end{array}$$

Solución

$$\begin{array}{ll} 15 / 12 = 1 & 15 \% 12 = 3 \\ 24 / 12 = 2 & 24 \% 12 = 0 \\ 123 / 100 = 1 & 123 \% 100 = 23 \\ 200 / 100 = 2 & 200 \% 100 = 0 \end{array}$$

4.2. ¿Cuál es el valor de cada una de las siguientes expresiones?.

$$\begin{array}{l} a) 15 * 14 - 3 * 7 \\ b) -4 * 5 * 2 \\ c) (24 + 2 * 6) / 4 = 9 \\ d) 3 + 4 * (8 * (4 - (9 + 3) / 6)) = 67 \\ e) 4 * 3 * 5 + 8 * 4 * 2 - 5 = 119 \\ f) 4 - 40 / 5 = -4 \\ g) (-5) \% (-2) = -1 \end{array}$$

Solución

$$\begin{array}{l} a) 15 * 14 - 3 * 7 = 189 \\ b) -4 * 5 * 2 = -40 \\ c) (24 + 2 * 6) / 4 = 9 \\ d) 3 + 4 * (8 * (4 - (9 + 3) / 6)) = 67 \\ e) 4 * 3 * 5 + 8 * 4 * 2 - 5 = 119 \\ f) 4 - 40 / 5 = -4 \\ g) (-5) \% (-2) = -1 \end{array}$$

4.3. Escribir las siguientes expresiones aritméticas como expresiones de computadora: La potencia puede hacerse con la función `pow()`, por ejemplo $(x + y)^2 == \text{pow}(x+y, 2)$.

$$\begin{array}{lll} a) \frac{x}{y} + 1 & d) \frac{b}{c + d} & g) \frac{xy}{1 - 4x} \\ b) \frac{x + y}{x - y} & e) (a + b) \frac{c}{d} & h) \frac{xy}{mn} \\ c) x + \frac{y}{z} + 1 & i) [(a + b)^2]^2 & j) (x + y)^2 * (a - b) \end{array}$$

Solución

$$\begin{array}{l} a) x / y + 1 \\ b) (x + y) / (x - y) \end{array}$$

- c) $x + y / z$
- d) $b / (c + d)$
- e) $(a + b) * (c / d)$
- f) $\text{pow}(\text{pow}(x + y, 2), 2)$
- g) $x * y / (1 - 4 * x)$
- h) $x * y / (m * n)$
- i) $\text{pow}(x + y, 2) * (a - b)$

4.4. ¿Cuál de los siguientes identificadores son válidos?

<i>N</i>	<i>85 Nombre</i>
<i>MiProblema</i>	<i>AAAAAAAAAA</i>
<i>Mi Juego</i>	<i>Nombre_Apellidos</i>
<i>MiJuego</i>	<i>Saldo_Actual</i>
<i>write</i>	<i>92</i>
<i>m&m</i>	<i>Universidad Pontificia</i>
<i>registro</i>	<i>Set 15</i>
<i>A B</i>	<i>* 143Edad</i>

Solución

<i>N</i>	Correcto
<i>MiProblema</i>	Correcto
<i>Mi Juego</i>	Incorrecto (lleva espacio en blanco)
<i>MiJuego</i>	Correcto (lleva espacio en blanco)
<i>write</i>	Correcto
<i>m&m</i>	Incorrecto
<i>registro</i>	Correcto
<i>A B</i>	Correcto
<i>85 Nombre</i>	Incorrecto (comienza por número)
<i>AAAAAAAAAA</i>	Correcto
<i>Nombre_Apellidos</i>	Correcto
<i>Saldo_Actual</i>	Correcto
<i>92</i>	Incorrecto (número)
<i>Universidad Pontificia</i>	Incorrecto (lleva espacio en blanco)
<i>Set 15</i>	Incorrecto (lleva espacio en blanco)
<i>*143Edad</i>	Incorrecto (no puede comenzar por *)

4.5. Si *x* es una variable entera e *y* una variable carácter. ¿Qué resultados producirá la sentencia `scanf("%d %c", &x, &y)` si la entrada es?

- a) *5 c*
- b) *5C*

Solución

Se asigna a la variable *x* el valor 5 y a la variable *y* el carácter *c* o el *C* mayúscula. Si se ejecuta el programa siguiente:

```
#include <stdio.h>
int main()
{
    int x;
    char y ;
    scanf("%d %c",& x,  & y);
```

```
printf("%d %d %c", x, y, y);
return 0;
}
```

para el caso *a* la salida es 5 99 c (99 es el valor ASCII del carácter c)

para el caso *b* la salida es 5 67 C (67 es el valor ASCII del carácter C)

- 4.6.** Realizar un programa que lea un entero, lo multiplique por 2 y a continuación lo escriba de nuevo en la pantalla.

Codificación

```
#include <stdio.h>
int main()
{
    int x;
    printf(" dame un numero entero\n");
    scanf("%d", &x);
    x = 2 * x;
    printf("su doble es %d", x);
    return 0;
}
```

- 4.7.** Realizar un programa que solicite al usuario la longitud y anchura de una habitación y a continuación visualice su superficie con cuatro decimales.

Codificación

```
#include <stdio.h>
int main()
{
    float x,y;
    printf(" dame la longitud de la habitacion\n");
    scanf("%f",&x);
    printf(" dame la anchura de la habitacion\n");
    scanf("%f",&y);
    printf("su superficie es %10.4f", x * y);
    return 0;
}
```

- 4.8.** ¿Cuáles son los resultados visualizados por el siguiente programa, si los datos proporcionados son 5 y 8 ?.

```
#include <stdio.h>
const int M = 6;
int main()
{
    int a, b, c;
    puts("Introduce el valor de a y de b");
    scanf("%d %d",&a,&b);
    c = 2 * a - b;
    c -= M;
    b = a + c - M;
    a = b * M;
```

```

printf("\n a = %d\n",a);
b = - 1;
printf(" b= %6d c= %6d",b,c);
return 0;
}

```

Solución

Los valores visualizados son:

a = -30

b = -1 c = -4

4.9. Un sistema de ecuaciones lineales

$$ax + by = c$$

$$dx + ey = f$$

se puede resolver con las siguientes fórmulas :

$$x = \frac{ce - bf}{ae - bd} \quad y = \frac{af - cd}{ae - bd}$$

Diseñar un programa que lea dos conjuntos de coeficientes (a, b, c, d, e, f) y visualice los valores de x e y.

Codificación

```

#include <stdio.h>
int main()
{
    float a, b, c,d,e,f,denominador,x,y;
    puts("Introduce el valor de a de b y de c");
    scanf("%f %f %f",&a,&b,&c);
    puts("Introduce el valor de d de e y de f");
    scanf("%f %f %f",&d,&e,&f);
    denominador= a*e-b*d;
    if (denominador==0)
        printf(" no solucion\n");
    else
    {
        x = (c * e - b * f) / denominador;
        y = (a * f - c * d) / denominador;
        printf( "la solucion es\n");
        printf( "%f %f \n", x, y);
    }
    return 0;
}

```

4.10. Teniendo como datos de entrada el radio y la altura de un cilindro, calcular el área total y el volumen del cilindro.

Análisis del problema

Sabiendo que el área total de un cilindro es igual a la suma de las áreas de los dos círculos mas el área del rectángulo formado por la longitud de la circunferencia y la altura del cilindro, y que el volumen se obtiene multiplicando la superficie de la base del círculo por la altura del cilindro, el siguiente programa realiza las tareas solicitadas:

Codificación

```
#include <stdio.h>
const float pi = 3.141592;
int main()
{
    float base, altura, area, volumen;
    puts("Introduce el valor de la base y la altura");
    scanf("%f %f",&base,&altura);
    if ((base <= 0)|| (altura <= 0))
        printf(" no solucion\n");
    else
    {
        area = 2 * pi * base * base + 2 * pi * base * altura;
        volumen = pi * base * base * altura;
        printf("la solucion es\n");
        printf("area total = %f \n", area);
        printf("volumen = %f \n", volumen);
    }
    return 0;
}
```

4.11. Calcular el área de un triángulo mediante la fórmula :

$$\text{Área} = \sqrt{p(p-a)(p-b)(p-c)}$$

donde p es el semiperímetro, $p = (a + b + c)/2$, y a, b, c los tres lados del triángulo.

Análisis del problema

Para que el triángulo exista debe cumplirse que los lados sean todos positivos, y además que la suma de dos lados cualesquiera sea mayor que el otro lado. El programa que se codifica comprueba que los datos leídos cumplen las condiciones.

Codificación

```
#include <stdio.h>
#include <math.h>
int main()
{
    float a,b,c,p,area;
    puts("Introduce el valor de los tres lados");
    scanf("%f %f %f",&a, &b, &c);
    if ((a <= 0) || (b <= 0) || (c <= 0) ||
        ((a + b) < c) || ((a + c) < b) || ((b + c) < a))
        printf(" no solucion\n");
    else
    {
        p =(a + b + c)/ 2;
        area = pow(p * ( p - a ) * ( p - b ) * ( p - c ) , 0.5);
        printf( "la solucion es\n");
        printf( "area = %f \n", area);
    }
    return 0;
}
```

- 4.12.** Construir un programa para obtener la hipotenusa y los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.

Análisis del problema

Se calcula la hipotenusa por la fórmula del teorema de Pitágoras, y se obtiene el ángulo mediante la función inversa del seno que es `asin()`. Además se convierte el valor devuelto por la función arco seno a grados (la función arco seno da su resultado en radianes).

Codificación

```
#include <stdio.h>
#include <math.h>
int main()
{
    const float pi = 3.141592;
    float cateto1, cateto2, hipotenusa, angulo;
    puts("Introduce el valor de los catetos");
    scanf("%f %f",&cateto1, &cateto2);
    if (( cateto1 <= 0 ) || ( cateto2 <= 0 ))
        printf(" no solucion\n");
    else
    {
        hipotenusa = sqrt( cateto1 * cateto1 + cateto2 * cateto2 );
        angulo = 180 / pi * asin( cateto1 / hipotenusa);           /* ángulo en grados */
        printf( "la solucion es\n");
        printf( "hipotenusa = %f \n", hipotenusa);
        printf("      angulo   = %f °\n", angulo);
        printf(" otro angulo  = %f °\n", 90 - angulo);
    }
    return 0;
}
```

- 4.13.** La fuerza de atracción entre dos masas, m_1 y m_2 separadas por una distancia d , está dada por la fórmula:

$$F = \frac{G * m_1 * m_2}{d^2}$$

donde G es la constante de gravitación universal, $G = 6.673 \times 10^{-8} \text{ cm}^3/\text{g} \cdot \text{seg}^2$

Escriba un programa que lea la masa de dos cuerpos y la distancia entre ellos y a continuación obtenga la fuerza gravitacional entre ella. La salida debe ser en dinas; un dina es igual a $\text{gr} \cdot \text{cm}/\text{seg}^2$.

Análisis del problema

Simplemente hay que leer las dos masas y aplicar la fórmula correspondiente.

Codificación.

```
#include <stdio.h>
#include <math.h>
int main()
{
    const float G = 6.673e-8;
```



```

float masa1, masa2, distancia, fuerza;
puts(" dame la masa de los dos cuerpos en gramos\n");
scanf("%f %f", &masa1, &masa2);
puts(" dame la distancia entre ellos en centimetros\n");
scanf("%f", &distancia);
if (( masa1 <= 0 ) || ( masa2 <= 0 ) || ( distancia <= 0 ))
    printf(" no solucion\n");
else
{
    fuerza = G * masa1 * masa2 / (distancia * distancia);
    printf( "la solucion es\n");
    printf( "Fuerza en dinas = %f \n", fuerza);
}
return 0;
}

```

- 4.14.** *Escribir un programa que lea dos enteros y calcule e imprima su producto, cociente y el resto cuando el primero se divide por el segundo.*

Análisis del problema

Se leerán en dos variables enteras los datos y se calcularán en las variables producto, cociente y resto los resultados.

Codificación

```

#include <stdio.h>
int main()
{
    int a, b, producto, cociente, resto;
    puts(" introduzca dos numeros\n");
    scanf("%d %d", &a, &b);
    producto = a * b;
    cociente = a / b;
    resto = a % b;
    printf(" producto = %d\n", producto);
    printf(" cociente = %d\n", cociente);
    printf(" resto = %d\n", resto);
    return 0;
}

```

- 4.15.** *Escribir un programa C que lea dos números y visualice el mayor.*

Codificación

```

#include <stdio.h>
int main()
{
    int x, y, Mayor;
    puts(" introduzca dos numeros\n");
    scanf("%d %d", &x, &y);
    Mayor = x;
    if (x < y)
        Mayor = y;
}

```

```
printf(" el mayor es %d\n", Mayor);  
return 0;  
}
```

- 4.16.** *Escribir un programa en el que se introducen como datos de entrada la longitud del perímetro de un terreno, expresada con tres números enteros que representan hectómetros, decámetros y metros respectivamente. Se ha de escribir, con un rótulo representativo, la longitud en decímetros.*

Análisis del problema

El programa que se codifica lee los hectómetros, decámetros y metros y realiza las conversiones correspondientes.

Codificación

```
#include <stdio.h>  
int main()  
{  
    int hectometros, decametros, metros, decimetros;  
    printf("Introduzca hectometros, decametros y metros ");  
    scanf("%d %d %d",&hectometros, &decametros, &metros);  
    decimetros = ((hectómetros * 10 + decametros) * 10 + metros)*10;  
    printf (" numero de decimetros es %d \n", decimetros);  
    return 0;  
}
```

- 4.17.** *Escribir un programa que desglose cierta cantidad de segundos introducida por teclado en su equivalente en semanas, días, horas, minutos y segundos.*

Análisis del problema

El programa que se codifica lee el número de segundos y realiza las conversiones, teniendo en cuenta que un día tiene 24 horas, una hora 60 minutos, y un minuto 60 segundos.

Codificación

```
#include <stdio.h>  
int main()  
{  
    int semanas, dias, horas, minutos, segundos, acu;  
    printf("Introduzca segundos ");  
    scanf("%d",&acu);  
    segundos = acu % 60;  
    acu = acu / 60;  
    minutos = acu % 60;  
    acu = acu / 60;  
    horas = acu % 24;  
    acu = acu / 24;  
    dias = acu % 7;  
    semanas = acu / 7;  
    printf (" numero de segundos %d\n", segundos);  
    printf (" numero de minutos %d\n", minutos);  
    printf (" numero de horas %d\n", horas);  
    printf (" numero de dias %d\n", dias);  
}
```

```
printf (" numero de semanas %d\n", semanas);
return 0;
}
```

- 4.18.** La famosa ecuación de Einstein para conversión de una masa m en energía viene dada por la fórmula: $E = cm^3$, c es la velocidad de la luz y su valor es: $c = 2.997925 \times 10^{10} \text{ m/sg}$.

Escribir un programa que lea una masa en gramos y obtenga la cantidad de energía producida cuando la masa se convierte en energía.

Nota: Si la masa se da en gramos, la fórmula produce la energía en ergios.

Codificación

```
#include <stdio.h>
int main()
{
    float m, energia;
    const float c = 2.997925e+10;
    puts(" introduzca masa\n");
    scanf("%f", &m);
    energia = c * m * m * m;
    printf(" energia en ergios %e\n", energia);
    return 0;
}
```

- 4.19.** Diseñar un programa que permita convertir una medida dada en pies a sus equivalentes en: a) yardas, b) pulgadas, c) centímetros y d) metros (1 pie = 12 pulgadas, 1 yarda = 3 pies, 1 pulgada = 2.54 cm. 1 m = 100 cm). Leer el número de pies e imprimir el número de yardas, pies, pulgadas, centímetros y metros.

Análisis del problema

El programa leerá el número de pies y realizará las transformaciones correspondientes de acuerdo con las equivalencias.

Codificación

```
#include <stdio.h>
int main()
{
    float pies, pulgadas, yardas, metros, centimetros;
    puts(" introduzca pies\n");
    scanf("%f", &pies);
    pulgadas = pies * 12;
    yardas = pies / 3;
    centimetros = pulgadas * 2.54;
    metros = centimetros / 100;
    printf(" pies %f \n", pies);
    printf(" pulgadas %f\n", pulgadas);
    printf(" yardas %f\n", yardas);
    printf(" centimetros %f\n", centimetros);
    printf(" metros %f\n", metros);
    return 0;
}
```

4.20. Escriba un programa que lea cuatro números enteros y nos calcule su media.

Análisis del problema

Se leerán los cuatro números enteros y se calculará la media en una variable de tipo real, obligando a que el cociente sea un número real.

Codificación

```
#include <stdio.h>
int main()
{
    int n1, n2, n3, n4;
    float media;
    puts(" introduzca los cuatro números\n");
    scanf("%d %d %d %d", &n1, &n2, &n3, &n4);
    media = (n1 + n2 + n3 + n4) / 4.0;
    printf(" la media es %f \n", pmedia);
    return 0;
}
```

PROBLEMAS PROPUESTOS

- 4.1.** Una temperatura Celsius (centígrados) puede ser convertida a una temperatura equivalente F de acuerdo a la siguiente fórmula :

$$F = \left(\frac{9}{5} \right) c + 32$$

Escribir un programa que lea la temperatura en grados Celsius y la escriba en F.

- 4.2.** Realizar un programa que lea la hora de un día de notación de 24 horas y la respuesta en notación de 12 horas. Por ejemplo, si la entrada es 13:45, la salida será:

1: 45 PM

El programa pedirá al usuario que introduzca exactamente cinco caracteres. Por ejemplo, las nueve en punto se introduce como

09:00

- 4.3.** Realizar un programa que determine si un año es bisiestro. Un año es bisiestro si es múltiplo de 4 (por ejemplo 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestros cuando a la vez son múltiplos de 400 (por ejemplo, 1800 no es bisiestro, mientras que 2000 si lo es).

- 4.4.** Construir un programa que indique si un número introducido por teclado es positivo, igual a cero, o negativo, utilizar para hacer la selección el operador ?.

- 4.5.** Implementar un programa que lea tres números y escriba el mayor y el menor.

- 4.6.** Implementar un programa que lea tres números y calcule la media.

- 4.7.** Implementar un programa que lea el radio de un círculo y calcule su área, así como la longitud de la circunferencia de ese radio.

- 4.8.** Implementar un programa que lea el radio y la altura de un cono y calcule su volumen y área total.

- 4.9.** Implementar un programa que lea tres enteros de tres dígitos y calcule e imprima su suma y su producto. La salida será justificada a derecha

- 4.10.** Implementar un programa que lea 3 números y si el tercero es positivo calcule y escriba la suma de los tres números, y si es negativo calcule y escriba su producto.

Estructuras de selección: sentencias `if` y `switch`

Los programas definidos hasta este punto se ejecutan de modo secuencial, es decir, una sentencia después de otra. La ejecución comienza con la primera sentencia de la función y prosigue hasta la última sentencia, cada una de las cuales se ejecuta una sola vez. Esta forma de programación es adecuada para resolver problemas sencillos. Sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se ejecutan y en qué momentos. Las *estructuras* o *construcciones de control* controlan la secuencia o flujo de ejecución de las sentencias. Las estructuras de control se dividen en tres grandes categorías en función del flujo de ejecución: *secuencia*, *selección* y *repetición*.

Este capítulo considera las *estructuras selectivas* o *condicionales* –sentencias `if` y `switch`– que controlan si una sentencia o lista de sentencias se ejecutan en función del cumplimiento o no de una condición.

5.1 Estructuras de control

Las **estructuras de control** controlan el flujo de ejecución de un programa o función. Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecución: *secuencia*, *selección (decisión)* y *repetición*.

5.2 La sentencia `if` con una alternativa

La sentencia `if` tiene dos alternativas o formatos posibles. El formato más sencillo tiene la sintaxis siguiente:

```
if (Expresión lógica) Sentencia
```

La sentencia `if` funciona de la siguiente manera. Si *Expresión* es *verdadera*, se ejecuta *Acción (Sentencia)*; en caso contrario no se ejecuta *Acción (Sentencia)*.

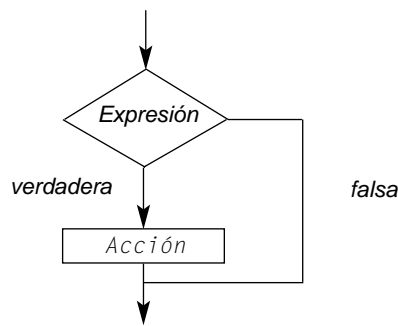


Figura 5.1 Diagrama de flujo de una sentencia básica if.

EJEMPLO 5.1 Prueba de divisibilidad.

```

int main()
{
    int n, d;
    printf( "Introduzca dos enteros: ");
    scanf("%d %d",&n,&d);
    if (n%d == 0) printf(" %d es divisible por %d\n",n,d);
    return 0;
}
  
```

EJECUCIÓN

```

Introduzca dos enteros: 24 4
24 es divisible por 4
  
```

5.3 Sentencia if de dos alternativas: if-else

Este formato de la sentencia if tiene la siguiente sintaxis:

```
if (expresión) Acción1 else Acción2
```

Cuando se ejecuta la sentencia if-else, se evalúa *Expresión*. Si *Expresión* es *verdadera*, se ejecuta *Acción1* y en caso contrario se ejecuta *Acción2*.

Una sentencia if es anidada cuando la sentencia de la rama *verdadera* o la rama *falsa*, es a su vez una sentencia if. Una sentencia if anidada se puede utilizar para implementar decisiones con varias alternativas o multi-alternativas.

Sintaxis:

```

if (condición1)
    sentencia1;
else if (condición2)
    sentencia2;
.
.
.
else if (condiciónn)
    sentencia_n;
else
    sentencia;
  
```

EJEMPLO 5.2 Calcular el mayor de dos números leídos del teclado y visualizarlo en pantalla.

```
#include <stdio.h>
int main()
{
    int x, y;
    printf( "Introduzca dos enteros: ");
    scanf("%d %d",&x,&y);
    if ( x > y)
        printf("%6d\n", x);
    else
        printf("%6d\n", y);
    return 0;
}
```

EJECUCIÓN

```
Introduzca dos enteros: 17 54
                    54
```

5.4 Sentencia de control `switch`

La sentencia `switch` es una sentencia C que se utiliza para hacer una selección entre múltiples alternativas. La expresión *selector* debe ser un tipo ordinal (*int*, *char*, ...). Cada *etiqueta* es un valor único, constante, y cada etiqueta debe tener un valor diferente de los otros. La expresión de control o *selector* se evalúa. Si su valor es igual a una de las etiquetas *case* – por ejemplo, *etiqueta_i* – entonces la ejecución comenzará con la primera sentencia de la secuencia *secuencia_i* y continuará hasta que se encuentra el final de la sentencia de control `switch`, o hasta encontrar la sentencia `break`.

Sintaxis:

```
switch (selector)
{
    case etiqueta1 : sentencias1;
    case etiqueta2 : sentencias2;
    .
    .
    .
    case etiquetan : sentenciasn;
    default:          sentencias;                               /* opcional */
}
```

5.5 Expresiones condicionales: el operador `?:`

Una expresión condicional tiene el formato `C ? A : B` y es realmente una operación ternaria (tres operandos) en la cual *C*, *A* y *B* son los tres operandos y `?:` es el operador.

Sintaxis:

```
condición ? expresión1 : expresión2
```


condición es una expresión lógica
expresión₁/expresión₂ son expresiones compatibles de tipos

La expresión `primerose evalúa condición`, si el valor de *condición* es *verdadera* (distinto de cero) entonces se devuelve como resultado el valor de *expresión₁*; si el valor de *condición* es *falsa* (cero) se devuelve como resultado el valor de *expresión₂*.

EJEMPLO 5.3 Se utiliza una expresión condicional para llamar, alternativamente a `f1(x)` o a `f2(x)`.

```
x == y ? f1(x) : f(x,y);
```

es equivalente a la siguiente sentencia:

```
if ( x == y )
    f1( x );
else
    f2( x , y );
```

5.6 Evaluación en cortocircuito de expresiones lógicas

La *evaluación en cortocircuito* de una expresión lógica significa que se puede detener la evaluación de una expresión lógica tan pronto como su valor pueda ser determinado con absoluta certeza. C realiza evaluación en cortocircuito con los operadores `&&` y `||`, de modo que evalúa primero la expresión más a la izquierda, de las dos expresiones unidas por `&&` o bien por `||`. Si de esta evaluación se deduce la información suficiente para determinar el valor final de la expresión (independiente del valor de la segunda expresión), el compilador de C no evalúa la segunda expresión. Esto permite disminuir en general el tiempo de ejecución. Por esta razón el orden de las expresiones con operadores `&&` y `||` puede ser crítico en determinadas situaciones.

PROBLEMAS RESUELTOS

5.1. ¿Qué errores de sintaxis tiene la siguiente sentencia?.

```
if x > 25.0
    y = x
else
    y = z;
```

Solución

La expresión correcta debe ser la siguiente:

```
if (x > 25.0 )
    y = x;
else
    y = z;
```

Por tanto le falta los paréntesis en la expresión lógica y un punto y coma después de la sentencia de asignación `y = x`.

5.2. ¿Qué valor se le asigna a `consumo` en la sentencia `if` siguiente si *velocidad* es 120?.

```
if (velocidad < 80)
    consumo = 10.00;
```

```

else if (velocidad > 100)
    consumo = 12.00;
else if (velocidad > 120)
    consumo = 15.00;

```

Solución

Si velocidad toma el valor de 120 entonces necesariamente consumo debe tomar el valor de 12.00.

5.3. ¿Cuál es el error del siguiente código?

```
if (x < y < z) printf("%d < %d < %d\n", x, y, z);
```

Solución

El error que presenta la sentencia es que `x < y < z` no es una expresión lógica. Debería haberse puesto `(x < y) && (y < z)`.

5.4. ¿Qué salida producirá el código siguiente, cuando se empotra en un programa completo y `primera_opcion` vale 1? ¿Y si `primera_opcion` vale 2?

```

int primera_opcion;

switch (primera_opcion + 1)
{
    case 1:
        puts("Cordero asado");
        break;
    case 2:
        puts("Chuleta lechal");
        break;
    case 3:
        puts("Chuletón");
    case 4:
        puts("Postre de pastel");
        break;
    default:
        puts("Buen apetito");
}

```

Solución

En el primer caso aparece escrito *Chuleta lechal*

En el segundo caso aparece escrito *Chuletón* y en la siguiente línea *Postre de pastel*, ya que `case 3:` no lleva la orden `break`.

5.5. Escribir una sentencia `if-else` que visualice la palabra *Alta* si el valor de la variable `nota` es mayor que 100 y *Baja* si el valor de esa `nota` es menor que 100.**Solución**

```

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

```

```

int nota;
printf(" dame nota: ");
scanf("%d", &nota);
if (nota < 100)
    printf(" Baja ");
else if (x > 100)m
    printf("Alta");
return 0;
}

```

5.6. Realizar un programa que determine el mayor de tres números.

Análisis del problema

Se realiza mediante un *algoritmo voraz*, de tal manera, que el mayor de un solo número es siempre el propio número. Si ya se tiene el mayor de una lista de números, y si a esa lista se le añade un nuevo número entonces el mayor o bien es el que ya se tenía, o bien es el nuevo.

Codificación

```

#include <stdio.h>
int main()
{
    int n1,n2,n3, mayor;
    puts(" introduzca tres nuumeros ");
    scanf(" %d %d %d", &n1, &n2,&n3);
    mayor = n1;
    if(mayor < n2)
        mayor = n2;
    if(mayor < n3)
        mayor = n3;
    printf(" el mayor es %d\n", mayor);
    return 0;
}

```

5.7. Escribir una sentencia *if-else* que clasifique un entero x en una de las siguientes categorías y escriba un mensaje adecuado:

$x < 0$ o bien $0 \leq x \leq 100$ o bien $x > 100$

Solución

```

#include <stdio.h>
int main()
{
    int x = 10;
    if (x < 0)
        printf("%d es negativo\n",x);
    else if (x <= 100)
        printf("0 <= x = %d <= 100\n", x);
    else
        printf("x = %d > 100\n", x);
    return 0;
}

```

- 5.8. Se trata de escribir un programa que clasifique enteros leídos del teclado de acuerdo a los siguientes criterios: si es 30 o mayor, o negativo, visualizar un mensaje en ese sentido; en caso contrario, si es un nuevo primo, potencia de 2, o un número compuesto, visualizar el mensaje correspondiente.

Análisis del problema

Se programa una función `primo()` que decide si un número es primo. Se inicializa una variable `d` a 2 y un bucle `while` incrementa `d` en una unidad cada vez que se ejecuta hasta encontrar un número `d` que divida al número que se quiere comprobar que es primo. El número será primo si el único divisor encontrado es el propio número. Se programa una función `p2()` que decide si un número es potencia de 2. Para comprobarlo se inicializa una variable `d` a 2, y *mientras* el número sea divisible por `d` lo que se hace es dividir el número por `d`. Por tanto el número cumplirá la condición de ser potencia de 2 si al final toma el valor uno.

Codificación

```
#include <stdio.h>

// función primo

int primo( int x)
{
    int d = 2;

    while ( x %d != 0)
        d++;
    return (d == x);
}

// función potencia de dos

int p2( int x)
{
    int d = 2;
    while ( x %d == 0)
        x = x / 2;
    return (x == 1);
}

// programa principal

int main()
{
    int x;
    puts(" introduzca numero entero \n");
    scanf(" %d ", &x);
    if (x < 0)
        printf("%d es negativo\n", x);
    else if (x > 30)
        printf(" x = %d > 30\n",x);
    elseif (primo(x))
    {
        if (x == 2)
            printf("x= %d es primo y potencia de dos\n",x);
        else
            printf("x = %d es primo y no potencia de dos \n", x);
    }
    else if (p2( x ))
        printf("x = %d es compuesto y potencia de dos\n", x);
}
```

```

    else
        printf("x = %d es compuesto y no potencia de dos\n",x);
    return 0;
}

```

- 5.9.** *Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestos cuando a la vez son múltiplos de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo es).*

Análisis del problema

Se diseña una función `bisiesto()` que decide si un número entero positivo representa a un año bisiesto.

Codificación

```

#include <stdio.h>
int bisiesto( int x);

int main()
{
    int x;
    puts(" introduzca año entero \n");
    scanf(" %d", &x);
    if (bisiesto(x))
        printf("%d es bisiesto\n", x);
    else
        printf (" %d no es un año bisiesto \n", x);
    return 0;
}

int bisiesto( int x)
{
    if (x % 400 ==0)
        return 1;
    else if (x % 100 == 0)
        return 0;
    else
        return (x % 4 == 0);
}

```

- 5.10.** *Escribir un programa que calcule el número de días de un mes, dados los valores numéricos del mes y el año.*

Análisis del problema

Se programa una función `ndd()` que calcula el número de días de un mes de un año concreto. La función llama a `bisiesto()` (ejercicio 5.9), que, como su nombre indica, determina si un año es bisiesto (Febrero, 29 días). El programa principal lee y valida tanto el dato `año` como el dato `mes`.

Codificación (Consultar la página web del libro)

5.11. *Escribir un programa que introduzca el número de un mes (1 a 12) y visualice el número de días de ese mes.*

Análisis del problema

Para resolver el problema, hay que tener en cuenta que el mes 2 corresponde a febrero que puede tener 29 o 28 días dependiendo de si es o no bisiesto el año correspondiente. De esta forma, además de leer el mes, se lee el año, y mediante la función `bisiesto()` (ejercicio 5.9), se decide si febrero tiene 28 o 29 días. El resto de los meses tiene 31 días excepto Abril, Junio, Septiembre y Noviembre.

Codificación

```
#include <stdio.h>
int bisiesto( int x) ;

int main()
{
    int mes, ano;
    puts(" introduzca mes entre 1 y 12 \n");
    scanf(" %d", &mes);
    puts(" introduzca año \n");
    scanf(" %d", &ano);
    if (mes == 2)
        if(bisiesto(ano))
            printf(" tiene 29 dias\n");
        else
            printf(" tiene 28 dias\n");
    else
        if((mes == 4) || (mes == 6) || (mes == 9) || (mes == 11))
            printf(" tiene 30 dias \n");
        else printf(" tiene 31 dias \n");
    return 0;
}
```

5.12. *Escribir y comprobar un programa que resuelva la ecuación cuadrática ($ax^2 + bx + c = 0$).*

Análisis del problema

Para resolver el problema se ha tenido en cuenta que:

1. Si $a \neq 0$ se presentan tres casos: el primero con dos soluciones dada por la fórmula que da la solución de la ecuación de segundo grado cuando el discriminante es positivo. El segundo con una solución dada por la fórmula cuando el discriminante es cero. El tercero con dos soluciones complejas, dada por la fórmula cuando el discriminante es negativo.
2. Si $a = 0$ se presentan a su vez otros tres casos: el primero es cuando $b \neq 0$ cuya solución es $-c/b$. El segundo es cuando $b = 0$ y $c = 0$, que es evidentemente una identidad. El tercero cuando $b = 0$ y $c \neq 0$ que no puede tener solución.

Codificación (Consultar la página web del libro)

5.13. *Escriba una sentencia que escriba menor, si el valor de la variable `dato` es menor que cero, y mayor, si el valor de `dato` es mayor que cero.*

Solución

```
#include <stdio.h>
void main()
{
    float x;
    printf(" dame dato\n");
    scanf("%f", &x);
    if( x < 0 )
        printf(" menor que cero ");
    if ( x > 0 )
        printf("mayor que cero");
}
```

5.14. Escribir un programa que lea tres enteros y emita un mensaje que indique si están o no en orden numérico.

Solución

Se leen los tres datos y se comprueba la condición con una *sentencia de selección doble*.

```
#include <stdio.h>
void main()
{
    float x, y, z;
    printf(" dame tres datos\n");
    scanf("%f %f %f", &x,&y,&z);
    if( ( x <= y ) && ( y <= z ) )
        printf(" ordenados ");
    else
        printf("no ordenados");
}
```

5.15. Codificar un programa que escriba la calificación correspondiente a una nota, de acuerdo con el siguiente criterio:

0	$a < 5.0$	<i>Suspenso</i>
5	$a < 6.5$	<i>Aprobado</i>
6.5	$a < 8.5$	<i>Notable</i>
8.5	$a < 10$	<i>Sobresaliente</i>
10		<i>Matrícula de honor</i>

Análisis del problema

El programa lee la nota en una variable real, y mediante `if` anidados escribirá el resultado pedido.

Codificación

```
#include <stdio.h>
void main()
{
    float nota;
    printf(" dame nota\n");
    scanf("%f", &nota);
    if(( nota < 0.0 ) || ( nota > 10 ))
```

```
    printf(" error en nota ");
else if ( nota < 5.0 )
    printf("Suspendido");
else if( nota < 6.5 )
    printf("Aprobado");
else if ( nota < 8.5)
    printf("Notable");
else if ( nota < 10)
    printf("Sobresaliente");
else
    printf("Matricula de Honor");
}
```

5.16. *Escriba un programa que determine el mayor de 5 números leídos del teclado.*

Análisis del problema

El programa lee un numero real *Mayor*. Posteriormente, lee en una iteración otros cuatro números quedándose en cada una de ellas en la variable *Mayor* con el número mayor leído hasta el momento.

Codificación

```
#include <stdio.h>
void main()
{
    float x, Mayor;
    int i;
    printf(" dame numero\n");
    scanf("%f" , &Mayor);
    for(i = 2; i <= 5; i++)
    {
        printf(" dame numero\n");
        scanf("%f" , &x);
        if( Mayor < x)
            Mayor = x;
    }
    printf(" el mayor es %f\n", Mayor);
}
```

5.17. *Se desea calcular el salario neto semanal de los trabajadores de una empresa de acuerdo a las siguientes normas:*

Horas semanales trabajadas ≤ 38 a una tasa dada.

Horas extras (38 o más) a una tasa 50 por 100 superior a la ordinaria.

Impuestos 0 por 100, si el salario bruto es menor o igual a 50.000 pesetas.

Impuestos 10 por 100, si el salario bruto es mayor de 50.000 pesetas.

Análisis del problema

Se escribe un programa que lee las *Horas*, la *Tasa*, calcula las horas extras, así como el *Salario Bruto* y el *Salario Neto* de acuerdo con los especificado.

Codificación

```
#include <stdio.h>
void main()
{
    float Horas, Extras, Tasa, SalarioBruto, SalarioNeto;
    printf(" dame Horas\n");
    scanf("%f", &Horas);
    if ( Horas <= 38 )
        Extras = 0;
    else
    {
        Horas = 38;
        Extras = Horas - 38;
    }
    printf("introduzca Tasa\n");
    scanf("%f",&Tasa);
    SalarioBruto = Horas * Tasa + Extras * Tasa * 1.5;
    if (SalarioBruto < 50000.0)
        SalarioNeto = SalarioBruto;
    else
        SalarioNeto = SalarioBruto * 0.9;
    printf(" Salario bruto %f \n", SalarioBruto);
    printf(" Salario neto %f \n", SalarioNeto) ;
}
```

5.18. *¿Qué salida producirá el siguiente código, cuando se empotra en un programa completo?*

```
int x = 2;
puts("Arranque");
if ( x <= 3)
if ( x != 0)
    puts("Hola desde el segundo if");
else
    puts("Hola desde el else.");
puts("Fin\nArranque de nuevo");
if ( x > 3)
if ( x != 0)
    puts("Hola desde el segundo if.");
else
    puts("Hola desde el else.");
puts("De nuevo fin");
```

Solución

Arranque
 Hola desde el segundo if
 Fin
 Arranque de nuevo
 De nuevo fin

5.19. *¿Cuál es el error de este código?*

```
printf("Introduzca n:");
scanf("%d", &n);
if (n < 0)
    puts("Este número es negativo. Pruebe de nuevo.");
    scanf("%d", &n);
else
    printf("conforme. n = %d\n", n);
```

Solución

El error está determinado porque el `else` no está bien enlazado con el `if`. O bien se elimina una de las dos sentencias o bien se pone llaves, como se indica a continuación.

```
#include <stdio.h>
void main()
{
    int n;
    printf("Introduzca n:");
    scanf("%d", &n);
    if ( n < 0 )
    {
        puts("Este número es negativo. Pruebe de nuevo .");
        scanf("%d", &n);
    }
    else
        printf("conforme. n= %d\n", n);
}
```

5.20. *¿Qué hay de incorrecto en el siguiente código?*

```
if (x = 0)
    printf("%d = 0\n", x);
else
    printf("%d != 0\n", x);
```

Solución

La sentencia anterior tiene de incorrecta que `x = 0` es una sentencia de asignación y no es una expresión lógica. Lo que el programador es probable que haya querido poner es:

```
if (x == 0)
    printf("%d = 0\n", x);
else
    printf("%d != 0\n", x);
```

5.21. *Cuatro enteros entre 0 y 100 representan las puntuaciones de un estudiante de un curso de informática. Escribir un programa para encontrar la media de estas puntuaciones y visualizar una tabla de notas de acuerdo al siguiente cuadro:*

<i>Media Puntuación</i>	
<i>[90 , 100]</i>	<i>A</i>
<i>[80, 90)</i>	<i>B</i>
<i>[70, 80)</i>	<i>C</i>
<i>[60, 70)</i>	<i>D</i>
<i>[0, 60)</i>	<i>E</i>

Análisis del problema

El programa que se escribe, lee las cuatro notas enteras, calcula la media real, y escribe la media obtenida y su puntuación de acuerdo con la tabla anterior.

Codificación

```
#include <stdio.h>
void main()
{
    int nota1,nota2,nota3,nota4;
    float media;
    printf("Dame nota 1 ");
    scanf("%d", &nota1);
    printf("Dame nota 2 ");
    scanf("%d", &nota2);
    printf("Dame nota 3 ");
    scanf("%d", &nota3);
    printf("Dame nota 4 ");
    scanf("%d", &nota4);
    media = (float)(nota1 + nota2 + nota3 + nota4) / (float)4;
    if(( media < 0) || ( media > 100 ))
        printf("fuera de rango ");
    else if( media >= 90)
        printf(" media = %f  A", media);
    else if(media >= 80)
        printf(" media = %f  B", media);
    else if(media >= 70)
        printf(" media = %f  C", media);
    else if(media >= 60)
        printf(" media = %f  D", media);
    else
        printf(" media = %f  E", media);
}
```

PROBLEMAS PROPUESTOS

- 5.1.** Explique las diferencias entre las sentencias de la columna de la izquierda y de la columna de la derecha. Para cada una de ellas deducir el valor final de `x` si el valor inicial de `x` es 0.

<code>if (x >= 0)</code>	<code>if (x >= 0)</code>
<code> x++;</code>	<code> x++;</code>
<code>else if (x >= 1);</code>	<code>if (x >= 1)</code>
<code> x += 2;</code>	<code> x += 2;</code>

- 5.2.** El domingo de Pascua es el primer domingo después de la primera luna llena posterior al equinoccio de primavera, y se determina mediante el siguiente cálculo sencillo:

```
A = año mod 19
B = año mod 4
C = año mod 7
D = (19 * A + 24) mod 30
E = (2 * B + 4 * C + 6 * D + 5) mod 7
N = (22 + D + E)
```

Donde `N` indica el número de día del mes de marzo (si `N` es igual o menor que 3) o abril (si es mayor que 31). Construir un programa que determine fechas de domingos de Pascua.

- 5.3.** Determinar el carácter asociado a un código introducido por teclado corresponde a un carácter alfabético, dígito, de puntuación, especial o no imprimible.
- 5.4.** Escribir un programa que lea la hora de un día de notación de 24 horas y la respuesta en notación de 12 horas. Por ejemplo, si la entrada es 13:45, la salida será: 1:45 PM. El programa pedirá al usuario que introduzca exactamente cinco caracteres. Por ejemplo, las nueve en punto se introduce como: 09:00.
- 5.5.** Escribir un programa que acepte fechas escritas de modo usual y las visualice como tres números. Por ejemplo, la entrada 15, Febrero 1989 producirá la salida 15 02 1989

- 5.6.** Escribir un programa que acepte un número de tres dígitos escrito en palabra y a continuación los visualice como un valor de tipo entero. La entrada se termina con un punto. por ejemplo, la entrada `doscientos veinticinco` producirá la salida `225`.

- 5.7.** Escribir un programa que acepte un año escrito en cifras arábigas y visualice el año escrito en números romanos, dentro del rango 1000 a 2000.

Nota: Recuerde que V = 5 X = 10 L = 50 C = 100
D = 500 M = 1000

IV = 4 XL = 40 CM = 900
MCM = 1900 MCML = 1950 MCMLX = 1960
MCMXL = 1940 MCMLXXXIX = 1989

- 5.8.** Se desea redondear un entero positivo `N` a la centena más próxima y visualizar la salida. Para ello la entrada de datos debe ser los cuatro dígitos `A`, `B`, `C`, `D`, del entero `N`. Por ejemplo, si `A` es 2, `B` es 3, `C` es 6 y `D` es 2, entonces `N` será 2362 y el resultado redondeado será 2400. Si `N` es 2342, el resultado será 2300, y si `N` = 2962, entonces el número será 3000. Diseñar el programa correspondiente.
- 5.9.** Se quiere calcular la edad de un individuo, para ello se va a tener como entrada dos fechas en el formato día (1 a 31), mes (1 a 12) y año (entero de cuatro dígitos), correspondientes a la fecha de nacimiento y la fecha actual, respectivamente. Escribir un programa que calcule y visualice la edad del individuo. Si es la fecha de un bebé (menos de un año de edad), la edad se debe dar en meses y días; en caso contrario, la edad se calculará en años.
- 5.10.** Se desea leer las edades de tres de los hijos de un matrimonio y escribir la edad mayor, la menor y la media de las tres edades.

Estructuras de control: bucles

Una de las características de las computadoras que aumentan considerablemente su potencia es su capacidad para ejecutar una tarea con gran velocidad, precisión y fiabilidad. Las tareas repetitivas es algo que los humanos encuentran difíciles y tediosas de realizar. En este capítulo se estudian las *estructuras de control iterativas* o *repetitivas* que realizan la repetición o iteración de acciones. C soporta tres tipos de estructuras de control: los bucles `while`, `for` y `do-while`. Estas estructuras de control o sentencias repetitivas controlan el número de veces que una sentencia o listas de sentencias se ejecutan.

6.1 La sentencia `while`

Un bucle `while` tiene una *condición* del bucle (una expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que en un bucle `while` se evalúa la condición *antes* de que se ejecute el cuerpo del bucle. La Figura 6.1 representa el diagrama del bucle `while`. El diagrama indica que la ejecución de la sentencia o sentencias se repite **mientras** la condición del bucle permanece *verdadera* (**true**) y termina cuando se hace *falsa* (**false**). En otras palabras, el cuerpo de un bucle `while` se ejecutará *cero o más veces*.

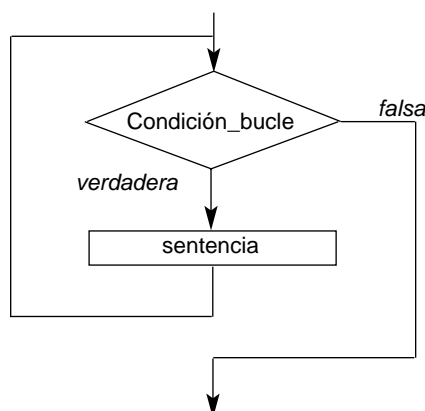


Figura 6.1 Diagrama del bucle `while`.

Sintaxis:

```

1  while (condición_bucle)
    sentencia; → cuerpo

2  while (condición_bucle)
    {
        sentencia-1;
        sentencia-2;
        .
        .
        .
        sentencia-n;
    } → cuerpo

```

<code>while</code>	→	palabra reservada C
<code>condición_bucle</code>	→	expresión lógica o booleana
<code>sentencia</code>	→	sentencia simple o compuesta

EJEMPLO 6.1 Bucle mientras para escribir de 0 a 10

```

int x = 0;
while ( x < 10)
    printf("X: %d", x++);

```

6.1.1 MISCELÁNEA DE CONTROL DE BUCLES While

Si la variable de control no se actualiza, el bucle se ejecutará "siempre". Tal bucle se denomina **bucle infinito**. En otras palabras un bucle infinito (sin terminación) se producirá cuando la condición del bucle permanece y no se hace falsa en ninguna iteración.

Bucles controlados por centinelas. Un centinela es un valor que sirve para terminar el proceso del bucle. Este valor debe ser elegido con cuidado por el programador para que no afecte al normal funcionamiento del bucle.

Bucles controlados por indicadores (banderas). En C se utiliza como bandera una variable entera que puede tomar dos valores, 1 ó 0. Un bucle controlado por bandera – *indicador*- se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador. Para que un bucle esté controlado por una bandera previamente debe ponerse la bandera a 1 (*true*, verdadero), y cuando se produzca el suceso que decide que hay que salirse del bucle se cambia el valor de la bandera a 0 (*false*, falso).

La sentencia break en los bucles. La sentencia `break` se utiliza para la salida de un bucle `while` o `do-while`, también se utiliza dentro de una sentencia `switch`, siendo éste su uso más frecuente.

```

while (condición1)
{
    if (condición2)
        break;
    /* sentencias */
}

```

El uso de `break` en un bucle no es muy recomendable ya que puede hacer difícil la comprensión del comportamiento del programa. En particular, suele hacer muy difícil verificar los invariantes de los bucles.

EJEMPLO 6.2 El siguiente código extrae y visualiza valores de entrada desde el dispositivo estándar de entrada hasta que se encuentra un valor especificado.

```
int clave=-1;
int entrada;
while (scanf("%d", &entrada))
{
    if (entrada != clave)
        printf("%d\n", entrada);
    else
        break;
}
```

6.2 Repetición: el bucle `for`

El bucle `for` es el más adecuado para implementar *bucles controlados por contador*, que son bucles en los que un conjunto de sentencias se ejecutan una vez por cada valor de un rango especificado, de acuerdo al algoritmo: por cada valor de una *variable_contador* de un rango específico ejecutar sentencias. El bucle `for` se diferencia del bucle `while` en que las operaciones de control del bucle se sitúan en un solo sitio: *la cabecera de la sentencia*.

Sintaxis:

```
for (Inicialización; CondiciónIteración; Incremento)
    Sentencias
```

El bucle `for` contiene las cuatro partes siguientes:

- *Parte de inicialización*: inicializa las variables de control del bucle. Se pueden utilizar variables de control del bucle simples o múltiples.
- *Parte de condición*: contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que la expresión sea verdadera.
- *Parte de incremento*: incrementa o decrementa la variable o variables de control del bucle.
- *Sentencias*: acciones o sentencias que se ejecutarán por cada iteración del bucle.

EJEMPLO 6.3 *Suma de los 10 primeros números múltiplos de tres.*

```
#include <stdio.h>
int main()
{
    int n, suma = 0;
    for (n = 1; n <= 10; n++)
        suma += 3*n;
    printf("La suma de los 10 primeros múltiplos de tres es:%d",suma);
    return 0;
}
```

Sentencias `break` y `continue`. La sentencia `break` termina la ejecución de un bucle, o, en general de cualquier sentencia. La sentencia `continue` hace que la ejecución de un bucle vuelva a la cabecera del bucle.

EJEMPLO 6.4 *Se escribe un bucle y para descartar un determinado valor clave se utiliza `continue`.*

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



```

int clave,i;
puts("Introduce -1 para acabar.");
for (i = 0; i < 5; i++)
{
    if (clave == -1) continue;
    scanf("%d",&clave);
}

```

Si en este bucle se introduce el valor de `-1`, entonces el bucle itera, como máximo 5 veces, pero no vuelve a leer ninguna clave ya que la orden `continue` hace que se itere, y no se pase por la orden `scanf()`.

6.3 Repetición: el bucle `do...while`

La sentencia `do-while` se utiliza para especificar un bucle condicional que se ejecuta al menos una vez.

Sintaxis:

```

do
    sentencia
while (expresión)

```

La construcción `do` comienza ejecutando *sentencia*. Se evalúa a continuación *expresión*; si *expresión* es *verdadera*, entonces se repite la ejecución de *sentencia*. Este proceso continúa hasta que *expresión* es *falsa*.

EJEMPLO 6.5 Bucle para imprimir las letras minúsculas del alfabeto.

```

char car = 'a';
do
{
    printf("%d ",car);
    car ++;
} while (car <= 'z');

```

6.4 Comparación de bucles `while`, `for` y `do-while`

Tabla 6.1 Formatos de los bucles

<code>while</code>	El uso más frecuente es cuando la repetición no está controlada por contador; el test de condición precede a cada repetición del bucle; el cuerpo del bucle puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es <i>falsa</i> .
<code>for</code>	Bucle de conteo cuando el número de repeticiones se conoce por anticipado y puede ser controlado por un contador; también es adecuado para bucles que implican control no contable del bucle con simples etapas de inicialización y de actualización; el test de la condición precede a la ejecución del cuerpo del bucle.
<code>do-while</code>	Es adecuada cuando se debe asegurar que al menos se ejecuta el bucle una vez.

PROBLEMAS RESUELTOS

6.1. *Cuál es la salida del siguiente segmento de programa?*

```
for (cuenta = 1; cuenta < 5; cuenta++)
    printf("%d ",(2 * cuenta));
```

Solución

En cada iteración se escribe el doble del valor de cuenta que ha sido previamente inicializada a uno. Por tanto se escribirán los valores 2, 4, 6, 8. Cuando cuenta toma el valor de 5, no entra en el bucle.

Puede comprobarse fácilmente este resultado ejecutando el siguiente programa:

```
#include <stdio.h>
void main()
{
    int cuenta;
    for (cuenta = 1; cuenta < 5; cuenta++)
        printf("%d ",(2 * cuenta));
}
```

6.2. *¿Cuál es la salida de los siguientes bucles?*

1. **for** ($n = 10$; $n > 0$; $n = n - 2$)
 {
 printf("Hola");
 printf(" %d \n",n);
 }

2. **double** $n = 2$;
 for (; $n > 0$; $n = n - 0.5$)
 printf("%g ",n);

Solución

1. En este primer caso la variable n se inicializa al valor 10 y se decrementa en cada iteración dos unidades saliendo del bucle cuando es negativa o nula. Por tanto la salida será:

```
Hola 10
Hola 8
Hola 6
Hola 4
Hola 2
```

Si se ejecuta el siguiente programa puede comprobarse los resultados:

```
#include <stdio.h>
void main()
{
    int n;
    for (n = 10; n > 0; n = n-2)
    {
```

```

    printf("Hola");
    printf(" %d \n",n);
}
}

```

2. En este segundo caso la variable *n* se inicializa al valor 2 y se decrementa en cada iteración 0.5 unidades saliendo del bucle cuando es negativa o nula. Por tanto la salida será: 2 1.5 1 0.5. Si se ejecuta el siguiente programa puede comprobarse los resultados.

```

#include <stdio.h>
void main()
{
    double n = 2;
    for (; n > 0; n = n - 0.5)
        printf("%g ", n);
}

```

6.3 ¿Cuál es la salida de los siguientes bucles?.

```

int n, m;
for ( n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m - -)
        printf("%d veces %d = %d \n", n , m, n * m );

```

Solución

La variable *n* toma los valores 1, 2, 3,..., 10. Para cada uno de estos valores la variable *m* toma los valores 10,9, 8, 7,..., 1. Por lo tanto la salida son las tablas de multiplicar de los números 1, 2, ... 10, pero en el orden inverso.

- 6.4. Escriba un algoritmo que usando un bucle *for* infinito, y una sentencia *break* calcule la suma de los *n* > 0 primeros números que se lean del teclado. El número *n* es un dato y es el primero de la secuencia.

Análisis del problema

En primer lugar se lee el valor de *n* que cumpla la condición pedida, para posteriormente mediante un bucle *for* infinito leer los números del teclado hasta que se cumpla que se haya leído la *n* indicada.

Codificación

```

#include <stdio.h>
int main()
{
    int n, c = 0, x, suma = 0;
    do
    {
        printf("Cuantos números? ");
        scanf("%d", &n);}
    while (n < 0);
    for (;;)
    {
        if(c < n)
            suma = suma + c;
        c++;
    }
}

```

/* inicialización */

/* bucle for que no termina nunca */

/* test */

```

        scanf( "%d", &x) ;
        suma += x;
        c++;
    }
    else
        break;
}
printf("suma =% d", suma);
return 0;
}

```

- 6.5.** *Diseñar un programa que lea un límite máximo entero positivo, una base entera positiva, y visualice todas las potencias de la base, menores que el valor especificado en límite máximo.*

Análisis del problema

Se implementan tres bucles. Un primer bucle `do-while`, valida la entrada del límite entero positivo. Un segundo bucle `do-while`, valida la entrada de la base entera positiva. Un tercer bucle controlado por un `for` escribe las distintas potencias.

Codificación

```

#include <stdio.h>
void main()
{
    int max_limit, base, pot;
    do
    {
        printf(" introduzca numero positivo ");
        scanf("%d", &max_limit);
    }
    while (max_limit < 0);
    do
    {
        printf(" introduzca base positiva ");
        scanf("%d", &base);
    }
    while (base < 0);
    printf( "sucesivas potencias de %d \n", base);
    for ( pot = 1; pot <= max_limit; pot *= base)
        printf("%d \n", pot );
}

```

- 6.6.** *¿Qué hace el siguiente bucle `while`? Reescribirlo con sentencias `for` y `do-while`.*

```

num = 10;
while (num <= 100)
{
    printf("%d \n", num);
    num += 10;
}

```

Análisis del problema

El programa anterior escribe en pantalla los siguientes números 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. En el siguiente programa se han escrito las sentencias equivalentes al `while` anterior con un bucle `for`, y con un bucle `do while` en un mismo programa.

Codificación

```
#include <stdio.h>
int main()
{
    int num = 10;
    while (num <= 100)
    {
        printf("%d \n",num);
        num += 10;
    }

    // con bucle for
    for (num = 10; num <= 100; num += 10)
        printf("%d \n", num);

    // con bucle do while
    num = 10;
    do
    {
        printf("%d \n", num);
        num += 10;
    }
    while (num <= 100);
    return 0;
}
```

6.7. ¿Cuál es la salida del siguiente fragmento de programa?.

```
#include <stdio.h>
int main()
{
    int i,j;
    i = 1;
    while (i * i < 10)
    {
        j = i ;
        while ( j * j < 100)
        {
            printf("%d ", i + j);
            j *= 2;
        }
        printf("\n");
        i++;
    }
    return 0;
}
```

Solución

En el bucle controlado por la variable i , los valores que puede tomar ésta son $i = 1, 2, 3$. En el momento que i toma el valor de 4, se sale del bucle ya que $4*4$ no es menor que 10. En el bucle controlado por la variable j , se observa que j se inicializa en los valores 1, 2, 3, respectivamente, y en cada iteración se va multiplicando por dos. Así cuando i vale 1 los valores que toma la variable j son 1, 2, 4, 8, y cuando toma el valor de 16 se sale del bucle ya que $16 * 16$ es mayor que 100. Cuando i vale 2, los valores que toma la variable j son 2, 4, 8, y cuando toma el valor de 16 se sale del bucle al igual que antes. Cuando i vale 3, la variable j toma los valores 3, 6, y cuando toma el valor de 12 se sale del bucle. Teniendo en cuenta lo anteriormente dicho, se tiene que la salida producida será:

2, 3, 5, 9
4, 6, 10
6, 9
6.8

- 6.8.** Diseñe un algoritmo que sume los 20 primeros números impares.

Análisis del problema

Se necesita un acumulador `suma` que será donde se sumen los respectivos números impares. Para calcular los 20 primeros números impares basta con recorrer mediante un bucle `for` los números del 0 al 19 y si i es la variable que lo controla, el correspondiente número impar es $2 * i + 1$.

Codificación

```
#include <stdio.h>
int main()
{
    int i, suma = 0;
    for (i = 0; i <= 19; i++)
        suma += 2 * i + 1;
    printf("La suma de los 20 primeros números impares: %d", suma);
    return 0;
}
```

- 6.9.** Escriba un programa que lea un número $n1$, y escriba la tabla de multiplicar del número.

Análisis del problema

Se lee el número y mediante un bucle `for` y se itera 10 veces escribiendo los resultados.

Codificación

```
#include <stdio.h>
int main()
{
    int n1, n2;
    printf("introduzca numero \n");
    scanf("%d", &n1);
    printf("tabla de multiplicar del %d \n", n1);
    for (n2 = 1; n2 <= 10; n2++)
        printf("%d X %d = %d\n", n1, n2, n1 * n2);
    return 0;
}
```

6.10. *Escriba un programa que escriba la tabla de multiplicar del 1, 2,...,9.*

Análisis del problema

Se hace de una manera análoga al ejercicio anterior, pero ahora anidando dos bucles for, y sin leer ningún dato.

Codificación

```
#include <stdio.h>
int main()
{
    int n1,n2;
    char ch;
    for (n1 = 1;n1 <= 9; n1++)
    {
        printf(" tabla de multiplicar del %d \n", n1);
        for (n2 = 1; n2 <= 10; n2++)
            printf(" %d X %d = %d\n", n1, n2, n1 * n2);
        scanf("%c", &ch);
    }
    return 0;
}
```

6.11. *Diseñar e implementar un programa que solicite a su usuario un valor no negativo n y visualice la siguiente salida:*

```
1      2      3 ..... n-1      n
1      2      3 ..... n-1
...    ...    ...    ...
1      2      3
1      2
1
```

Análisis del problema

Un primer bucle debe validar la entrada del dato n. Para escribir la tabla anterior se implementan dos bucles anidados.

Codificación

```
#include <stdio.h>
int main()
{
    int i, j, n;
    do
    {
        printf("valor de n >0\n");
        scanf("%d", &n);
    }
    while (n <= 0);

    for (i = n; i >= 1; i--) // termina la lectura de n
    {                       // para cada una de las filas descendentemente
        for (j = 1;j <= i; j++)
        {
            // para cada una de las columnas
```

```

        printf("%2d", j);
        printf("\n");
    }
    return 0;
}

```

// salta de línea

6.12. Implementar y ejecutar un programa que invierta los dígitos de un entero positivo dado.

Análisis del problema

Para resolver el problema se inicializa una variable $n1$ a cero. Un bucle controlado por una variable n (la leída) termina cuando su valor es cero. En cada iteración del bucle se calcula en la propia variable n el valor del cociente entero de n entre 10. Así si la variable n toma el valor de 234, en las sucesivas iteraciones irá tomando los valores 234, 23, 2 y cero. En cada iteración del bucle, se va calculando el resto del cociente entero de n entre 10. Es decir se van calculando los valores, 4, 3, 2. Para conseguir obtener el número invertido, basta con observar que $432 = 4 \cdot 10 \cdot 10 + 3 \cdot 10 + 2 = (((0 \cdot 10 + 4) \cdot 10 + 3) \cdot 10 + 2)$. (Método de *Horner* de evaluación de polinomios). Es decir, basta con acumular en $n1$ el valor de $n1$ multiplicado por 10 y sumarle el resto de la división entera. De todo lo dicho anteriormente, se deduce obviamente el siguiente programa.

Codificación

```

#include <stdio.h>
int main()
{
    int i, n, n1;
    do
    {
        printf("valor de n >0\n");
        scanf("%d", &n);
    }
    while (n <= 0);
    n1 = 0;
    while (n != 0)
    {
        i = n % 10;
        n = n / 10;
        n1 = n1 * 10 + i;
    }
    printf(" número invertido %d", n1);
    return 0;
}

```

6.13. Implementar el algoritmo de Euclides que encuentra el máximo común divisor de dos números enteros y positivos.

Análisis del problema

El algoritmo transforma un par de enteros positivos (n, m) en una par $(n1, m1)$, dividiendo repetidamente el entero mayor por el menor y reemplazando el mayor por el menor y el menor por el resto. Cuando el resto es 0, el número más pequeño distinto de cero de la pareja será el máximo común divisor de la pareja original.

La codificación que se realiza, lee primeramente los números enteros n y m , validando la entrada. Posteriormente mediante otro bucle se efectúan las correspondientes transformaciones para obtener el máximo común divisor.

Codificación

```

#include <stdio.h>
int main()
{
    int r, n, m;
    do
    {
        printf("valor de n >0\n");
        scanf("%d", &n);
    } while (n <= 0);
    do
    {
        printf("valor de m >0\n");
        scanf("%d", &m);
    } while (m <= 0);

    /* no es necesario comenzar en n con el mayor, ya que el
       algoritmo de Euclides lo primero que hace es intercambiar
       valores */
    printf(" el maximo comun divisor entre %3d y %3d\n", n, m);
    r= n % m;
    while ( r != 0 )
    {
        n = m;
        m = r;
        r = n % m;
    }
    printf(" es %3d\n", m);
    return 0;
}

```

6.14. *Escriba un algoritmo que lea dos números enteros positivos y calcule el mínimo común múltiplo de los dos números.*

Análisis del problema

Una forma sencilla de resolver el problema es tener en cuenta que siempre el producto de dos números positivos cualesquiera coincide con el producto del máximo común divisor por el mínimo común múltiplo. Entonces, modificando el problema 6.13, se puede obtener el mínimo común múltiplo. Otra manera también sencilla de resolverlo, es tomar el mínimo común múltiplo como el mayor de los dos números, y mediante un bucle que itere mientras que los números dados no dividan al mínimo común múltiplo hacer incrementar en una unidad el mínimo común múltiplo.

Codificación

```

#include <stdio.h>
int main()
{
    int n, m, mcm;
    do
    {
        printf("valor de n >0\n");
        scanf("%d", &n);
    } while ( n <= 0 );
    do

```

```

{
    printf("valor de m > 0\n");
    scanf("%d", &m);
} while (m <= 0);
if( n < m)
    mcm = m ;
else
    mcm = n ;
while ((mcm % m != 0) || (mcm % n != 0))
    mcm++;
printf(" el minimo comun multiplo es %3d\n", mcm);
return 0;
}

```

6.15. Escribir un programa que lea el radio de una esfera y visualice su área y su volumen.

Análisis del problema

Teniendo en cuenta que las fórmulas que dan el área y volumen de una esfera son: $a = 4\pi r^2$, $v = 4/3\pi r^3$ para resolver el problema sólo se tiene que leer el radio (positivo), validarlo en un bucle y aplicar las fórmulas anteriores para obtener el área y el volumen.

Codificación

```

#include <stdio.h>
int main()
{
    float r, a, v, pi = 3.141592;
    do
    {
        printf("valor del radio > 0\n");
        scanf("%f", &r);
    } while (r <= 0);

    a = 4 * pi * r * r;
    v = 4.0 / 3 * pi * r * r * r;

    printf("el area y volumen de la esfera de radio r=%f es:\n", r);
    printf("area = %f \n volumen = %f \n", a, v);
    return 0;
}

```

6.16. Escriba un programa que escriba los valores de la función seno(2x)-x para los valores de x igual a 0, 0.5, 1.0,.....9.5, 10.

Análisis del problema

Se define la constante simbólica `m` como 10 y una "función en línea" `f(x)` (también llamada una macro con argumentos). El bucle se realiza 21 veces; en cada iteración el valor de `x` se incrementa en 0.5, se calcula el valor de la función y se escriben los resultados.

Codificación

```

#include <math.h>

```

```
#include <stdio.h>
#define M 10
#define f(x) sin( 2 * x) - x
int main()
{
    double x;
    for (x = 0.0; x <= M; i += 0.5)
        printf("%f = f\n", x, f(x));
    return 0;
}
```

- 6.17.** *Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de n números ($n > 0$). El valor de n se solicitará al principio del programa y los números serán introducidos por el usuario.*

Análisis del problema

Primeramente se lee el número n en bucle `do-while` que valide que es positivo. Posteriormente se lee el primer número de la serie, y se inicializa el mayor, el menor y la media a ese número. Un bucle `for` va leyendo el resto de los números, y mediante la técnica *voraz* (el mejor de todos es o el mejor de todos los anteriores o es el que se acaba de leer) se recalculan los nuevos mínimo y máximo, y a la vez se acumula en media el último valor leído. Al final se escriben los resultados y la media que es la suma obtenida en la variable `media` dividido por n .

Codificación

```
#include <stdio.h>
int main()
{
    int i, n;
    float M, m, media, num;
    do
    {
        printf("valor de  n >0\n");
        scanf("%d", &n);
    } while ( n <= 0 );

                                                                    // fin de entrada de datos

    printf ( "introduzca %d numeros \n", n);
    scanf("%f", &num);
    M = num;
    m = num;
    media = num;

                                                                    // bucle voraz

    for (i = 2; i <= n; i++)
    {
        scanf("%f",&num);
        if (M < num)
            M = num;
        if (m > num)
            m = num;
        media = media + num;

                                                                    // se recalcularon los nuevos máximos, mínimos y suma= media
    }
    media = media / n;
    printf(" media = %f \n", media);
}
```

```
printf(" menor = %f \n", m);
printf(" mayor = %f \n", M);
return 0;
}
```

- 6.18.** *Un número perfecto es un entero positivo, que es igual a la suma de todos los enteros positivos (excluido el mismo) que son divisores del número. El primer número perfecto es 6, ya que los divisores de 6 son 1, 2, 3 y $1 + 2 + 3 = 6$. Escribir un programa que lea un número entero positivo n y decida si es perfecto.*

Análisis del problema

Se lee el número n en un bucle validando la entrada. Posteriormente en un bucle `for`, prueba todos los posibles candidatos a divisores menores que n (basta con empezar en 1 y avanzar de uno en uno hasta llegar a $n-1$. Podría mejorarse el bucle llegando sólo a la raíz cuadrada de n). Estos divisores se van acumulando en un acumulador, para al final del bucle comprobar la condición de *perfecto* y dar el mensaje correspondiente.

Codificación

```
#include <stdio.h>
int main()
{
    int i, n, resto, acu;
    do
    {
        printf("valor de n > 0\n");
        scanf("%d", &n);
    }
    while ( n <= 0);
    acu = 0;

    // acu contendrá en todo momento la suma de todos lo divisores
    // conocidos de n menores que i

    for (i = 1; i < n; i++)
    {
        resto = n % i;
        if (resto == 0) acu += i;           /* nuevo divisor*/
    }
    if (n == acu)
        printf(" el numero %d es perfecto\n", n);
    else
        printf(" el numero %d no es perfecto \n", n);
    return 0;
}
```

- 6.19.** *El valor de e^x se puede aproximar por la suma*

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Escribir un programa que tome un valor de x como entrada y visualice la suma para cada uno de los valores de 1 a 100.

Análisis del problema

El problema se resuelve teniendo en cuenta que para calcular el valor de la serie, basta con ir acumulando los sucesivos valores del término, y que cada término de la serie se obtiene del inmediatamente anterior, multiplicando por x y dividiendo por

i , siendo i un contador que indica el número de término que se está sumando. Por ejemplo $x^3/3! = x^2/2! * (x/3)$. El término cero es 1, el término 1 es x , y así sucesivamente.

Codificación

```
#include <stdio.h>
int main()
{
    int i;
    float x, t, e;
    printf("valor de x \n");
    scanf("%f", &x);
    e = 1;
    t = 1;

    // se han inicializado el valor de e y del término t
    printf(" distintos valores \n");
    for (i = 1; i <= 100; i++)
    {
        t *= x / i;

        // se recalculó el nuevo término

        e += t;

        // se recalculó la nueva suma

        printf(" i= %d , e= %f \n", i, e);
    }
    return 0;
}
```

- 6.20.** El matemático italiano Leonardo Fibonacci propuso el siguiente problema. Suponiendo que un par de conejos tiene un par de crías cada mes y cada nueva pareja se hace fértil a la edad de un mes. Si se dispone de una pareja fértil y ninguno de los conejos muere, ¿cuántas parejas habrá después de n años? Mejorar el problema calculando el número de meses necesarios para producir un número dado de parejas de conejos.

Análisis del problema

Para resolver el problema, basta con observar que en cada mes el número de parejas fértiles, coincide con la suma de las parejas fértiles que había en los dos meses inmediatamente anteriores, con lo que para obtener el resultado, basta con usar una variable auxiliar `aux` en la cual se suman los dos valores que se tiene en los dos meses anteriores, para actualizar de nuevo los valores de los nuevos meses de acuerdo con lo indicado. El programa que se codifica posteriormente, lee primeramente un número n positivo. Inicializa `f1` y `f2` con los valores 1, para posteriormente en un bucle `for` de uno en uno y comenzando por el valor 2, hacer `aux=f1+f2`, `f1=f2` y `f2=aux` (`f1` es el mes anterior y `f2` es el actual). Para mejorar la solución solicitada, basta con leer el número de parejas, y entrar en un bucle `while` controlado, en este caso, por la condición ser menor que n_p , y realizar la misma operación que se hizo en el bucle `for`. Al final se escribe el valor de la variable i que va contando el número de iteraciones.

Codificación

```
#include <stdio.h>
int main()
{
    int n, i, f1, f2, aux, np;
    do
    {
        printf("valor de n \n");
```

```

    scanf("%d", &n);
} while (n < 0);
f1 = 1;
f2 = 1;

// Se calculan los sucesivos términos de la sucesión de fibonacci
for (i = 2; i <= n; i++)
{
    aux = f1 + f2;
    f1 = f2;
    f2 = aux;
}
printf(" valor de fibonacci para n= %d, es %d \n", n, f2);

// comienza la mejora

printf ( "numero de parejas necesarias \n");
scanf("%d", &np);

// se supone que np es positivo.

f1 = 1;
f2 = 1;
i = 1;
while (f2 < np)
{
    aux = f1 + f2;
    f1 = f2;
    f2 = aux;
    i++;
}
printf(" numero de meses %d \n", i);
return 0;
}

```

6.21. Determinar si un número dado leído del teclado es primo o no.

Análisis del problema

Un número positivo es primo, si sólo tiene por divisores el uno y él mismo. Teniendo en cuenta que si hay un número i que divide a otro n menor que la raíz cuadrada de n , entonces hay otro que también lo divide que es mayor que la raíz cuadrada de n , se tiene que basta con comprobar los posibles divisores menores o iguales que la raíz cuadrada del número dado. El programa codificado, se ha realizado con un solo bucle, en el cual se van comprobando los posibles divisores, siempre y cuando, no se haya encontrado ya algún divisor anterior, o no se tenga que controlar ningún otro divisor. La codificación que se realiza, lee primeramente el valor de n , validando que sea positivo, y posteriormente se realiza con otro bucle lo expuesto anteriormente.

Codificación

```

#include <stdio.h>
int main()
{
    int primo, i, n;
    do
    {
        printf("valor de n >0\n");
        scanf("%d", &n);
    } while (n <= 0);
}

```

```

primo = 1;
//inicialmente el número es primo
for (i = 2; (i * i < n) & primo; i++)
    /* mientras sea primo y queden posibles divisores menores
       o iguales que la raíz de n hacer */
{
    primo = (n % i) != 0;
}
if (primo)
    printf(" el numero %d es primo\n", n);
else
    printf(" el numero %d no es primo \n", n);
return 0;
}

```

6.22. Calcular la suma de la serie $1/1 + 1/2 + \dots + 1/n$ donde n es un número que se introduce por teclado.

Análisis del problema

Para realizar la suma de la serie, basta con acumular en una variable s los distintos valores de los términos $t = 1/i$. Previamente se lee del el valor del número de términos n validando la entrada y posteriormente con un bucle `for` controlado por la variable i se va realizando la correspondiente acumulación.

Codificación

```

#include <stdio.h>
int main()
{
    int i,n;
    float t, s;
    do
    {
        printf("valor de n \n");
        scanf("%d", &n);
    } while (n <= 0);
    s = 1;
    for (i = 1; i <= n; i++)
    {
        t = 1.0 / i;
        // para obligar a que la división sea real se pone 1.0
        s += t;
    }
    printf(" valor de la suma = %f\n", s);
    return 0;
}

```

6.23. Calcular la suma de los términos de la serie: $1/2 + 2/2^2 + 3/2^3 + \dots + n/2^n$.

Análisis del problema

Para realizar la suma de la serie, basta con acumular en una variable s los distintos valores de los términos $t = i/2^i$. Previamente se lee del el valor del número de términos n validando la entrada, y posteriormente se realiza la acumulación mediante un bucle `for`.

Codificación

```
#include <stdio.h>
#include <math.h>
int main()
{
    int i, n;
    float t, s;
    do
    {
        printf("valor de n \n");
        scanf("%d", &n);
    } while ( n <= 0 );
    s = 1;
    for (i = 1; i <= n; i++)
    {
        t = float( i ) / pow(2, i);
        // Se obliga a que el cociente sea real.

        s += t;
    }
    printf(" valor de la suma = %f\n", s);
    return 0;
}
```

- 6.24.** Encontrar un número natural n más pequeño tal que la suma de los n primeros números naturales exceda el valor de una cantidad introducida por el teclado máximo.

Análisis del problema

En primer lugar se lee el valor de la cantidad introducida por teclado máximo validando la entrada; a continuación, se acumula la serie dada por los distintos números naturales, hasta que se exceda el valor introducido. Esto se realiza mediante un bucle `for` cuya salida viene dada precisamente por el valor `s >= máximo`.

Codificación

```
#include <stdio.h>
int main()
{
    int n;
    float s, maximo;
    do
    {
        printf("valor maximo n \n");
        scanf("%f", &maximo);
    } while (maximo <= 0);
    s = 0;
    for (n = 0; s < maximo;)
    {
        n++;
        s += n;
    }
    printf(" valor de la suma = %.2f\n", s);
    printf(" valor del numero de terminos = %d\n", n);
    return 0;
}
```


- 6.25.** *Escriba un programa que lea un número entero positivo y calcule su factorial, mediante un for, un while y mediante un do-while.*

Análisis del problema

Primeramente se lee el valor del número $n1$, mediante una sentencia `do-while` validando el dato para posteriormente escribir los tres bucles, con sus correspondientes inicializaciones.

Codificación (Se encuentra en la página web del libro)

- 6.26.** *Encontrar el número mayor de una serie de números introducidos por teclado.*

Análisis del problema

La codificación realizada, comienza pidiendo el primer número que será distinto de -1 . Posteriormente, se lee la serie de números. El fin de la entrada de datos viene dado por el valor de -1 . El cálculo del máximo se realiza en el cuerpo del segundo bucle controlado por el valor -1 mediante la técnica, "el mayor de todos hasta el último leído coincide con el mayor del último número que se ha leído o bien coincide con el mayor de todos los que se leyeron anteriormente".

Codificación

```
#include <stdio.h>
int main()
{
    int s, maximo;
    do
    {
        printf(" introduzca primer valor <> -1 \n");
        scanf("%d", &maximo);
    } while (maximo == -1);
    s = maximo;
    while ( s != -1)
    {
        printf(" introduzca valor -1= fin \n");
        scanf("%d", &s);
        if( s != -1)
            if (maximo < s)
                máximo = s;
    }
    printf(" valor del  maximo= %d\n", maximo);
    return 0;
}
```

- 6.27.** *Calcular todos los números de tres cifras tales que la suma de los cuadrados de las cifras es igual al valor del número.*

Análisis del problema

La solución se plantea mediante un bucle que recorre todos los números de tres cifras. En cada iteración del bucle se calcula cada una de las cifras del número y se comprueba la condición en cuyo caso se escribe. Si el número $i = c3c2c1$ entonces la condición indicada es $i = c1*c1 + c2*c2 + c3*c3$. Para calcular las cifras basta con usar el cociente y la división entera.

Codificación

```
#include <stdio.h>
int main()
{
    int c1, c2, c3, i, x;
    printf(" lista de numeros que cumplen la condicion\n");
    for(i = 100; i <= 999; i++)
    {
        x = i ;
        c1 = x % 10;
        x= x / 10;
        c2 = x % 10;
        x = x / 10;
        c3 = x;

        if( c1 * c1 + c2 * c2 + c3 * c3 == i)           // ya se han calculado las tres cifras
            printf(" numero %d\n", i);
    }
    return 0;
}
```

6.28. *Escriba un programa que sume los números pares comprendidos entre 2 y 100.*

Análisis del problema

Se inicializa una variable `suma` a 2 y otra variable `numero` a 4. Posteriormente mediante un bucle `while` se suman los números pares que va tomando `numero` a la variable `suma`, hasta que `numero` sobrepase el valor de 100.

Codificación

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

int main()
{
    int numero, suma;
    suma = 2;
    numero = 4;
    while (numero <= 100)
    {
        suma = suma + numero;
        numero = numero + 2;
    }
    printf("\nSuma pares entre 2 y 100 = %d", suma);
    return 0;
}
```

6.29. *Escriba un programa que lea números enteros del teclado y cuente los ceros que se introducen. El final de datos viene dado cuando se pulse por teclado una tecla distinta de s o S.*

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

- 6.1.** Seleccione y escriba el bucle adecuado que mejor resuelva las siguientes tareas:
- Sumar de serie $1/2+1/3+1/4+1/5+\dots+1/50$.
 - Lectura de la lista de calificaciones de un examen de Historia.
 - Visualizar la suma de enteros en el intervalo $11\dots50$.

- 6.2.** ¿Cuál es la salida del siguiente bucle?:

```
suma = 0;
while (suma < 100)
    suma += 5;
printf(" %d \n", suma);
```

- 6.3.** ¿Cuál es la salida de los siguientes bucles?:

A

```
for (i = 0; i < 10; i++)
    printf(" 2* %d = %d \n", i, 2 * i);
```

B

```
for (i = 0; i <= 5; i++)
    printf(" %d ", 2 * i + 1);
```

C

```
for (i = 1; i < 4; i++)
{
    printf(" %d ", i);
    for (j = i; j >= 1; j--)
        printf(" %d \n", j);
}
```

- 6.4.** Describir la salida de los siguientes bucles:

A

```
for (i = 1; i <= 5; i++)
{
    printf(" %d \n", i);
    for (j = i; j >= 1; j-=2)
        printf(" %d \n", j);
}
```

B

```
for (i = 3; i > 0; i--)
    for (j = 1; j <= i; j++)
        for (k = i; k >= j; k--)
            printf("%d %d %d \n", i, j, k);
```

C

```
for (i = 1; i <= 3; i++)
    for (j = 1; j <= 3; j++)
    {
        for (int k = i; k <= j; k++)
            printf("%d %d %d \n", i, j, k);
        putchar('\n');
    }
```

- 6.5.** Diseñar e implementar un programa que cuente el número de sus entradas que son positivos, negativos y cero.

- 6.6.** Diseñar e implementar un programa que extraiga valores del flujo de entrada estándar y a continuación visualice el mayor y el menor de esos valores en el flujo de salida estándar. El programa debe visualizar mensajes de advertencias cuando no haya entradas.

- 6.7.** Diseñar e implementar un programa que solicite al usuario una entrada como un dato tipo fecha y a continuación visualice el número del día correspondiente del año. Ejemplo, si la fecha es 30 12 1999, el número visualizado es 364.

- 6.8.** Un carácter es un espacio en blanco si es un blanco (' '), una tabulación ('\t'), un carácter de nueva línea ('\n') o un avance de página ('\f'). Diseñar y construir un programa que cuente el número de espacios en blanco de la entrada de datos.

- 6.9.** Escribir un programa que lea las temperaturas en grados Celsius e imprima las equivalente en grados Fahrenheit.

- 6.10.** Escribir un programa que convierta: (a) centímetros a pulgadas; (b) libras a kilogramos. (Ver problema propuesto 4.1)

- 6.11.** Escribir un programa que lea 3 enteros positivos día, mes y año y a continuación visualice la fecha que represente, el número de días, del mes y una frase que diga si el año es o no bisiesto. Ejemplo, 4/11/1999 debe visualizar 4 de Noviembre de 1999. Ampliar el programa de modo que calcule la fecha correspondiente a 100 días más tarde.

- 6.12.** En una empresa de computadoras, los salarios de los empleados se van a aumentar según su contrato actual:

Contrato	Aumento %
0 a 9.000 dólares	20
9.001 a 15.000 dólares	10
15.001 a 20.000 dólares	5
más de 20.000 dólares	0

Escribir un programa que solicite el salario actual del empleado y calcule y visualice el nuevo salario.

- 6.13.** La constante π (3.1441592...) es muy utilizada en matemáticas. Un método sencillo de calcular su valor es:

$$Pi = 4 * \left(\frac{2}{3}\right) * \left(\frac{4}{5}\right) * \left(\frac{6}{5}\right) * \left(\frac{6}{7}\right) \dots$$

Escribir un programa que efectúe este cálculo con un número de términos especificados por el usuario.

- 6.14.** Escribir un programa que visualice un cuadrado mágico de orden impar n , comprendido entre 3 y 11; el usuario elige el valor de n . Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n^2 . La suma de los números que figuran en cada línea, cada columna y cada diagonal son idénticas. Un ejemplo es:

8	1	6
3	5	7
4	9	2

Un método de construcción del cuadrado consiste en situar el número 1 en el centro de la primera línea, el número siguiente en la casilla situada encima y a la derecha, y así sucesivamente. Es preciso considerar que el cuadrado se cierra sobre sí mismo: la línea encima de la primera es de hecho la última y la columna a la derecha

de la última es la primera. Sin embargo, cuando la posición del número caiga en una casilla ocupada, se elige la casilla situada debajo del número que acaba de ser situado. Puede verse una solución con arrays en el ejercicio resuelto 9.13.

- 6.15.** Calcular la media de las notas introducidas por teclado con un diálogo interactivo semejante al siguiente:

```
¿Cuántas notas? 20
Nota 1 : 7.50
Nota 2: 6.40
Nota 3: 4.20
Nota 4: 8.50
...
Nota 20: 9.50
Media de estas 20: 7.475
```

- 6.16.** Contar el número de enteros negativos introducidos en una línea.

Funciones

Una función es un miniprograma dentro un programa. Las funciones contienen varias sentencias bajo un sólo nombre, que un programa puede utilizar una o más veces para ejecutar dichas sentencias. Las funciones ahorran espacio, reduciendo repeticiones y haciendo más fácil la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables. En otros lenguajes como BASIC o ensamblador se denominan *subrutinas*; en Pascal, las funciones son equivalentes a *funciones* y *procedimientos*. C, C++, JAVA, C# utilizan funciones.

Este capítulo examina el papel (rol) de las funciones en un programa C. Las funciones pueden existir de modo autónomo o bien como miembros de una clase. Como ya conoce, cada programa C tiene al menos una función **main()**; sin embargo, cada programa C consta de muchas funciones en lugar de una función **main()** grande. La división del código en funciones hace que las mismas se puedan reutilizar en su programa y en otros programas. Después de que escriba, pruebe y depure su función, se puede utilizar nuevamente una y otra vez. Para reutilizar una función dentro de su programa, sólo se necesita llamar a la función.

Si se agrupan funciones en bibliotecas otros programas pueden reutilizar las funciones, por esa razón se puede ahorrar tiempo de desarrollo. Y dado que las bibliotecas contienen rutinas presumiblemente comprobadas, se incrementa la fiabilidad del programa completo.

Las funciones son una de las piedras angulares de la programación en C y un buen uso de todas las propiedades básicas ya expuestas, así como de las propiedades avanzadas de las funciones, le proporcionarán una potencia, a veces impensable, a sus programaciones. La compilación separada y la recursividad son propiedades cuyo conocimiento es esencial para un diseño eficiente de programas en numerosas aplicaciones.

7.1 Concepto de función

Un programa C se compone de varias funciones, cada una de las cuales realiza una tarea principal. El mejor medio para escribir un programa es escribir funciones independientes para cada tarea que haga el programa. Cada función realiza una determinada tarea y cuando se ejecuta `return` o termina el código de la función, se retorna al punto en que fue llamada por el programa o función principal.

7.2 Estructura de una función

Una **función** es, sencillamente, un conjunto de sentencias que se pueden llamar desde cualquier parte de un programa. Las funciones en C no se pueden anidar. En C todas las funciones son externas o globales, es decir, pueden ser llamadas desde cualquier parte del programa. La estructura de una función en C es:

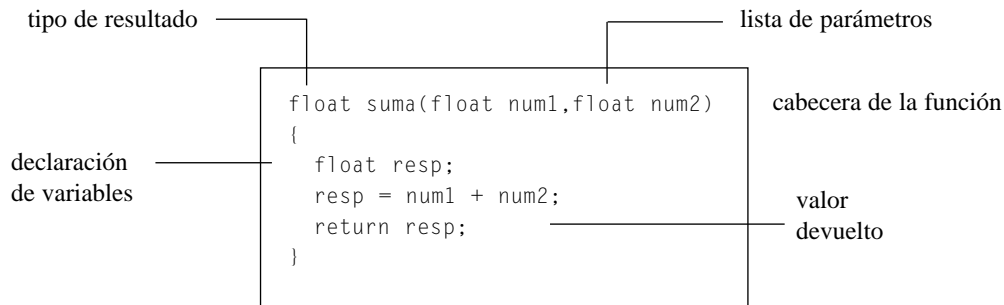
```
tipo_de_retorno nombreFunción (lista De Parámetros)
{
    cuerpo de la función
    return expresión
}
```

<i>tipo_de_retorno</i>	tipo de valor devuelto por la función o la palabra reservada <code>void</code> si la función no devuelve ningún valor.
<i>nombreFunción</i>	identificador o nombre de la función.
<i>ListaDeParámetros</i>	lista de declaraciones de los parámetros de la función separados por comas.
<i>expresión</i>	valor que devuelve la función.

Los aspectos más sobresalientes en el diseño de una función son:

- *Tipo de resultado*. Es el tipo de dato que devuelve la función C .
- *Lista de parámetros*. Es una lista de parámetros con tipos que utilizan el formato siguiente: *tipo1 parámetro1, tipo2 parámetro2, ...*
- *Cuerpo de la función*. Se encierra entre llaves de apertura ({) y cierre (}).
- *Paso de parámetros*. El paso de parámetros en C se hace siempre por valor.
- *Declaración local*. Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- *Valor devuelto por la función*. Mediante la palabra reservada `return` se puede devolver el valor de la función.
- *La llamada a una función*. Debe ser una sentencia de otra función. Esta sentencia debe ser tal que debe haber coincidencia en número orden y tipo entre la lista de parámetros formales y actuales de la función.

EJEMPLO 7.1: Codifica la función `suma()` y se muestra su estructura.



7.3 Prototipos de las funciones

La *declaración* de una función se denomina *prototipo*. Específicamente un prototipo consta de los siguientes elementos: nombre de la función, lista de argumentos encerrados entre paréntesis y un punto y coma. En C no es necesario incluir el prototipo aunque si es recomendable para que el compilador pueda hacer chequeos en las llamadas a las funciones. Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la función `main()`. El compilador utiliza los prototipos para validar que el número y los tipos de datos de los argumentos reales de la llamada a la función son los mismos que el número y tipo de argumentos formales en la función llamada. Si una función no tiene argumentos, se ha de utilizar la palabra reservada `void` como lista de argumentos del prototipo (también se puede escribir paréntesis vacíos). Un formato especial de prototipo es aquel que tiene un número no especificado de argumentos, que se representa por tres puntos (`...`)

EJEMPLO 7.2 Calcular el área de un rectángulo. El programa se descompone en dos funciones, además de `main()`.

[illegible]

```

int main()
{
    float b, h;
    printf("\n Base del rectangulo: ");
    b = entrada();
    printf("\n Altura del rectangulo: ");
    h = entrada();
    printf("\n Area del rectangulo: %.2f",area_rectangulo(b,h));
    return 0;
}

/* devuelve número positivo */
float entrada()
{
    float m;
    do {
        scanf("%f",&m);
    } while (m <= 0.0);
    return m;
}

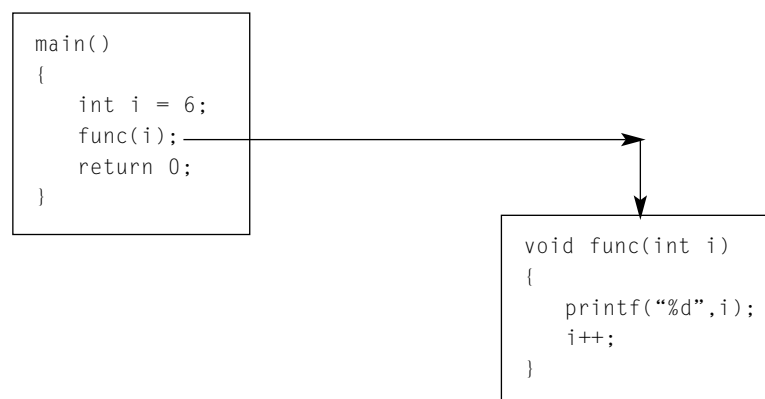
/* calcula el area de un rectángulo */
float area_rectangulo(float b, float a)
{
    return (b*a);
}

```

7.4 Parámetros de una función

C siempre utiliza el método de *parámetros por valor* para pasar variables a funciones. Para que una función devuelva un valor a través de un argumento hay que pasar la dirección de la variable, y que el argumento correspondiente de la función sea un puntero, es la forma de conseguir en C un paso de *parámetro por referencia*. C permite utilizar punteros para implementar parámetros por referencia, ya que por defecto, en C el paso de parámetros es por valor. Los parámetros por valor reciben copias de los valores de los argumentos que se les pasan. La asignación a parámetros valor de una función nunca cambian el valor del argumento original pasado a los parámetros. Los parámetros por referencia (declarados con *, *punteros*) reciben la dirección de los argumentos pasados; a éstos les debe de preceder del operador &, excepto los arrays. En una función, las asignaciones a parámetros referencia (punteros) cambian los valores de los argumentos originales. Con el objeto de añadir seguridad adicional a las funciones, se puede añadir a la descripción de un parámetro el especificador `const`, que indica al compilador que son sólo para pasar información al interior de la función. Si se intenta modificar este parámetro se producirá un mensaje de error.

EJEMPLO 7.3 Esquema del paso de parámetros por valor en la llamada a una función.



EJEMPLO 7.4. *Esquema de paso de parámetros por referencia.*

```
int i = 3, j = 50;
printf("i = %d y j = %d \n", i,j);
intercambio(&i, &j);
printf("i = %d y j = %d \n", i,j);

void intercambio(int* a, int* b)
{
    int aux = *a;
    *a      = *b;
    *b      = aux;
}
```

7.5 Funciones en línea, macros con argumentos

Las funciones en línea se usan cuando la función es una expresión (su código es pequeño y se utiliza muchas veces en el programa). Realmente no son funciones, el preprocesador expande o sustituye la expresión cada vez que es llamada. La sintaxis general es :

```
#define NombreMacro(parámetros sin tipos) expresión_texto
```

La definición ocupará sólo una línea, aunque si se necesita más texto, se puede situar una barra invertida (\) al final de la primera línea y continuar en la siguiente, en caso de ser necesarias más líneas proceder de igual forma; de esa manera se puede formar una expresión más compleja. Entre el nombre de la macro y los paréntesis de la lista de argumentos no puede haber espacios en blanco. Es importante tener en cuenta que en la macros con argumentos *no hay comprobación de tipos*.

EJEMPLO 7.5 *Función en línea para definir una función matemática.*

```
#include <stdio.h>
#define fesp(x) (x*x + 2*x -1)

void main()
{
    float x;
    for (x = 0.0; x <= 6.5; x += 0.3)
        printf("\t f(%.1f) = %6.2f ",x, fesp(x));
}
```

7.6 Ámbito (alcance)

El ámbito es la zona de un programa en la cual es visible una variable. Existen cuatro tipos de ámbitos: *programa*, *archivo fuente*, *función* y *bloque*. Normalmente la posición de la sentencia en el programa determina el ámbito.

- Las variables que tienen *ámbito de programa* pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman *variables globales*. Para hacer una variable global, declárela simplemente al principio de un programa, fuera de cualquier función.
- Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra reservada `static` tiene *ámbito de archivo fuente*. Las variables con este ámbito se pueden referenciar desde el punto del programa en que están declaradas hasta el final del archivo fuente.
- Una variable que tiene ámbito de una función se puede referenciar desde cualquier parte de la función. Las variables declaradas dentro del cuerpo de la función se dice que son *locales* a la función.

- Una variable declarada en un bloque tiene *ámbito de bloque* y puede ser referenciada en cualquier parte del bloque, desde el punto en que está declarada hasta el final del bloque. Las variables locales declaradas dentro de una función tienen ámbito de bloque de la función; no son visibles fuera del bloque.

EJEMPLO 7.6 *Declaración de variables y funciones en ámbitos diferentes.*

```
int i;                /*Ambito de programa */
static int j;         /*Ambito de archivo */
float func(int k)      /* K ámbito de función */
{
    {
        int m;        /*Ambito de bloque */
        ...
    }
}
```

7.7 Clases de almacenamiento

Los especificadores de clases (tipos) de almacenamiento permiten modificar el ámbito de una variable. Los especificadores pueden ser uno de los siguientes: `auto`, `extern`, `register`, `static` y `typedef`.

Variables Automáticas. Las variables que se declaran dentro de una función se dice que son automáticas (`auto`), significando que se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función. La palabra reservada `auto` es opcional.

EJEMPLO 7.7 *Declaración de variables automáticas.*

```
auto int x1;          es igual que      int x1;
```

Variables Externas. Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro archivo fuente.

EJEMPLO 7.8 *Declara que la función leerReal() está implementada en otro archivo fuente. También, declara la variable f definida en otro archivo.*

```
/* archivo fuente exter1.c */
#include <stdio.h>
extern void leerReal(void); /* función definida en otro archivo; no es estrictamente
                             necesario extern, se asume por defecto */

float f;
int main()
{
    leerReal();
    printf("Valor de f = %f",f);
    return 0;
}

/* archivo fuente exter2.c */
#include <stdio.h>
void leerReal(void)
{
    extern float f;          /* variable definida en otro archivo (extern1.c) */
    printf("Introduzca valor en coma flotante: ");
    scanf("%f",&f);
}
```

En el archivo `extern2.c` la declaración externa de `f` indica al compilador que `f` se ha definido en otra parte (archivo). Posteriormente, cuando estos archivos se enlacen, las declaraciones se combinan de modo que se referirán a las mismas posiciones de memoria.

Variables registro. Precediendo a la declaración de una variable con la palabra reservada `register`, se sugiere al compilador que la variable se almacene en uno de los registros *hardware* del microprocesador. Para declarar una variable registro, hay que utilizar una declaración similar a: `register int k;`. Una variable registro debe ser local a una función, nunca puede ser global al programa completo.

Variables estáticas. Las *variables estáticas* no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Al contrario que las variables locales normales, una variable `static` se inicializa sólo una vez y se declaran precediendo a la declaración de la variable con la palabra reservada `static`.

EJEMPLO 7.9 Se declaran dos variables con almacenamiento permanente (*static*).

```
func_uno()
{
    int i;
    static int j = 25;           /*j, k variables estáticas */
    static int k = 100;
    ...
}
```

7.8 Concepto y uso de funciones de biblioteca

Todas las versiones de C ofrecen una biblioteca estándar de funciones que proporciona soporte para operaciones utilizadas con más frecuencia. Las *funciones estándar* o *predefinidas*, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo *archivo de cabecera*. Los nombres de los *archivos de cabecera* estándar utilizados en los programas se muestran a continuación encerrados entre corchetes tipo ángulo:

<code><assert.h></code>	<code><ctype.h></code>	<code><errno.h></code>	<code><float.h></code>
<code><limits.h></code>	<code><locale.h></code>	<code><math.h></code>	<code><setjmp.h></code>
<code><signal.h></code>	<code><stdarg.h></code>	<code><stddef.h></code>	<code><stdio.h></code>
<code><stdlib.h></code>	<code><string.h></code>	<code><time.h></code>	

7.9 Miscelánea de funciones

Funciones de carácter. El archivo de cabecera `<ctype.h>` define un grupo de funciones/macros de manipulación de caracteres. Todas las funciones devuelven un resultado de valor *verdadero* (distinto de cero) o *falso* (cero).

Funciones numéricas. Virtualmente cualquier operación aritmética es posible en un programa C. Las funciones matemáticas disponibles son las siguientes: trigonométricas; logarítmicas; exponenciales; funciones matemáticas de carácter general; aleatorias. La mayoría de las funciones numéricas están en el archivo de cabecera `math.h`; las funciones de *valor absoluto* `abs` y `labs` están definidas en `stdlib.h`, y las funciones de *división entera* `div` y `ldiv` también están en `stdlib.h`.

Funciones de fecha y hora. Los microprocesadores tienen un sistema de reloj que se utiliza principalmente para controlar el microprocesador, pero se utiliza también para calcular la fecha y la hora. El archivo de cabecera `time.h` define estructuras, macros y funciones para manipulación de fechas y horas. La fecha se guarda de acuerdo con el calendario gregoriano (mm/dd/aa). Las funciones `time` y `clock` devuelven, respectivamente, el número de segundos desde la *hora base* y el tiempo de CPU empleado por el programa en curso.

Funciones de utilidad. El lenguaje C incluye una serie de funciones de utilidad que se encuentran en el archivo de cabecera `stdlib.h` como las siguientes: `abs(n)`, que devuelve el valor absoluto del argumento `n`; `atof(cad)` convierte los dígitos de la cadena `cad` a número real; `atoi(cad)`, `atol(cad)` convierte los dígitos de la cadena `cad` a número entero y entero largo respectivamente.

Visibilidad de una función. El *ámbito* de un elemento es su visibilidad desde otras partes del programa y la *duración* de un objeto es su tiempo de vida, lo que implica no sólo cuánto tiempo existe la variable, sino cuando se crea y cuando se hace

disponible. El ámbito de un elemento en C depende de donde se sitúe la definición y de los modificadores que le acompañan. Se puede decir que un elemento definido dentro de una función tiene *ámbito local* (alcance local), o si se define fuera de cualquier función, se dice que tiene un *ámbito global*.

Compilación separada. Los programas grandes son más fáciles de gestionar si se dividen en varios archivos fuente, también llamados *módulos*, cada uno de los cuales puede contener una o más funciones. Estos módulos se compilan y enlazan por separado posteriormente con un *enlazador*, o bien con la herramienta correspondiente del entorno de programación. Cuando se tiene más de un archivo fuente, se puede referenciar una función en un archivo fuente desde una función de otro archivo fuente. Al contrario que las variables, las funciones son externas por defecto. De modo opcional y por razones de legibilidad, puede utilizar la palabra reservada `extern` con el prototipo de función.

PROBLEMAS RESUELTOS

- 7.1.** La función factorial se define de la siguiente forma. $Factorial(n) = 1$ si $n=0$, y $factorial(n) = n * factorial(n-1)$ si $n>0$. Escriba la función factorial, y un programa que la llame para distintos valores de n .

Análisis del problema

La función factorial, se programa no recursivamente, usando un bucle ascendente, inicializando un acumulador a 1 y multiplicando en cada iteración el acumulador por la variable de control del bucle.

Codificación

```
#include "stdio.h"
float factorial (int x);

void main (void )
{
    float x,y,i;
    printf(" dame dos números ");
    scanf("%f%f",&x,&y);
    for (i = x; i <= y; i++)
    {
        printf("%8.0f %s %8.0f\n",i,"factorial", factorial(i));
    }
}

float factorial (int x)
{
    float i,f;
    f = 1.0 ;
    for ( i = 1; i <= x; i++)
        f = f * i;
    return (f);
}
```

- 7.2.** Escriba una función que intercambie el valor de dos números enteros y un programa que realice las llamadas.

Análisis del problema

Se escribirá una función `inter1()` que intercambia el valor de los números enteros, realizando la transmisión de parámetros por referencia.

Codificación

```
#include "stdio.h";
void main(void )
{
    int x,y;
    printf(" dame dos enteros \n");
    scanf(" %d %d",&x,&y);
    interl(&x,&y);
    printf (" cambiados %d %d \n",x,y);
}
void interl( int *a, int *b)
{
    int aux ;
    aux = *a;
    *a = *b;
    *b = aux;
}
```

7.3. *Escriba dos macros que permitan calcular el área lateral y el volumen de un cilindro.*

Análisis del problema

El volumen de un cilindro viene dado por $\text{volumen} = \pi \cdot \text{radio}^2 \cdot \text{altura}$ y el Areatotal viene dada por $\text{Areatotal} = 2 \cdot \pi \cdot \text{radio} \cdot \text{altura} + \pi \cdot \text{radio}^2$. Para resolver el problema basta con declarar las variables correspondientes declarar la constante π y los dos macros que permitan definir las dos funciones, con lo que la codificación queda de la siguiente forma.

Codificación

```
#include <stdio.h>
const float Pi = 3.141592;
#define VOLCILINDRO(radio,altura) ((Pi*(radio*radio)*altura))
#define AREATOTAL(radio,altura) (2*Pi*radio*altura+Pi*radio*radio)

int main()
{
    float radio, altura, volumen, Areatotal;
    do
    {
        printf("Introduzca radio del cono:\n ");
        scanf("%f",&radio);
        printf("Introduzca altura del cono:\n ");
        scanf("%f",&altura);
    }while (( radio <= 0) || (altura <= 0));
    volumen = VOLCILINDRO(radio, altura);
    Areatotal = AREATOTAL(radio, altura);
    printf("El volumen del cilindro es: %f\n",volumen);
    printf("El area tetla n del cilindro es: %f\n",Areatotal);
    return 0;
}
```

7.4. *Escriba una función que lea tres números enteros del teclado y devuelva el mayor.*

Análisis del problema

La función que resuelve el problema no tiene ningún parámetro, y lo que hace es leer secuencialmente los tres números y calcular el mayor. Se codifica también un programa principal que se encarga de llamar a la función anterior.

Codificación

```
#include "stdio.h"
int  mayor ();

void  main (void )
{
    printf(" el mayor es %d ",mayor());
}

int  mayor ()
{
    int i,m ;
    printf(" dame numero \n");
    scanf("%d", &m);
    printf(" dame numero \n");
    scanf("%d", &i);
    if (i > m)
        m = i;
    printf(" dame numero \n");
    scanf("%d", &i);
    if (i > m)
        m = i;
    return (m);
}
```

7.5. *Escriba un programa que calcule los valores de la función `funcionx` definida de la siguiente forma:*

$funcionx(0) = 0,$
 $funcionx(1) = 1$
 $funcionx(2) = 2$
 $funcionx(n) = funcionx(n-3) + 2*funcionx(n-2) + funcionx(n-1) \quad \text{si } n > 2.$

Análisis del problema

La función `funcionx` está definida recursivamente. Para programarla se puede usar la idea de definir tres variables locales estáticas que guarden los últimos valores obtenidos de la función. De esta forma si se le llama desde un bucle *desde*, *hasta* el valor que se quiera calcular se obtiene la lista de valores de la función que es la siguiente 0, 1, 2, 4, 9, 19, 41,.....

Codificación

```
#include <stdio.h>
long int funcionx();
int main()
{
    int n,i;
    printf("Cuantos numeros de la funcionx ?: ");
```

```

scanf("%d",&n);
printf("\nSecuencia de funcionx: 0,1,2");
for (i = 3; i <= n; i++)
    printf(",%d",funcionx());
return 0;
}
long int funcionx()
{
    static int x = 0;
    static int y = 1;
    static int z = 2;
    int aux;
    aux = x + 2 * y + z ;
    x = y;
    y = z;
    z = aux;
    return z;
}

```

- 7.6.** *Escriba una función que tenga como parámetro dos números enteros positivos n y m , y calcule el cociente de la división entera del mayor de ellos entre el menor mediante sumas y restas.*

Análisis del problema

Un programa principal leerá los dos números y llamará a la función `cociente` que se encargará de resolver el problema. La función `cociente` determina el mayor y el menor de los dos números almacenándolos en las variables `Mayor` y `menor`. Mediante un acumulador inicializado a la variable `menor` y un contador, `c`, inicializado a cero, se cuenta el número de veces que es necesario sumar el `menor` para sobrepasar (ser estrictamente mayor) el número `Mayor`. Como `c` se ha inicializado a cero, cuando en el acumulador ya se ha sumando una vez el `menor`, el resultado final solicitado será el dado por el acumulador `c`.

Codificación

```

#include "stdio.h"
int  cociente (int n, int m);
void main (void )
{
    int n,m;
    do
    {
        printf(" dame dos numeros :");
        scanf("%d %d",&n,&m);
    } while ((n <= 0) || (m <= 0));
    printf(" el cociente es %d \n", cociente(n,m));
}

int  cociente (int n, int m)
{
    int c, Mayor, menor, acu ;
    if (n < m)
    {
        Mayor = m;
        menor = n;
    }
}

```

```

    }
    else
    {
        Mayor = n;
        menor = m;
    }
    acu = menor;
    c = 0;
    while (acu <= Mayor)
    {
        acu += menor;
        c++;
    }
    return (c);
}

```

- 7.7.** *Escriba una función que tenga como parámetro dos números enteros positivos n y m , y calcule el resto de la división entera del mayor de ellos entre el menor mediante suma y restas.*

Análisis del problema

Un programa principal leerá los dos números asegurándose que son positivos mediante un bucle `do-while` y llamará a la función `resto` que se encargará de resolver el problema. La función `resto`, primeramente, determina el mayor y el menor de los dos números almacenándolos en las variables `Mayor` y `menor`. Mediante un acumulador inicializado a la variable `menor` y mediante un bucle `while` se suma al acumulador el valor de `menor`, hasta que el valor del acumulador sea mayor que el número `Mayor`. Necesariamente el resto debe ser el valor de la variable `Mayor` menos el valor de la variable acumulador `acu` menos el valor de la variable `menor`.

Codificación

```

#include "stdio.h"
int resto(int n, int m);

void main (void )
{
    int n, m;
    do
    {
        printf(" dame dos numeros :");
        scanf("%d %d",&n,&m);
    } while ((n <= 0) || (m <= 0));
    printf(" el cociente es %d \n", resto(n,m));
}

int resto(int n, int m)
{
    int Mayor, menor, acu ;
    if (n < m)
    {
        Mayor = m;
        menor = n;
    }
    else

```



```

{
    Mayor = n;
    menor = m;
}
acu = menor;
while (acu <= Mayor)
    acu += menor;
return (Mayor - acu - menor);
}

```

7.8. Escriba un programa que calcule los valores de la función *funcionx* definida de la siguiente forma:

$funcionx(0) = 0,$
 $funcionx(1) = 1$
 $funcionx(2) = 2$
 $funcionx(n) = funcionx(n-3) + 2*funcionx(n-2) + funcionx(n-1) \quad \text{si } n > 2.$

Análisis del problema

El ejercicio número 7.5 se ha programado mediante variables `static` aquí se resuelve mediante un bucle, y sólo se retorna el resultado final.

Codificación

```

#include <stdio.h>
long int funcionx(int n);
int main()
{
    int n,i;
    printf("Cuantos numeros de la funcionx ?: ");
    scanf("%d",&n);
    printf("\nSecuencia de funcionx: 0,1,2");
    for (i = 3; i <= n; i++)
        printf(" ,%d",funcionx(i));
    return 0;
}

long int funcionx(int n)
{
    long int x = 0 ,y = 1,z = 2, i, aux;
    if (n <= 2)
        return (n);
    else
    {
        for (i = 3; i <= n;i++)
        {
            aux = x + 2 * y + z ;
            x = y;
            y = z;
            z = aux;
        }
        return z;
    }
}

```

7.9. *Escriba una función que calcule la suma de los divisores de un número entero positivo.*

Análisis del problema

La función `divisores` calculará la suma de todos los divisores del número incluyendo el uno y el propio número. Para realizarlo basta con inicializar un acumulador a cero, y mediante un bucle `for` recorrer todos los números naturales desde el uno hasta el propio `n`, y cada vez que un número sea divisor de `n` sumarlo al acumulador correspondiente. En la codificación se incluye un programa principal que se encarga de leer el número y de llamar a la función `divisores`.

Codificación

```
#include "stdio.h"
int  divisores (int n);

void  main (void )
{
    int n;
    do
    {
        printf(" dame un numero :");
        scanf("%d",&n);}
    } while (n <= 0);
    printf("la suma de divisores es %d \n",divisores(n));
}

int  divisores(int n)
{
    int i, acu;
    acu = 0;
    for(i = 1; i <= n; i++)
        if (n % i == 0)
            acu += i;
    return (acu);
}
```

7.10. *Escriba una función que decida si un número es perfecto.*

Análisis del problema

Se programa la función `perfecto`, de tal manera que sólo se suman los posibles divisores del número `n` que recibe como parámetro comprendido entre 1 y $n - 1$. Esta función es de tipo lógico, y por lo tanto devuelve el valor de la expresión `acu==n`.

Codificación

```
int  perfecto(int n)
{
    int i,acu ;
    acu = 0;
    for(i = 1; i < n; i++)
        if (n % i == 0)
            acu += i;
    return (acu == n);
}
```

- 7.11.** *Escriba una función que decida si dos número enteros positivos son amigos. Dos números son amigos, si la suma de los divisores distintos de sí mismo de cada uno de ellos coincide con el otro número. Ejemplo 284 y 220 son dos números amigos.*

Análisis del problema

Para resolver el problema basta con usar la función `divisores` implementada en el ejercicio 7.9 y escribir la función `amigos` que detecta la condición. También se presente un programa principal que llama a las funciones.

Codificación

```
#include "stdio.h"
int divisores (int n);
int amigos (int n, int m);

void main (void )
{
    int n, m;
    do
    {
        printf(" dame dos numeros :");
        scanf("%d %d",&n, &m);
    } while ((n <= 0) || (m <= 0));
    if (amigos(n, m))
        printf(" los numeros %d  %d son amigos\n", n);
    else
        printf(" loa numeroa %d  %d no son amigos\n", n);
}

int divisores(int n)
{
    int i,acu ;
    acu=0;
    for(i = 1; i < n; i++)
        if (n % i == 0)
            acu += i;
    return (acu);
}

int amigos (int n, int m)
{
    return ((n == divisores(m)) && (m == divisores(n)));
}
```

- 7.12.** *Escriba una función que decida si un número entero positivo es primo.*

Análisis del problema

Un número entero positivo es primo, si sólo tiene por divisores la unidad y el propio número. Una posible forma de resolver el problema consiste en comprobar todos los posibles divisores desde el dos hasta uno menos que el dado. El método que se usa, aplica la siguiente propiedad: “si un número mayor que la raíz cuadrada de n divide al propio n es porque hay otro número entero menor que la raíz cuadrada que también lo divide”. Por ejemplo: si n vale 64 su raíz cuadrada es 8, el número 32 divide a 64 que es mayor que 8 pero también lo divide el número 2 que es menor que 8, ya que $2 \cdot 32 = 64$. De esta forma para decidir si un número es primo basta con comprobar si tiene divisores menores o iguales que su raíz cuadrada por supuesto eliminando la unidad. El programa que se codifica a continuación, usa esta propiedad, en la función lógica `primo`.

Codificación

```

#define TRUE 1
#define FALSE 0

int primo(int n)
{
    int i,tope, p;
    p = TRUE;
    i = 2;
    printf("%d",tope);
    while (p && (i <= tope))
    {
        p =!( n % i == 0);
        i++;
    }
    return (p);
}

```

- 7.13.** Se define un número c elevado a un número entero n ($n > 0$) como el producto de c por sí mismo n veces. Escriba un programa que calcule la potencia de varios números.

Análisis del problema

Se programa la función `potencia`, mediante un bucle que multiplica por sí mismo el primer parámetro tantas veces como indique el segundo. El programa principal lee el exponente n y calcula la potencia de varios números.

Codificación

```

#include "stdio.h"
float potencia (float a, int n);

void main (void )
{
    float x,y,i;
    int n;
    printf(" dame dos numeros ");
    scanf("%f%f",&x,&y);
    printf(" dame el exponente entero");
    scanf("%d",&n);
    for (i = x; i <= y; i+=1.0)
    {
        printf("%8.0f %s %8d %s ",i, "elevado a", n,"es");
        printf("%8.0f\n",potencia(i,n));
    }
}

float potencia (float a, int n)
{
    float i,f;
    f = 1.0;
    for ( i = 1.0; i <= n; i+=1.0)
        f = f * a;
}

```

```
    return (f);
}
```

- 7.14.** Escriba una función para calcular las coordenadas x e y de la trayectoria de un proyectil de acuerdo a los parámetros ángulo de inclinación α y velocidad v a intervalos de 0.1 s.

Análisis del problema

Las fórmulas que dan las coordenadas x e y del proyectil son:

$$x = v \cdot \cos(\alpha) \cdot t$$

$$y = v \cdot \sin(\alpha) \cdot t - a \cdot t^2 / 2$$

donde α es un ángulo que está en el primer cuadrante v es la velocidad inicial y $a = 40 \text{ m/s}^2$ es la aceleración. La función debe terminar cuando y valga cero.

Codificación

```
void tratectoria(float a, float v)
{
    float t, x, y;
    printf("      x      y");
    printf("      0      0");
    t = 0.1;
    y = 1.0 ;
    while (y > 0)
    {
        x = v * cos(a) * t;
        y = v * sin(a) * t - 40 / 2 * t * t;
        printf(" %f %f \n", x, y);
        t = t + 0.1;
    }
}
```

- 7.15.** Se define el número combinatorio m sobre n de la siguiente forma: .

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

Escriba un programa que lea los valores de m y de n y calcule el valor de m sobre n .

Análisis del problema

El programa se codifica usando la función `factorial`, y programando la función `combinatorio`, con sus correspondientes llamadas. El programa principal se encarga de leer los datos m y n y de llamar a la función `combinatorio`.

Codificación

```
#include "stdio.h"
int factorial (int x);
int combinatorio( int m, int n);

void main(void )
```

```

{
    int m, n, s;
    printf(" dame dos numeros enteros ");
    scanf("%d%d",&m, &n);
    if (m < n)
    {
        printf("%8i %s %8.i %s",n,"sobre",m,"=");
        printf("%8i\n", combinatorio(n,m));
    }
    else
    {
        printf("%8i %s %8.i %s",m,"sobre",n,"=");
        printf("%8i\n", combinatorio(m,n));
    }
}

int factorial (int x)
{
    int i,f ;
    f = 1 ;
    for ( i = 1; i <= x; i++)
        f = f * i;
    return (f);
}

int combinatorio( int m,int n)
{
    return( factorial(m) / ( factorial(n) * factorial(m - n)));
}

```

- 7.16.** Dado un número real p entre cero, un número entero n positivo, y otro número entero i comprendido entre 0 y n , se sabe que si un suceso tiene probabilidad de que ocurra p , y el experimento aleatorio se repite n veces, la probabilidad de que el suceso ocurra i veces viene dado por la función binomial de parámetros n , p e i dada por la siguiente fórmula.

$$\text{Probabilidad } (X = i) = \binom{n}{i} p^i (1 - p)^{n-i}$$

Escriba un programa que lea los valores de p , n e i , y calcule el valor dado por la función binomial.

Análisis del problema

El problema usa la función factorial, combinatorio, potencia, y además la binomial programada de acuerdo con la fórmula.

Codificación

```

#include "stdio.h"
float factorial (int x);
float combinatorio( int m,int n);
float potencia (float a, int n);
float binomial( float p, int i, int n);

void main (void )
{

```

```

int n, i;
float p;
do
{
    printf(" dame probabilidad p y valor de n ");
    scanf("%f %i",&p,&n);
} while ((p <= 0) || (p >= 1) || (n <= 0));
do
{
    printf(" dame valor de i entre 0 y el valor de n \n");
    scanf("%i",&i);
} while (( i < 0 ) || ( i > n ));
printf("%8f\n", binomial(p, i, n));
}

float factorial (int x)
{
    float i, f;
    f = 1 ;
    for ( i = 1; i <= x; i++)
        f = f*i;
    return (f);
}

float combinatorio( int m, int n)
{
    float x ;
    x = factorial(m)/( factorial(n)*factorial(m-n));
    return(x);
}

float potencia (float a, int n)
{
    float i,f ;
    f = 1.0 ;
    for ( i = 1; i <= n; i++)
        f = f * a;
    return (f);
}

float binomial( float p, int i, int n)
{
    return(combinatorio(n,i) * potencia(p,i) * potencia(1-p,n-i));
}

```

7.17. Escriba un programa que mediante funciones calcule:

- Las anualidades de capitalización si se conoce el tiempo, el tanto por ciento y el capital final a pagar.
- El capital c que resta por pagar al cabo de t años conociendo la anualidad de capitalización y el tanto por ciento.
- El número de años que se necesitan para pagar un capital c a un tanto por ciento r .

Análisis del problema

El programa se codifica de la siguiente forma: la función `menu`, se encarga de realizar las llamadas a los distintos apartados del problema. La función `aa` calcula la anualidad de capitalización, teniendo en cuenta que viene dada por:

$$aa = \frac{cr}{(1+r)((1+r)^t - 1)}$$

La función `cc` calcula el apartado segundo teniendo en cuenta que viene dada por la fórmula:

$$cc = a(1+r) \left(\frac{(1+r)^t - 1}{r} \right)$$

La función `tt` calcula el tercer apartado, teniendo en cuenta que viene dada por la fórmula:

$$tt = \frac{\log \left(1 + \frac{cr}{a(1+r)} \right)}{\log(1+r)}$$

El programa principal, se encarga de leer los datos, y realizar las distintas llamadas.

Codificación

```
#include <math.h>
#include <stdio.h>
float cc (float r, float t, float a);
float tt( float r, float a, float c);
float aa( float r, float t, float c);
int menu(void );

void main (void )
{
    int opcion;
    float c, r, t, a;
    for (;;)
    {
        char sigue;
        opcion = menu();
        switch (opcion)
        {
            case 1: printf( " dame r   t   y   a\n");
                    scanf("%f%f%f",&r,&t,&a);
                    c = cc(r / 100, t, a);
                    printf(" capital = %f \n", c);
                    scanf("%c",&sigue);
                    break;
            case 2: printf( " dame r   t   y   c\n");
                    scanf("%f%f%f",&r,&t,&c);
                    a= aa(r / 100, t, c);
                    printf(" anualidad = %f \n", a);
                    scanf("%c",&sigue);
                    break;
```



```

        case 3: printf( " dame r   a   y   c\n");
                scanf("%f%f%f",&r,&a,&c);
                t= tt( r / 100, a, c);
                printf( " años = %f \n ", t);
                scanf("%c",&sigue);
                break;
        case 4: exit();
                break;
    }
}

float cc( float r, float t, float a)
{
    return ( a* (1 + r) * ( pow(1 + r, t) - 1) / r);
}

float aa( float r, float t, float c)
{
    return (c * r / ( (1 + r) * ( pow(1 + r, t) - 1)));
}

float tt( float r, float a, float c)
{
    float x;
    x = c * r / (a * (1 + r));
    return (log( 1 + x) / log( 1 + r));
}

int menu(void )
{
    char s[80];
    int  c;
    printf(" 1. calcular  anualidad A de capitalización \n");
    printf(" 2. calcular  el capital C al cabo de t años \n");
    printf(" 3. calcular  el numero de años \n");
    printf(" 4   fin \n");
    do
    {
        printf (" introduzca opción \n");
        gets(s);
        c = atoi(s);
    } while (c < 0 || c > 4);
    return c;
}

```

7.18 La ley de probabilidad de que ocurra el suceso r veces de la distribución de Poisson de media m viene dado por:

$$\text{Probabilidad } (X = r) = \frac{m^r}{r!} e^{-m}$$

Escriba un programa que calcule mediante un menú el valor de:

- a) El suceso ocurra exactamente r veces.
- b) El suceso ocurra a lo sumo r veces.
- c) El suceso ocurra por lo menos r veces.

Análisis del problema

Se programa una función `menú` que elige entre las tres opciones.

Se programa una función `Poisson` que calcula

$$\text{Probabilidad}(X = r) = \frac{m^r}{r!} e^{-m}$$

y la

$$\text{Probabilidad}(X \leq r) = \sum_{i=0}^r \frac{m^i}{i!} e^{-m}$$

que resuelven el apartado a y el b.

Para resolver el apartado c basta con observar que:

$$\text{Probabilidad}(X \geq r) = 1 + \text{Probabilidad}(X = r) - \text{Probabilidad}(X \leq r)$$

El programa principal, lee los valores de r , el valor de la media m y llama al `menú`.

Codificación (Consultar la página web del libro)

7.19 La función seno viene definida mediante el siguiente desarrollo en serie.

$$\text{sen}(x) = \sum_{i=0}^n (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

Escriba una función que reciba como parámetro el valor de x así como una cota de error, y calcule el seno de x con un error menor que la cota que se le pase.

Análisis del problema

Las fórmula que calcula los valores de $\text{sen}(x)$ puede obtenerse de la siguiente forma:

$$\text{sen}(x) = t_1 + t_3 + t_5 + t_7 + t_9 + \dots$$

donde

$$t_1 = x \text{ y } t_i = -\frac{x \cdot x}{i(i-1)} t_{i-2}$$

La función seno se programa teniendo en cuenta las fórmulas anteriores, y además la parada se realiza, cuando se han sumado, como máximo 10 términos ($i=20$), o el siguiente término a sumarse tiene un valor absoluto menor que una cota de error que se le pasa como parámetro.

El programa principal lee los valores de `valor1`, `valor2`, `incremento`, y cota de error y llame a la función coseno de biblioteca y la compare con el valor calculado mediante la función programada, para los valores:

`valor1`, `valor1+incremento`, `valor1+ 2*incremento`,...

hasta que se sobrepase el `valor2`.

Codificación

```

#include <math.h>
#include <stdio.h>
float seno(float x, float error);

void main (void )
{
    float error,valor1,valor2, inc, x;
    do
    {
        printf (" dame valor1 valor2 inc error positivo ");
        scanf(" %f %f %f %f", &valor1, &valor2,&inc,&error);
    } while ( (valor1 > valor2) || (inc <0) || (error <0));
    for (x = valor1; x <= valor2; x += inc)
        printf(" %f      %f      %f      \n", x , sin(x), seno(x,error));
}

float seno( float x, float error)
{
    float term,suma,xx;
    int i, ii ;
    suma = x;
    i= 1;
    term = x;
    xx = x * x;
    while (fabs(term) > error && i < 20)
    {
        i += 2;
        term = -term * xx/(i * (i - 1));
        suma = suma + term;
    }
    return(suma);
}

```

7.20 La función clotoide viene definida por el siguiente desarrollo en serie, donde A y θ son datos.

$$x = A\sqrt{2\Theta} \sum_{i=0}^n (-1)^i \frac{\Theta^{2i}}{(4i+1)(2i)!}$$

Escriba un programa en calcule los valores de la clotoide para el valor de $A=1$ y para los valores de θ siguientes $0, \pi/20, 2\pi/20, 3\pi/20, \dots, \pi$. La parada de la suma de la serie, será cuando el valor absoluto del siguiente término a sumar sea menor o igual que $1 e^{-10}$.

Análisis del problema

Las fórmulas que calculan los distintos valores de x y de y , pueden obtenerse de la siguiente forma:

$$\begin{aligned}
 x &= t_0 + t_2 + t_4 + t_6 + t_8 + \dots \\
 y &= t_1 + t_3 + t_5 + t_7 + t_9 + \dots
 \end{aligned}$$

siendo $t_0 = A\sqrt{2\Theta}$ y $t_i = (-1)^{i-1} \frac{2i-1}{i(2i+1)} \Theta t_{i-1}$

Se programa la función `term`, que calcula t_i en función de t_{i-1} , i , θ .

Se programa la función `clotoide`, que recibe como datos el valor de a y el de θ y devuelve como resultado los valores de x e y .

El programa principal, calcula el valor de π mediante la función `arcocoseno`, para posteriormente mediante un bucle, realizar las distintas llamadas a la función `clotoide`.

Codificación (Se encuentra en la página web del libro)

7.21 La función coseno viene definida mediante el siguiente desarrollo en serie.

$$\cos(x) = \sum_{i=0}^n (-1)^i \frac{x^{2i}}{(2i)!}$$

Escribe una función que reciba como parámetro el valor de x así como una cota de error, y calcule el coseno de x con un error menor que la cota que se le pase. Compare el valor obtenido, por la función de biblioteca y la programada.

Análisis del problema

Las fórmula que calcula los valores de $\cos(x)$ puede obtenerse de la siguiente forma:

$$\cos(x) = t_0 + t_2 + t_4 + t_6 + t_8 + \dots$$

$$\text{donde } t_0 = 1 \text{ y } t_i = -\frac{x \cdot x}{i(i-1)} t_{i-2}$$

La función `coseno` se programa teniendo en cuenta las fórmulas anteriores, y además la parada se realiza, cuando se han sumado, como máximo 20 términos ($i=20$), o el siguiente término a sumarse tiene un valor absoluto menor que una cota de error que se le pasa como parámetro.

El programa principal lee los valores de `valor1`, `valor2`, `incremento`, y cota de error y llame a la función `coseno` de biblioteca y la compare con el valor calculado mediante la función programada, para los valores:

`valor1, valor1+ incremento, valor1+ 2*incremento ,...`

hasta que se sobrepase el `valor2`.

Codificación

```
#include <math.h>
#include <stdio.h>
float coseno(float x, float error);

void main (void )
{
    float error, valor1, valor2, inc, x;
    do
    {
        printf (" dame valor1 valor2 inc error positivo ");
        scanf(" %f %f %f %f", &valor1, &valor2, &inc, &error);
    } while ( (valor1 > valor2) || (inc < 0) || (error < 0));
    for (x = valor1; x <= valor2; x += inc)
        printf(" %f    %f    %f    \n", x ,cos(x), coseno(x, error));
}

float coseno( float x, float error)
```

```

{
    float term, suma, xx;
    int i, ii;
    suma = 1.0;
    i = 0;
    term = 1.0;
    xx = x * x;
    while (fabs( term ) > error && i < 20)
    {
        i += 2;
        term = -term * xx/(i * (i - 1));
        suma = suma + term;
    }
    return(suma);
}

```

7.22 La descomposición en base 2 de todo número, permite en particular que todo número en el intervalo (0,1), se pueda escribir como límite de la serie

$$\sum_{i=1}^n \pm \frac{1}{2^i}$$

donde la elección del signo $sg(i)$ depende del número que se trate.

El signo del primer término es siempre positivo. Una vez calculado los signos de los n primeros, para calcular el signo del siguiente término se emplea el esquema: signo es positivo $sg(n+1)=+1$ si se cumple:

$$2 \sum_{i=1}^n sg(i) \frac{1}{2^i} > x$$

en caso contrario, $sg(n+1)=-1$.

Escriba un programa que calcule el logaritmo en base dos de un número $x>0$ con un error absoluto menor o igual que ϵ (x y ϵ son datos).

Análisis del problema

Si x está en el intervalo (0,1) entonces $\log_2(x) = -\log_2(1/x)$. Si x es mayor que 2 entonces es obvio que $\log_2(x) = 1 + \log_2(x/2)$.

Por tanto para programar la función $\log_2()$ basta con tener en cuenta las propiedades anteriores. El problema se resuelve escribiendo las siguientes funciones:

- $Alog_2$ que calcula el logaritmo en base 2 de cualquier x , y que llamará a las funciones $alog01$, $alog12$, o bien $alog1i$.
- $Alog12$ se programa teniendo en cuenta el desarrollo de la serie en el intervalo $[1,2]$.
- La función $signo$ determina en el propio avance del desarrollo de la suma.
- La función $Alog01$, se programa, de acuerdo con la propiedad del intervalo (0,1). Llamará, o bien a la función $alog12$, o bien a la función $alog1i$, dependiendo de que $1/x$ sea menor, o igual que dos, o bien sea estrictamente mayor.
- La función $alog2i$, se programa de acuerdo con la propiedad de los números estrictamente mayores que dos.

El programa principal lee la cota de error, y llama a la función $alog2$ para varios valores de prueba. Se muestra el resultado obtenido de la función $alog2$ y con la función de librería $\log()$.

Codificación

```

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

```

```
float alog2(float x, float eps);
float alog01(float x, float eps);
float alog12(float x, float eps);
float alog2i(float x, float eps);

void main (void )
{
    float x, eps;
    do
    {
        printf( " dame cota de error positiva ");
        scanf("%f",&eps);
    } while (eps <= 0);
    printf( "      dato      log2      máquina \n");
    for ( x= 0.01; x <= 1.06; x += 0.05)
    {
        printf("%6.3f %10.6f",x, alog2(x, eps));
        printf("%10.6f\n",log( x ) / log(2));
    }
}

float alog2(float x, float eps)
{
    float valor;
    if( x<= 0)
    {
        printf ( " error logaritmo negativo ");
        exit(1);
    }
    else if ( x < 1)
        valor = alog01(x, eps);
    else if( x <= 2)
        valor = alog12(x, eps);
    else
        valor = alog2i(x, eps);
    return valor;
}

float alog01(float x, float eps)
{
    if ((1 / x) > 2.0)
        return (-alog2i(1 / x, eps));
    else
        return(-alog12(1 / x, eps));
}

float alog12(float x, float eps)
{
    float term = 1.0, suma = 1.0;
    l2 = log(2.0);
    while (fabs(term) > eps)
    {
        term = term / 2;
```

```

        if (exp(12 * suma ) <= x)
            suma += term;
        else
            suma -= term;
    }
    return(suma);
}

float alog2i(float x, float eps)
{
    float acu = 0;
    while (x > 2)
    {
        x = x / 2;
        acu = acu + 1;
    }
    acu = acu + alog12(x, eps);
    return (acu);
}

```

7.23 Escriba un programa para gestionar fracciones.

Análisis del problema

El problema se ha estructurado de la siguiente forma.

- Una función *mcd*, calcula el máximo común divisor de dos números naturales, mediante el conocido algoritmo de *Euclides*. Para ello convierte los dos números en positivos.
- Una función *mcm* calcula el mínimo común múltiplo de dos números enteros, usando la propiedad siguiente: el máximo común divisor, multiplicado por el mínimo común múltiplo de dos números coincide con el producto de ambos números.
- Una función *simplificaf*, que simplifica una fracción dejando siempre el signo negativo, en caso de que la fracción sea negativa en el numerador.
- Una función *leerf*, se encarga de leer una fracción, asegurándose de que el denominador sea distinto de cero y que además el numerador y denominador sean primos entre sí.
- Una función *escribef*, se encarga de escribir una fracción en la salida.
- Las funciones *sumaf*, *restaf*, *multiplicaf*, *dividaf*, se encarga de sumar, restar, multiplicar y dividir fracciones.
- La función *elige* se encarga de leer un número comprendido entre los dos que se le pasen como parámetro.
- La función *leerfracciones* se encarga de leer dos fracciones.
- Por último, el programa principal, mediante un menú llama a las distintas funciones.

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

- 7.1. Escriba una función `dígito` que determine si un carácter es uno de los dígitos, `0, 1, 2, ...`.
- 7.2. Realice un procedimiento que realice la conversión de coordenadas polares (r, a) a cartesianas. Nota: $x = r \cdot \cos(a)$, $y = r \cdot \sin(a)$.
- 7.3. Escriba una función que calcule la media de un conjunto de $n > 0$ números leídos del teclado.
- 7.4. Escriba una función que decida si un número entero es capicúa.
- 7.5. Escriba una función que sume los 30 primeros números impares.
- 7.6. Dado el valor de un ángulo escribir un programa que muestre el valor de todas las funciones trigonométricas correspondientes al mismo.
- 7.7. Escriba una función que calcule la suma de los 20 primeros números primos.
- 7.8. Escriba una función que encuentre y escriba todos los números perfectos menores que un valor constante `max`.
- 7.9. Escriba un programa que mediante funciones determine el área del círculo correspondiente a la circunferencia circunscrita de un triángulo del que se conocen las coordenadas de los vértices.
- 7.10. Escriba un programa que lea dos enteros positivos n , y b y mediante una función `CambiarBase` visualice la correspondiente representación del número n en la base b .

Recursividad

La recursividad (recursión) es la propiedad que posee una función de permitir que dicha función puede llamarse a sí misma. Se puede utilizar la recursividad como una alternativa a la iteración. La recursión es una herramienta poderosa e importante en la resolución de problemas y en programación. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo las llamadas suplementarias a las funciones; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver.

8.1 La naturaleza de la recursividad

Una función *recursiva* es aquella que se llama a sí misma bien directamente, o bien a través de otra función. En matemáticas existen numerosas funciones que tienen carácter recursivo de igual modo numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo. Una función que contiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es *recursiva*.

8.2 Funciones recursivas

Una función **recursiva** es una función que se invoca a sí misma de forma directa o indirecta. En **recursión directa** el código de la función `f()` contiene una sentencia que invoca a `f()`, mientras que en **recursión indirecta** `f()` invoca a la función `g()` que invoca a su vez a la función `p()`, y así sucesivamente hasta que se invoca de nuevo a la función `f()`. Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo puede no terminar nunca.

EJEMPLO 8.1 *Una función con recursividad directa*

```
int f( int x)
{
    if (x <= 0)
        return 2;
    else
```

```

    return( n + 2 * f(n - 2));
}

```

La recursividad indirecta se produce cuando una función llama a otra, que eventualmente terminará llamando de nuevo a la primera función. Puede generalizarse mediante tres funciones $f1()$, $f2()$, $f3()$. La función $f1()$ realiza una llamada a $f2()$. La función $f2()$ realiza una llamada a $f3()$. La función $f3()$ realiza una llamada a $f1()$. Cuando se implementa una función recursiva es preciso considerar una *condición de terminación (caso base)*, ya que en caso contrario la función continuaría indefinidamente, llamándose a sí misma y llegaría un momento en que la memoria se agotaría. En consecuencia, sería necesario establecer en cualquier función recursiva la *condición de parada* de las llamadas recursivas y evitar indefinidamente las llamadas.

EJEMPLO 8.2. *Implementación de dos funciones con recursividad indirecta.*

```

float f( float y);

float g(float y)
{
    if (y <= 3)
        return ( y );
    else
        return( y + f(y - 2));
}

float f ( float y)
{
    if (y <= 2)
        return ( y );
    else
        return( y + g(y - 2));
}

```

8.3 Recursión *versus* iteración

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección*. La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas a funciones. La recursión invoca repetidamente al mecanismo de llamadas a funciones y en consecuencia se necesita un tiempo suplementario para realizar cada llamada. Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Las funciones con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de desbordamiento (*overflow*). La iteración se produce dentro de una función de modo que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas. Toda función recursiva puede ser transformada en otra función con esquema iterativo, para ello a veces se necesitan pilas donde almacenar cálculos parciales y valores de variables locales. La razón fundamental para elegir la recursión es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de diseñar e implementar con algoritmos de este tipo.

EJEMPLO 8.3 *Escribir una función no recursiva que permita generar los números de fibonacci.*

La secuencia de números de fibonacci: 0, 1, 1, 2, 3, 5, 8, 13 ... se obtiene partiendo de los números 0, 1 y a partir de ellos cada número se obtiene sumando los dos anteriores:

$a_n = a_{n-1} + a_{n-2}$. La función fibonacci tiene dos variables `static`, `x` e `y`. Se inicializan `x` a 0 y `y` a 1, a partir de esos valores se calcula el número de fibonacci actual, `y`, dejando preparado `x` para la siguiente llamada. Al ser variables `static` mantienen el valor entre llamada y llamada:

```

long int fibonacci()
{
    static int x = 0;
    static int y = 1;
    y = y + x;
    x = y - x;
    return y;
}

/* bucle para escribir n números de fibonacci */
for (i = 2; i < n; i++)
    printf("%ld", fibonacci());

```

8.4 Recursión infinita

La **recursión infinita** se produce cuando una llamada recursiva realiza otra llamada recursiva y ésta a su vez otra llamada recursiva y así indefinidamente. El flujo de control de una función recursiva requiere para una terminación normal distinguir los casos generales y triviales:

- Al caso general de un problema debe proporcionar una solución general mediante la realización de una o varias llamadas recursivas para un subproblema o subproblemas más pequeño.
- Al caso o casos triviales, debe proporcionar una solución trivial que no incluya llamadas recursivas.

8.5 Algoritmos divide y vencerás

El diseño de algoritmos basados en la técnica *divide y vence* consiste en transformar (*dividir*) un problema de tamaño n en problemas más pequeños, de tamaño menor que n pero similares al problema original, de modo que resolviendo los subproblemas y combinando las soluciones se pueda construir fácilmente una solución del problema completo (*vencerás*). Normalmente, el proceso de división de un problema en otros de tamaño menor conduce a problemas unitarios, *caso base*, cuya solución es inmediata. A partir de la obtención de la solución del problema para el *caso base*, se combinan soluciones que amplían el tamaño del problema resulto, hasta que el problema original queda resuelto. La implementación de estos algoritmos se puede realizar con funciones recursivas.

PROBLEMAS RESUELTOS

8.1. *Escriba una función recursiva para calcular el factorial de un número entero positivo.*

Análisis del problema

Si n es un número positivo se sabe que $0! = 1$ y si $n > 0$ entonces se tiene que $n! = n \cdot (n-1)!$. La función `factorial` codifica el algoritmo recursivamente.

Codificación

```

int factorial(int n)
{
    int aux;

```

```

    if (n <= 1)
        aux = 1;
    else
        aux = n * factorial(n - 1);
    return(aux);
}

```

8.2. *Escriba una función que calcule la potencia a^n recursivamente, siendo n positivo.*

Análisis del problema

La potencia de a^n es 1 si $n = 0$ y es $a * n^{n-1}$ en otro caso.

Codificación

```

float potencia(float a, int n)
{
    if (n <= 0)
        return( 1 );
    else
        return(a * potencia(a, n - 1));
}

```

8.3. *Escriba una función que dado un número entero positivo n calcule el número de fibonacci asociado.*

Análisis del problema

Los números de fibonacci son 0, 1, 1, 2, 3, 5, 8, 13, ... donde para $n=0$ $fibonacci(0)=0$, para $n=1$ $fibonacci(1)=1$, y si $n>1$ tenemos que $fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$.

Codificación

```

int fibonacci( int n)
{
    if (n <= 1)
        return(n);
    else
        return(fibonacci(n - 1)+ fibonacci(n - 2));
}

```

8.4. *Escriba una función recursiva que calcule el cociente de la división entera de n entre m , siendo m y n dos números enteros positivos recursivamente.*

Análisis del problema

El cociente de la división entera de n entre m , siendo ambos números enteros positivos se calcula de la siguiente forma si $n < m$ entonces $cociente(n,m) = 0$, si $n \geq m$ entonces $cociente(n, m) = 1 + cociente(n - m, m)$.

Codificación

```

int cociente( int n, int m)
{
    if (n < m)

```

```

        return(0);
    else
        return(1 + cociente(n - m, m));
}

```

- 8.5.** Escriba una función recursiva que calcule los valores de la función *funcionx* definida de la siguiente forma:

funcionx(0) = 0,

funcionx(1) = 1

funcionx(2) = 2

funcionx(n) = *funcionx*(n-3) + 2**funcionx*(n-2) + *funcionx*(n-1) si $n > 2$.

Análisis del problema

La función *funcionx* está definida recursivamente por lo que su codificación sigue exactamente el esquema indicado. Los primeros valores de esta función son: 0, 1, 2, 4, 9, 19, 41,...

Codificación

```

int funcionx( int n)
{
    if (n <= 2)
        return(n);
    else
        return(funcionx(n-3) + 2 * funcionx(n-2) + funcionx(n-1));
}

```

- 8.6.** Escriba una función recursiva que lea números enteros positivos ordenados crecientemente del teclado, elimine los repetidos y los escriba al revés. El fin de datos viene dado por el número especial 0.

Análisis del problema

La función recursiva que se escribe tiene un parámetro llamado *ant* que indica el último número leído. Inicialmente se llama a la función con el valor -1 que se sabe que no puede estar en la lista. Lo primero que se hace es leer un número *n* de la lista. Como el final de la lista viene dada por 0, si se lee el 0 entonces “*se rompe*” la recursividad y se da un salto de línea. En caso de que el número leído no sea 0 se tienen dos posibilidades: la primera es que *n* no coincida con el anterior dado en *ant*, en cuyo caso se llama a la recursividad con el valor de *ant*, dado por *n*, para posteriormente y a la vuelta de la recursividad escribir el dato *n*; la segunda es que *n* coincida con *ant*, en cuyo caso se llama a la recursividad con el nuevo valor de *ant*, dado por *n*, pero ahora a la vuelta de la recursividad no se escribe *n* pues está repetido.

Codificación

```

#include <stdio.h>
int elimina (int ant );

int main()
{
    elimina(-1);
    return 0;
}

int elimina( int ant)
{
    int n;

```

```

scanf("%d",&n);
if(n == 0)
    printf("\n");
else if ( !(n == ant))
{
    elimina ( n );
    printf("%d", n);
}
else
    elimina( n );
return( 0 );
}

```

8.7. El problema de las torres de Hanoi general tiene tres varillas o torres denominadas Origen, Destino y Auxiliar y un conjunto de $n > 0$ discos de diferentes tamaños. Cada disco tiene una perforación en el centro que le permite colocarse en cualquiera de las torres. En la varilla Origen se encuentran colocados inicialmente los n discos de tamaños diferentes ordenados de mayor a menor, como se muestra en el dibujo. Se trata de llevar los n discos de la varilla Origen a la varilla Destino utilizando las siguientes reglas:

1. Sólo se puede llevar un solo disco cada vez.
2. Un disco sólo puede colocarse encima de otro con diámetro ligeramente superior.
3. Si se necesita puede usarse la varilla Auxiliar.

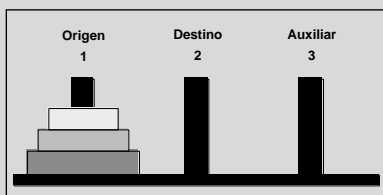
Análisis del problema

El problema de pasar 3 discos del pivote origen al destino se puede resolver en tres pasos:

Paso 1: pasar 2 discos de la varilla Origen a la varilla Auxiliar.

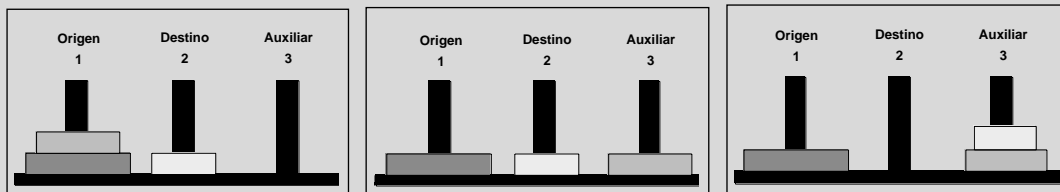
Paso 2: pasar un disco a la varilla Origen a la Destino.

Paso 3: pasar dos discos de la varilla Auxiliar a la Destino.



El paso 1, vulnera la primera regla, pero para solucionarlo, se puede recurrir a la misma mecánica: mover 1 disco de la varilla Origen a la Destino; mover el siguiente disco a la varilla Auxiliar; mover el disco de la varilla Destino a la Auxiliar:

Fases del Paso 1



El paso 2 se realiza de la manera óbvía, y el paso 3 es análogo al paso 1.

Algorítmicamente, el problema tiene una solución muy sencilla usando la recursividad y la técnica *divide y vence*. Para resolver el problema basta con observar que si sólo hay un disco $n=1$ (caso trivial), entonces se lleva directamente de la varilla Origen a la varilla Destino. Si hay que llevar $n > 1$ (caso general) discos de la varilla Origen a la varilla Destino, entonces:

Se llevan $n-1$ discos de la varilla Origen a la Auxiliar.

Se lleva un solo disco de la varilla Origen a la Destino.

Se traen los $n-1$ discos de la varilla Auxiliar a la Destino.

Codificación

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

void Hanoi(int n, int Origen, int Destino, int Auxiliar)
{
    if (n == 1)
        printf("llevo disco %3d de la varilla %3d a la varilla %3d\n", n, Origen, Destino);
    else
    {
        Hanoi(n - 1, Origen, Auxiliar, Destino);
        printf("llevo disco %3d de la varilla %3d a la varilla %3d\n", n, Origen, Destino);
        Hanoi(n - 1, Auxiliar, Destino, Origen);
    }
}

int main()

{
    char sig;
    Hanoi(3, 1, 2, 3);
    puts("\nPresione una tecla para continuar . . . ");
    scanf ("%c",&sig);
    return 0;
}
```

Ejecución

```
llevo disco 1 de la varilla 1 a la varilla
llevo disco 2 de la varilla 1 a la varilla
llevo disco 1 de la varilla 2 a la varilla
llevo disco 3 de la varilla 1 a la varilla
llevo disco 1 de la varilla 3 a la varilla
llevo disco 2 de la varilla 3 a la varilla
llevo disco 1 de la varilla 1 a la varilla
Presione una tecla para continuar . . .
```

8.8. Realizar una función recursiva que calcule el producto de números naturales.

Análisis del problema

El producto $x*y$ si x e y son números naturales se define recursivamente de la siguiente forma:

```
x * 0 = 0                                si y == 0
x * y = x * (y - 1) + x                  si y > 0
```


Codificación

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

int producto (int x, int y)
{
    if (y == 0)
        return( 0 );
    else
        return( producto(x, y -1 ) + x);
}

int main()
{
    int x, y;
    do
    {
        printf("dame x e y >= 0\n");
        scanf("%d %d", &x,&y);
    } while (x < 0 || y < 0);
    printf("el producto de %d por %d es:%d\n",x, y, producto(x,y));
    return 0;
}
```

- 8.9.** Codificar un programa que mediante la técnica de recursividad indirecta escriba el alfabeto en minúsculas.

Análisis del problema

La codificación usa dos funciones recursivas `f1()` que llama a la función `f2()` y `f2()` que llama a la función `f1()`. Ambas funciones reciben como parámetro un carácter `c`, y si ese carácter es menor que `'z'` entonces lo escriben; sitúan el carácter `c` en el siguiente carácter y llaman a la otra función. En caso de que el carácter `c` que reciben como parámetro sea el `'z'` lo escriben y no realizan ninguna llamada recursiva. El programa principal simplemente se encarga de llamar a una función (en este caso `f1()`) con `'a'`.

Codificación

```
#include <stdio.h>

void f1(char c);
void f2(char c);

int main()
{
    f1('a');
    printf("\n");
    return 0;
}

void f1(char c)
{
    if (c < 'z')
    {
```

```

        printf("%c ", c);
        c++;
        f2( c );
    }
    else
        printf("%c", c);
}

void f2(char c)
{
    if (c < 'z')
    {
        printf("%c ", c);
        c++;
        f1( c );
    }
    else
        printf("%c", c);
}

```

8.10. *Escriba una función recursiva que calcule la suma de los n primeros términos de la serie armónica.*

$$s = \sum_{i=1}^n \frac{1}{i}$$

Análisis del problema

La suma de la serie armónica realizada descendentemente puede definirse recursivamente de la siguiente forma:

$$\begin{array}{ll}
 S(1) = 1 & \text{si } i = 1 \\
 S(i) = 1 / i + S(i - 1) & \text{si } i > 1
 \end{array}$$

Si se realiza ascendentemente la definición recursiva es:

$$\begin{array}{ll}
 S1(i,n) = 1 / i & \text{si } i = n \\
 S1(i,n) = 1 / i + S1(i + 1, n + 1) & \text{si } i < n
 \end{array}$$

Se codifica el programa principal y las dos versiones. Hay que tener en cuenta que si i es entero entonces $1/i$ da como resultado el entero 0. Para evitarlo hay que obligar a que el operador $/$ realice la división real. Esto se codifica en el problema como $1/(\text{float})i$.

Codificación

```

#include <stdio.h>
float S(int i)
{
    if (i == 1)
        return 1;
    else
        return( 1 / (float)i + S(i - 1));
}

float S1(int i, int n)

```

```

{
    if (i == n)
        return 1/ (float)i;
    else
        return( 1 / (float)i + S1(i + 1, n));
}

int main()
{

    printf("\n %f, la otra %f \n", S(6), S1(1,6));
    return 0;
}

```

8.11. *Escriba una función iterativa y otra recursiva para calcular el valor aproximado del número e , sumando la serie:*

$$e = 1 + 1/1! + 1/2! + \dots + 1/n!$$

hasta que los términos adicionales a sumar sean menores que $1.0e^{-8}$.

Análisis del problema

La función `loge()`, calcula iterativamente la suma de la serie indicada de la siguiente forma: la variable `delta` contendrá en todo momento el valor del siguiente término a sumar. Es decir tomará los valores de 1, $1/1!$, $1/2!$, $1/3!$, ... $1/n!$. La variable `suma` contendrá las sumas parciales de la serie, por lo que se inicializa a cero, y en cada iteración se va sumando el valor de `delta`. La variable `n` contendrá los valores por los que hay que dividir `delta` en cada iteración para obtener el siguiente valor de `delta` conocido el valor anterior. La función `logeR()` codifica la serie recursivamente. Tiene como parámetro el valor de `n` y el valor `delta`. El valor de `n` debe ser incrementado en una unidad en cada llamada recursiva. El parámetro `delta` contiene en cada momento el valor del término a sumar de la serie. Si el término a sumar es mayor o igual que $1.0 \cdot e^{-8}$, se suma el término a la llamada recursiva de la función recalculando en la propia llamada el nuevo valor de `delta` (dividiéndolo entre `n+1`). Si el término a sumar `delta` es menor que $1.0 \cdot e^{-8}$ se retorna el valor de 0, ya que se ha terminado de sumar la serie.

Codificación

```

#include <stdio.h>

double loge(void)
{
    double suma, delta;
    int n;
    suma = 0;
    delta = 1.0;
    n = 0;
    do
    {
        suma += delta;
        n++;
        delta = delta/n;
    } while (delta >= 1.0e-8);
    return suma;
}

```

```
double logeR(int n, float delta)
{
    if (delta >= 1.0e-8)
        return ( delta + logeR(n + 1, delta / (n + 1)));
    else
        return 0;
}

int main()
{
    double aux, aux1;
    char sig;
    aux = logeR(0, 1.0);
    aux1 = loge();
    printf (" recursivo %f \n no recursivo %f \n", aux, aux1);
    puts("\nPresione una tecla para continuar . . . ");
    scanf ("%c",&sig);
    return 0;
}
```

Ejecución

```
recursivo      2.718282
no recursivo 2.718282
Presione una tecla para continuar
```

8.12. Escriba una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

$$\begin{aligned}
 A(m, n) &= n + 1 && \text{si } m = 0 \\
 A(m, n) &= A(m - 1, 1) && \text{si } n = 0 \\
 A(m, n) &= A(m - 1, A(m, n - 1)) && \text{si } m > 0, \text{ y } n > 0
 \end{aligned}$$
Análisis del problema

La función recursiva queda perfectamente definida en el propio problema por lo que su codificación en C se realiza exactamente igual que la propia definición. En la codificación se incluye un programa principal con sucesivas llamadas a la función `Akerman`.

Codificación

```
#include <stdio.h>
double akerman (int m, int n)
{
    if (m == 0)
        return n + 1;
    else
        if (n == 0)
            return (akerman(m - 1, 1));
        else
            return (akerman(m - 1, akerman(m, n - 1)));
}

int main()
{
    long a;
```

```

int i, j;
char sig;
for (i = 1; i <= 3; i++)
{
    printf(" fila %d :", i);
    for (j = 1; j <= 8; j++)
    {
        a = akerman (i, j);
        printf ( " %d ", a);
    }
    printf ("\n");
}
puts("\nPresione una tecla para continuar . . . ");
scanf ("%c",&sig);
return 0;
}

```

Ejecución

```

C:\> Seleccionar C:\Dev-Cpp\Project1.exe
fila 1 : 3 4 5 6 7 8 9 10
fila 2 : 5 7 9 11 13 15 17 19
fila 3 : 13 29 61 125 253 509 1021 2045
Presione una tecla para continuar . . . _

```

8.13. Escriba una función recursiva que calcule el máximo de un vector de enteros.

Análisis del problema

Suponiendo que el vector tenga al menos dos elementos, el elemento mayor se puede calcular recursivamente, usando la función `max()` que devuelva el mayor de dos enteros que reciba como parámetros. Posteriormente se puede definir la función `maxarray()` que calcula el máximo del array recursivamente con el siguiente planteamiento:

Caso trivial : $n == 1$, solución trivial `max(a[0],a[1])`
Caso general: $n > 1$, `maxarray = max(maxarray(a,n-1),a[n])`

Codificación

```

int max(int x, int y)
{
    if (x < y)
        return y;
    else
        return x;
}

int maxarray( int a[], int n)
{
    if (n ==1)
        return (max(a[0], a[1]));
    else
        return( max(maxarray(a, n - 1), a[n]));
}

```

8.14. Escriba una función recursiva, que calcule el producto de los elementos de un vector v que sean mayores que un valor b .

Análisis del problema

Se programa la función producto de la siguiente forma: la función tiene como parámetros el vector v , el valor de b , y un parámetro n que indica que falta por resolver el problema para los datos almacenados en el vector desde la posición 0 hasta la n . De esta forma si n vale 0 el problema tiene una solución trivial: si $v[0] \leq b$ devuelve 1, y si $v[0] > b$ devuelve $v[0]$. Si n es mayor que 0, entonces, debe calcularse recursivamente el producto desde la posición 0 hasta la $n-1$ y después multiplicarse por $v[n]$ en el caso de que $v[n]$ sea mayor que b .

Codificación

```
float producto(float v[], float b, int n)
{
    if (n == 0)
        if (v[0] <= b)
            return 1;
        else
            return v[0];
    else
        if (v[n] <= b)
            return producto(v, b, n - 1);
        else
            return v[n]* producto(v, b, n - 1);
}
```

PROBLEMAS PROPUESTOS

- 8.1** Diseñar una función recursiva de prototipo `int vocales(const char * cd)` para calcular el número de vocales de una cadena.
- 8.2** Diseñar una función recursiva que calcule la suma de los elementos de un vector recursivamente.
- 8.3** Aplique el esquema de los algoritmos *divide y vence* para que dados las coordenadas (x,y) de dos puntos en el plano, que representan los extremos de un segmento, se dibuje el segmento.
- 8.4** Diseñar una función recursiva que sume los n primeros números naturales. Compuebe el resultado en otra función sabiendo que la suma de los primeros n números enteros responde a la fórmula:

$$1 + 2 + 3 + \dots + n = n(n + 1) / 2$$
- 8.5** Un palíndromo es una palabra que se escribe exactamente igual leída en un sentido o en otro. Palabras tales como *level*, *deed*, *ala*, etc., son ejemplos de palíndromos. Aplicar el esquema de los algoritmos *divide y vence* para escribir una función recursiva que devuelva 1, si una palabra pasada como argumento es un palíndromo y en caso contrario devuelva 0. Escribir un programa en el que se lea una cadena hasta que esta sea palíndromo.
- 8.6** Diseñar un programa que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe hallar la suma de los dígitos de cada entero y encontrar cual es el entero cuya suma de dígitos es mayor. La suma de dígitos calcúlese con una función recursiva.

- 8.7** Leer un número entero positivo $n < 10$. Calcular el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^2 de la forma $x^{**}i$.

Sugerencia:

$$(x + 1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + \dots + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de n

La relación de recurrencia de los coeficientes binomiales es:

$$C(n, 0) = 1$$

$$C(n, n) = 1$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

- 8.8** Sea A una matriz cuadrada de $n \times n$ elementos, el determinante de A se puede definir de manera recursiva:

- a. Si $n=1$ entonces $\text{Deter}(A) = a_{1,1}$
 b. Para $n > 1$, el determinante es la suma alternada de productos de los elementos de una fila o columna elegida al azar por sus menores complementarios. A su vez, los menores complementarios son los determinantes de orden $n-1$ obtenidos al suprimir la fila y columna en que se encuentra el elemento.

La expresión matemática es:

$$\text{Det}(A) = \sum_{i=1}^n (-1)^{i+j} * A[i, j] * \text{Det}(\text{Menor}(A[i, j]))$$

para cualquier columna j

- 8.9** Diseñar un programa que transforme números enteros en base 10 a otro en base b . Siendo la base b de 2 a 9. La transformación se ha de realizar siguiendo una estrategia recursiva.

Arrays¹

(listas y tablas)

En capítulos anteriores se han descrito las características de los tipos de datos básicos o simples (carácter, entero y coma flotante). Así mismo, se ha aprendido a definir y utilizar constantes simbólicas utilizando `const`, `#define` y el tipo `enum`. En este capítulo se continúa con el examen de los restantes tipos de datos de C, examinando especialmente el tipo **array** o **arreglo** (lista o tabla) y aprenderá el concepto y tratamiento de los *arrays*. Un *array* almacena muchos elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. El array es muy importante por diversas razones. Una muy importante es almacenar secuencias o *cadenas* de texto. Hasta el momento C, proporciona datos de un sólo carácter; utilizando el tipo array, se puede crear una variable que contenga un grupo de caracteres.

9.1 Arrays

Un **array** o **arreglo** (lista o tabla) es una secuencia de datos del mismo tipo. Los datos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3... (*valores índice o subíndice* del array). En general, el elemento *i-ésimo* está en la posición *i-1*. De modo que si el array *a* tiene *n* elementos, sus nombres son `a[0]`, `a[1]`, ..., `a[n-1]`. El tipo de elementos almacenados en el array puede ser cualquier tipo de dato de C, incluyendo estructuras definidas por el usuario, como se describirá más tarde.

Un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el *tamaño o longitud* del array. Para indicar al compilador el *tamaño o longitud* del array se debe hacer seguir al nombre, el tamaño encerrado entre corchetes. La *sintaxis* para declarar un array de una dimensión determinada es:

```
tipo nombreArray[numeroDeElementos];
```

Observación: C no comprueba que los índices del array están dentro del rango definido.

El índice de un *array* se denomina, con frecuencia, *subíndice del array*. El método de numeración del elemento *i-ésimo* con el índice o subíndice *i-1* se denomina *indexación basada en cero*. Su uso tiene el efecto de que el índice de un elemento del array es siempre el mismo que el número de “pasos” desde el elemento inicial `a[0]` a ese elemento. Por ejemplo, `a[3]` está a 3 pasos o posiciones del elemento `a[0]`.

¹ En Latinoamérica es muy frecuente el uso del término **arreglo** como traducción del término *array*.

EJEMPLO 9.1 *Posiciones válidas de un array.*

En el array `int a[10]` los índices válidos son `a[0]`, `a[1]`, ..., `a[9]`. Pero si se considera `a[15]` no se proporciona un mensaje de error y el resultado puede ser impredecible.

Los elementos de los *arrays* se almacenan en bloques contiguos de memoria. En los programas se pueden referenciar elementos del *array* utilizando fórmulas o expresiones enteras para los subíndices. Los *arrays* de caracteres funcionan de igual forma que los *arrays* numéricos, partiendo de la base de que cada carácter ocupa normalmente un byte. Hay que tener en cuenta, que en las cadenas de caracteres el sistema siempre inserta un último carácter (el carácter *nulo*) para indicar fin de cadena.

El operador `sizeof` devuelve el número de bytes necesarios para contener su argumento. Si se usa `sizeof` para solicitar el tamaño de un array, esta función devuelve el número de bytes reservados para el array completo. Conociendo el tipo de dato almacenado en el array y su tamaño, se tiene la longitud del array, mediante el cociente `sizeof(a)/tamaño(dato)`. Al contrario que otros lenguajes de programación, C no verifica el valor del índice de la variable que representa al array esté dentro del rango de variación válido, por lo que si se sobrepasa el valor máximo declarado, los resultados pueden ser impredecibles.

EJEMPLO 9.2 *Posiciones ocupadas por los elementos de un array.*

Si `a` es un array de número reales y cada número real ocupa 4 bytes, entonces si el elemento `a[0]` ocupa la dirección `x`, el elemento `a[i]` ocupa la dirección de memoria `x + (i-1)*4`.

9.2 Inicialización de un array

Se deben asignar valores a los elementos del array antes de utilizarlos, tal como se asignan valores a variables. Para asignar valores a cada elemento del array de enteros `p`, se puede escribir:

```
p[0] = 10; p[1] = 20; p[2] = 30; p[3] = 40
```

La primera sentencia fija `p[0]` al valor 10, `p[1]` al valor 20, etc. Sin embargo, este método no es práctico cuando el array contiene muchos elementos. El método utilizado, normalmente, es inicializar el array completo en una sola sentencia. Cuando se inicializa un array, el tamaño del array se puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves.

EJEMPLO 9.3 *Declaraciones e inicialización de arrays.*

```
int num[6] = {10, 20, 30, 40, 50, 60};
int x[ ] = {1,2,3}                      /*Declara e inicializa un array de 3 elementos */
char ch[] = {'L','u','c','a','s'};      /*Declara un array de 5 datos */
```

9.3 Arrays de caracteres y cadenas de texto

Una cadena de texto es un conjunto de caracteres. C soporta cadenas de texto utilizando un array de caracteres que contenga una secuencia de caracteres. Sin embargo, no se puede asignar una cadena a un array del siguiente modo: `Cadena = "ABC-DEF"`. La función de la biblioteca estándar `strcpy()` (“copiar cadenas”) permite copiar una constante de cadena en una cadena. Para copiar el nombre “Ejemplo” en el array `x`, se puede escribir:

```
strcpy(x, "Ejemplo");                    /*Copia Ejemplo en x */
```

EJEMPLO 9.4 *Inicialización de una cadena de caracteres.*

Una *cadena* de caracteres es un array de caracteres que contiene al final el carácter *carácter nulo* (`\0`). Mediante la sentencia declarativa `char Cadena[] = "abcdefg"` el compilador inserta automáticamente un carácter nulo al final de la cadena, de modo que la secuencia real sería:

```
char Cadena[7] = "ABCDEF";
```

Cadena	A	B	C	D	E	F	\0
--------	---	---	---	---	---	---	----

9.4 Arrays multidimensionales

Los *arrays multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arrays más usuales son los de dos dimensiones, conocidos también por el nombre de *tablas* o *matrices*. Sin embargo, es posible crear arrays de tantas dimensiones como requieran sus aplicaciones, esto es, tres, cuatro o más dimensiones. Los elementos de los arrays se almacenan en memoria por filas. Hay que tener en cuenta que el subíndice más próximo al nombre del array es la *fila* y el otro subíndice, la *columna*. La sintaxis para la declaración de un *array* de dos dimensiones es:

```
<TipoElemento><nombrearray>[<NúmeroDeFilas>][<NúmeroDeColumnas>]
```

La sintaxis para la declaración de un array de tres dimensiones es:

```
<tipodedatoElemento><nombrearray> [ <Cota1> ] [ <Cota2> ] [ <Cota3> ]
```

EJEMPLO 9.5 Declaración y almacenamiento de array

Dada la declaración: `int a[5][6];`

El orden de almacenamiento es el siguiente:

```
a[0][0], a[0][1], a[0][2], ..., a[0][5], a[1][0], a[1][1], ..., a[1][5], ..., a[4][0],
a[4][1], ..., a[4][5].
```

Los arrays multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves, como en el ejemplo siguiente:

```
int ejemplo[2][3] = {1,2,3,4,5,6};
```

El formato general para asignación directa de valores a los elementos es para la *inserción de elementos*:

```
<nombre array>[indice fila][indice columna] = valor elemento.
```

Para la extracción de elementos:

```
<variable> = <nombre array> [indice fila][indice columna]
```

Las funciones de entrada o salida se aplican de igual forma a los elementos de un array unidimensional. Se puede acceder a los elementos de arrays bidimensionales mediante bucles anidados. Su sintaxis general es:

```
int IndiceFila, IndiceCol;

for (IndiceFila = 0; IndiceFila < NumFilas; ++IndiceFila)
    for (IndiceCol = 0; IndiceCol < NumCol; ++IndiceCol)
        Procesar elemento[IndiceFila][IndiceCol];
```

9.5 Utilización de arrays como parámetros

En C *todos los arrays se pasan por referencia* (dirección) a las funciones. C trata automáticamente la llamada a una función como si hubiera situado el operador de dirección & delante del nombre del array que realiza la llamada. La declaración en la función de un parámetro tipo array se realiza con uno de los formatos siguientes:

1. `<tipo de datoElemento> <nombre array> [<Cota1>]`
2. `<tipo de datoElemento> <nombre array> []`

En arrays bidimensionales se realiza siempre indicando el número de columnas de la siguiente forma:

`<tipo de datoElemento> <nombre array> [<Cota1>][cota2].` O bien,
`<tipo de datoElemento> <nombre array> [][]cota2]`

Cuando se pasa un array a una función, se pasa realmente *sólo* la dirección de la celda de memoria donde comienza el array. Este valor se representa por el *nombre del array*. La función puede cambiar el contenido del *array* accediendo directamente a las celdas de memoria en donde se almacenan los elementos del *array*.

EJEMPLO 9.6 Declaración de funciones con parámetros array:

```
float suma(float a[5]) ,o float suma(float a[]), o bien float suma(float *a).
void calcula (float x[][5])
```

En el caso de la función `suma()`, si se tiene declarado `int b[5]`, una posible llamada a la función sería `suma(b)`.

Cuando se utiliza una variable array como argumento, la función receptora puede no conocer cuántos elementos existen en el array. Aunque la variable array apunta al comienzo del mismo, no proporciona ninguna indicación de donde termina el array. Se pueden utilizar dos métodos alternativos para permitir que una función conozca el número de argumentos asociados con un array que se pasa como argumento de una función:

- a) Situar un valor de señal al final del array, que indique a la función que se ha de detener el proceso en ese momento.
- b) Pasar un segundo argumento que indica el número de elementos del array.

La técnica de *paso de arrays como parámetros* se utiliza, también, para pasar cadenas de caracteres a funciones. Las cadenas terminan en nulo ('`\0`' o *nulo*, es el carácter cero del código de caracteres ASCII) por lo que el primer método dado anteriormente sirve para controlar el tamaño de un array.

PROBLEMAS PROPUESTOS

9.1. ¿Cuál es la salida del siguiente programa?.

```
#include <stdio.h>
void main(void)
{
    int i;  int Primero[21];
    for (i = 1; i <= 6; i++)
        scanf("%d",&Primero[i]);
    for(i = 3; i > 0; i - -)
        printf("%4d",Primero[2*i]);
    return;
}
```

Solución

Si la entrada de datos es por ejemplo: 3 7 4 -1 0 6. Estos se colocan en las posiciones del array números 1, 2, 3, 4, 5, 6 y por lo tanto la salida será 6 -1 7 ya que el bucle es descendente y se escriben las posiciones del array números 6 4 y 2.

9.2. ¿Cuál es la salida del siguiente programa?.

```
#include <stdio.h>
void main(void)
{
    int i,j,k; int Segundo[21];
    scanf("%d",&k);
    for(i =3; i <= k;)
        scanf("%d",&Segundo[i++]);
    j = 4;
    printf("%d %5d\n",Segundo[k],Segundo[j+1]);
}
```

Solución

Si la entrada de datos es por ejemplo 6 3 0 1 9. Estos números se almacenan en k el número 6 y el resto en las posiciones del array 3, 4, 5, 6. Por lo tanto la salida de resultados serán los números 9 1.

9.3. ¿Cuál es la salida del siguiente programa?.

```
#include <stdio.h>
void main(void)
{
    int i,j,k;int Primero[21];
    for(i = 0; i < 10;i++)
        Primero[i] = i + 3;
    scanf("%d %d",&j,&k);
    for(i = j; i <= k;)
        printf("%d\n",Primero[i++]);
}
```

Solución

Si la entrada de datos es por ejemplo 7 2, el programa no tendrá ninguna salida ya que el segundo bucle es ascendente y el límite inferior es mayor que el superior. Si la entrada de datos es por ejemplo 2 7, el programa escribirá las posiciones del array 2, 3, 4, 5, 6, 7 que se han inicializado, previamente, con los valores 5, 6, 7, 8, 9, 10.

9.4. ¿Cuál es la salida del siguiente programa?.

```
void main(void)
{
    int i, j, k;
    int Primero[21], Segundo[21];
    for(i = 0; i < 12; i++)
        scanf("%d",&Primero[i]);
    for(j = 0; j < 6; j++)
        Segundo[j] = Primero[2*j] + j;
```

```

    for(k = 3; k < 7; k++)
        printf(“%d %d \n”,Primero[k+1],Segundo [k-1]);
}

```

Solución

Si la entrada de datos es 2 7 3 4 9 -4 6 -5 0 5 -8 10, y teniendo en cuenta que el programa lee primeramente 12 números almacenándolos consecutivamente en el array `Primero`, para posteriormente almacenar 6 valores en el array `Segundo` en las posiciones 0,1,2,3,4,5 que en este caso serán respectivamente 2, 3+1, 9+2, 6+3, 0+4, -8+ 5, la salida de resultados será:

```

9      11
-4     9
6      4
-5     3

```

- 9.5. Escriba un programa que lea por filas una matriz de orden 3 por 4 (tres filas y cuatro columnas) y la escriba por columnas.

Análisis del problema

Se definen las constantes *filas* y *columnas*. En primer lugar, mediante dos bucles anidados, se lee la matriz por filas y posteriormente se escribe por columnas mediante otros dos bucles anidados.

Codificación

```

#include <stdio.h>
#define filas 3
#define columnas 4
void main(void)
{
    int i,j;
    int M[filas][columnas];

    // lectura por filas
    for(i = 0; i < filas; i++)
        for (j = 0;j < columnas; j++)
            scanf(“%d”,M[i][j]);

    // escritura por columnas
    for(j = 0; j < columnas;j++)
    {
        for (i = 0; i< filas;i++)
            printf(“%5d”,M[i][j]);
        printf(“\n”);
    }
}

```

- 9.6. Escriba un programa que lea una matriz cuadrada la presente en pantalla, y presente la suma de todos los números que no están en la diagonal principal.

Análisis del problema

El problema se resuelve en un solo programa principal. Dos bucles `for` anidados leen la matriz, otros dos bucles anidados la escriben, y otros dos bucles anidados se encargan de realizar la suma de los elementos que no están en la diagonal principal, que son aquellos que cumplen la condición $i < > j$.

Codificación

```
#include <stdio.h>
#define filas 5
void main(void)
{
    int i,j, suma; int M[filas][filas];
    // lectura por filas
    for(i = 0; i < filas; i++)
        for (j = 0; j < filas; j++)
            scanf("%d", M[i][j]);

    // escritura por filas
    for(i = 0; i < filas; i++)
    {
        for (j = 0; j < filas; j++)
            printf("%5d", M[i][j]);
        printf("\n");
    }

    suma=0; // realización de la suma
    for(i = 0; i < filas; i++)
        for (j = 0; j < filas; j++)
            if(!(i == j))
                suma += M[i][j];
    printf(" suma &d \n", suma);
}
```

- 9.7.** Escriba un programa C que intercambie el valor de la fila *i* con la fila *j*, de una matriz cuadrada de orden 7.

Análisis del problema

El programa que se presenta, sólo muestra el segmento de código que se encarga de intercambiar la fila *i* con la fila *j*.

Codificación

```
#include <stdio.h>
#define filas 5
void main(void)
{
    int i,j, k, aux;
    int M[filas][filas];

    // intercambio de fila i con fila j
    for (k = 0; k < filas; k++)
    {
        aux = M[i][k];
        M[i][k] = M[j][k];
        M[j][k] = aux;
    }
}
```

- 9.8.** Escriba un programa en C que declare un vector de longitud máxima *max*, y llame a funciones que se encarguen de leer el vector, escribirlo, sumar dos vectores, restar dos vectores, hacer cero a un vector, rellenar el vector de unos.

Análisis del problema

La solución se plantea de la siguiente forma. Las funciones que se encargan de resolver cada uno de los apartados tienen un parámetro entero n que indica la dimensión del vector, y un vector de longitud máxima $\text{max} = 11$. El programa principal se encarga de llamar a los distintos módulos. Las funciones que resuelven el problema son:

rellena. Se encarga de leer de la entrada, las n componentes del vector.

escribe. Se encarga de presentar los distintos valores de las componentes del vector.

suma. Se encarga de recibir como parámetros dos vectores a y b , y dar como resultado, el vector c .

resta. Se encarga de recibir como parámetros dos vectores a y b y dar como resultado el vector diferencia c .

cero. Inicializa a cero todas las componentes del vector.

Identidad. Pone todas las componentes del vector a uno.

Codificación (Consultar en la página web del libro)

- 9.9.** *Escriba un programa que lea un total de 10 números enteros, calcule la suma de todos ellos así como la media presentando los resultados.*

Análisis del problema

Una constante `NUM` nos declara el valor 20. Dos bucles `for` se encargan de leer los datos y de calcular los resultados.

Codificación

```
#include <stdio.h>
#define NUM 10
int main()
{
    int numeros[NUM];
    int i, total=0;
    for (i = 0; i < NUM; i++)
    {
        printf(" Introduzca el número: que ocupa poscion %d :",i);
        scanf("%d",&numeros[i]);
    }
    printf("\n Lista de números leídos: ");
    for (i = 0; i < NUM; i++)
    {
        printf("%d ",numeros[i]);
        total += numeros[i];
    }
    printf("\nLa suma total de todos los números es %d",total) ;
    printf("\n La media es %f",float(total)/NUM) ;
    return 0;
}
```

- 9.10.** *Escriba funciones C que usando las definiciones del ejercicio 9.8 se encargue de asignar a un vector otro vector, que escriba, el mayor y menor elemento de un vector así como sus posiciones, que decida si un vector es simétrico, antisimétrico, y mayoritario.*

Análisis del problema

La solución se plantea mediante la programación de las siguientes funciones:

- *asigna*. Recibe como parámetro un vector *a* y devuelve un vector *b* que es una copia del vector *a*.
- *mayormenor*. Se encarga de escribir el mayor y menor elemento del vector así como las posiciones en que se encuentra.
- *simétrico*. Decide si un vector es *simétrico*. (Un vector de *n* datos se dice que es simétrico si el contenido de la posición *i_ésima* coincide con el que ocupa la posición *n-i_ésima*, siempre que el número de elementos que almacene el vector sea *n*).
- *antisimétrico*. Decide si un vector es *antisimétrico*. (Un vector de *n* datos se dice que es antisimétrico si el contenido de la posición *i_ésima* coincide con el que ocupa la posición *n-i_ésima* cambiada de signo, siempre que el número de elementos que almacene el vector sea *n*).
- *Mayoritario*. Decide si un vector es *mayoritario*. (Un vector de *n* datos se dice que es mayoritario, si existe un elemento almacenado en el vector que se repite más de *n/2* veces).

Codificación (Consultar en la página web del libro)

- 9.11.** Escriba funciones que calculen el producto escalar de dos vectores, la norma de un vector y el coseno del ángulo que forman.

Análisis del problema

El producto escalar de dos vectores de *n* componentes se define de la siguiente forma:

$$pe(n,a,b) = \sum_{i=1}^n a(i) \cdot b(i)$$

la norma de un vector de *n* componentes se define de la siguiente forma:

$$norma(n,a) = \sqrt{\sum_{i=1}^n a(i) \cdot a(i)} = \sqrt{pe(n,a,a)}$$

El coseno del ángulo que forman dos vectores de *n* componentes se define de la siguiente forma:

$$coseno(n,a,b) = \frac{\sum_{i=1}^n a(i) \cdot b(i)}{\sqrt{\sum_{i=1}^n a(i) \cdot a(i)} \cdot \sqrt{\sum_{i=1}^n b(i) \cdot b(i)}} = \frac{pe(n,a,b)}{norma(n,a) \cdot norma(n,b)}$$

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define max 11
float productoescalar(int n, float a[max], float b[max]);
float norma(int n, float a[max]);
float coseno(int n, float a[max], float b[max]);

float productoescalar(int n, float a[max], float b[max])
{
    int i; float acu;
    acu = 0;
    for (i = 0; i < n; i++)
        acu += a[i] * b[i];
    return acu;
}
```



```

}

float norma(int n, float a[max])
{
    float aux;
    aux = productoescalar(n,a,a);
    aux = sqrt(aux);
    return aux;
}

float coseno(int n, float a[max], float b[max])
{
    float aux;
    aux = productoescalar(n,a,b) / (norma(n,a) * norma(n,b));
    return aux;
}

```

- 9.12.** *Escriba funciones para calcular la media m , desviación media dm , desviación típica dt , media cuadrática mc y media armónica ma , de un vector de hasta n elementos siendo n un dato.*

Análisis del problema

La media, desviación media, desviación típica, media cuadrática, y media armónica se definen de la siguiente forma:

$$m = \left(\sum_{i=1}^n a(i) \right) / n$$

$$dm = \frac{\sum_{i=1}^n \text{abs}(a(i) - m)}{n}$$

$$mc = \sqrt{\frac{\sum_{i=1}^n a(i)^2}{n}}$$

$$ma = \frac{n}{\sum_{i=1}^n \frac{1}{a(i)}}$$

$$dt = \sqrt{\frac{\sum_{i=1}^n (a(i) - m)^2}{n}}$$

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define max 11
float m(int n, float a[max]);
float dm(int n, float a[max]);

```

```
float mc (int n, float a[max]);
float ma(int n, float a[max]);
float dt (int n, float a[max]);

float m(int n, float a[max])
{
    int i;
    float aux;
    aux = 0;
    for(i = 0; i < n; i++)
        aux += a[i];
    return aux / n;
}

float dm(int n, float a[max])
{
    int i;
    float aux, media;
    aux = 0;
    media = m(n,a);
    for(i = 0; i < n; i++)
        aux += abs(a[i]- media);
    return aux / n;
}

float mc (int n, float a[max])
{
    int i;
    float aux;
    aux = 0;
    for(i = 0; i < n; i++)
        aux +=a[i] * a[i];
    return sqrt(aux / n);
}

float ma(int n, float a[max])
{
    int i;
    float aux;
    aux=0;
    for(i = 0; i < n; i++)
        aux += 1/a[i];
    return n / aux;
}

float dt (int n, float a[max])
{
    int i;
    float aux = 0,media;
    media = m(n,a);
    for(i = 0; i < n; i++)
        aux+=(a[i] - media) * (a[i] - media);
    return sqrt(aux / n);
}
```

- 9.13.** Escriba un programa que lea un número natural impar n menor o igual que 11, y calcule un cuadrado mágico de orden n . Un cuadrado de orden $n \times n$ se dice que es mágico si contiene los valores 1, 2, 3, ..., $n \times n$, y cumple la condición de que la suma de los valores almacenados en cada fila y columna coincide.

Análisis del problema

En un array bidimensional se almacenará el cuadrado mágico. El problema se resuelve usando las siguientes funciones:

- Una función *sig* que tiene como parámetro dos números enteros i y n , de tal manera que i es mayor o igual que cero y menor o igual que $n - 1$. La función devuelve el siguiente valor de i que es $i + 1$. En el caso de que al sumarle a i el valor 1, i tome el valor n se le asigna el valor de 0. Sirve para ir recorriendo los índices de las filas de la matriz que almacenará el cuadrado mágico de orden n .
- Una función *ant* que tiene como parámetro dos números enteros i y n , de tal manera que i es mayor o igual que cero y menor o igual que $n - 1$. La función devuelve el anterior valor de i que es $i - 1$. En el caso de que al restarle a i el valor 1, i tome el valor -1 se le asigna el valor de $n - 1$. Sirve para ir recorriendo los índices de las columnas de la matriz que almacenará el cuadrado mágico de orden n .
- Una función *comprueba* que escribe la matriz que almacena el cuadrado mágico y además escribe la suma de los valores de cada una de las filas y de cada una de las columnas, visualizando los resultados, para poder ser comprobado por el usuario.
- Una función *cuadrado* que calcula el cuadrado mágico mediante el siguiente conocido algoritmo:
 - Se pone toda la matriz a ceros, para indicar que las casillas están libres.
 - Se inicializa la fila $i = 1$ la columna $j = n / 2$.
 - Mediante un bucle que comienza por el valor 1, se van colocando los valores en orden creciente hasta el $n \times n$ de la siguiente forma.
 - Si la posición fila i columna j de la matriz está libre se almacena, y se recalcula la fila con la función *ant* y la columna con la función *sig*.
 - Si la posición fila i columna j de la matriz está ocupada se recalcula i aplicándole dos veces la función *sig* y se recalcula j aplicándole una vez la función *ant*. Se almacena el valor en la posición fila i columna j (siempre está libre), para posteriormente recalcular la fila con la función *ant* y la columna con la función *sig*.

Codificación

```
#include <stdio.h>
#define max 11
int sig(int i, int n);
int ant(int i, int n);
void cuadrado(int n, int a[][max]);
void comprueba(int n, int a[][max]);

int main()
{
    int n, a[max][max];
    do
    {
        printf("introduzca valor de n <= %d e impar \n", max);
        scanf("%d", &n);
        // bucle que controla que n está en el rango indicado y que
        // n es impar
    } while ((n <= 0) || (n > max) || (n % 2 == 0));
    cuadrado(n, a);
    comprueba(n, a);
    return 0;
}
```

```
int sig( int i, int n)
// calcula el siguiente de i en circulo. Los valores son: 0,1...n-1
{
    i++;
    if(i > n - 1)
        i = 0;
    return i;
}

int ant( int i, int n)
//calcula el anterior de i en circulo. Los valores son 0,1...n-1
{
    i- -;
    if(i < 0)
        i = n - 1;
    return i;
}

void cuadrado(int n, int a[][max])
// encuentra el cuadrado mágico de orden n mediante el
// algoritmo voraz
{
    int i,j,k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            a[i][j] = 0;

    j = n/2;
    i = 1;
    for (k = 1; k <= n * n; k++)
        if (a[i][j] == 0)
        {
            a[i][j] = k;
            i = ant(i,n);
            j = sig(j,n);
        }
        else
        {
            i = sig(i,n);
            j = ant(j,n);
            i = sig(i,n);
            a[i][j] = k;
            i = ant(i,n);
            j = sig(j,n);
        }
    }

void comprueba(int n, int a[][max])
// escribe el cuadrado magico asi como la suma de los
// elementos de cada fila, y cada columna
{
    int i,j, acu;
    printf(" cuadrado mágico de orden %d\n", n );
```

```

printf(" ultima columna = suma de elementos de fila\n");
printf(" ultima fila = suma de elementos de columna\n");
for (i = 0; i < n; i++)
{
    acu = 0;
    for (j = 0; j < n; j++)
    {
        printf("%4d", a[i][j]);
        acu += a[i][j];
    }
    printf("%6d \n", acu);
}
for (j = 0; j < n; j++)
{
    acu = 0;
    for (i = 0; i < n; i++)
        acu += a[i][j];
    printf("%4d", acu);
}
}

```

9.14. Implementar un programa que permita visualizar el triángulo de Pascal

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1

```

En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando primeramente un array bidimensional y posteriormente uno de una sola dimensión.

Análisis del problema

Se declaran dos array uno bidimensional *a* y otro unidimensional. La solución se estructura de la siguiente forma:

- Un módulo *pascalbi* tiene como parámetros el array bidimensional y una variable entera y calcula el triángulo de Pascal de orden *n*.
- Un módulo *escribibi* tiene como parámetros el array bidimensional y una variable entera y escribe el triángulo de pascal de orden *n*.
- Un módulo *pascaluni* tiene como parámetros el array unidimensional y una variable entera, calcula y escribe el triángulo de Pascal de orden *n*. En este módulo el bucle *j* se hace descendente para no “pisar” los valores anteriormente calculados.

Codificación

```

#include <stdio.h>
#define max 11
void pascalbi(intn, int a[][max]);
void escribibi(intn, int a[][max]);
void pascaluni(int n, intaa[max]);

int main()
{

```

```

int n, a[max][max], aa[max];
do
{
    printf("introduzca valor de n <= %d \n", max);
    scanf(" %d", &n);
} while ((n <= 0) || (n > max) );
pascalbi(n,a);
escribibi(n,a);
pascaluni(n,aa);
return 0;
}

void pascalbi(int n, int a[][max])
                                                    calcula el triangulo de Pascal
{
    int i,j;
    for (i=0; i < n; i++)
        for (j = 0; j <= i; j++)
            if(j == 0 || j == i )
                a[i][j] = 1;
            else
                a[i][j]= a[i - 1][j - 1] + a[i - 1][j];
}

void escribibi(int n, int a[][max])
                                                    // escribe el triangulo de Pascal
{
    int i,j;
    printf(" Triangulo de Pascal de orden %d\n", n );
    for (i = 0; i < n; i++)
    {
        for (j = 0; j <= i; j++)
            printf("%4d", a[i][j]);
        printf(" \n");
    }
}

void pascaluni(int n, int aa[max])
{
    int i,j;
    printf(" Triangulo de Pascal de orden %d\n", n );
    for (i = 0; i < n; i++)
    {
        for (j = i; j >= 0; j--)
            if (i == j || j == 0)
                aa[j] = 1;
            else
                aa[j] = aa[j] + aa[j - 1];
        for (j = 0; j <= i; j++)
            printf("%4d", aa[j]);
        printf("\n");
    }
}

```

9.15. Escriba tres funciones en C que calculen la suma de dos matrices, la resta de dos matrices y el producto de dos matrices.

Análisis del problema

Las tres funciones tienen como parámetro una variable entera y las matrices a, b, c.

En la matriz c se devuelve el resultado de la operación:

Suma $c(i, j) = a(i, j) + b(i, j)$

Resta $c(i, j) = a(i, j) - b(i, j)$

Producto $c(i, j) = \sum_k a(i, k) \cdot b(k, j)$

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#define max 11
void suma(int n, float a[][max], float b[][max], float c[][max]);
void resta(int n, float a[][max], float b[][max], float c[][max]);
void produc(int n, float a[][max], float b[][max], float c[][max]);
void resta(int n, float a[][max], float b[][max], float c[][max])
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            c[i][j] = a[i][j] - b[i][j];
}

void suma(int n, float a[][max], float b[][max], float c[][max])
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
}

void produc(int n, float a[][max], float b[][max], float c[][max])
{
    int i, j, k;
    float acu;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            acu = 0;
            for (k = 0; k < n; k++)
                acu += a[i][k] * b[k][j];
            c[i][j] = acu;
        }
}
```

9.16. Escriba funciones en C que calculen la matriz cero la matriz identidad, asigne una matriz a otra, y usando la función producto definida en el ejercicio anterior calcule la potencia de una matriz.

Análisis del problema

La matriz cero y la matriz identidad se calculan mediante sendas funciones que reciben como parámetros una variable entera n y devuelve en la matriz a la solución. La asignación de matrices se resuelve mediante una función que recibe como parámetro una matriz y devuelve la matriz copia como resultado de la asignación. La matriz potencia se calcula mediante una función que recibe como parámetros una variable entera n , otra variable entera m que indica el exponente al que hay que elevar la primera matriz que se recibe como parámetro y devuelve en otra matriz el resultado de calcular la potencia m -ésima de la matriz.

Cero $c(i, j) = 0$
 Identidad $c(i, j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$
 Potencia $b = a^m$
 Asigna $b(i, j) = a(i, j)$

Codificación (Consultar la página web del libro)

- 9.17.** Escriba funciones que calculen la traspuesta de una matriz cuadrada sobre sí misma, y decidan si una matriz es simétrica o antisimétrica.

Análisis del problema

La traspuesta de una matriz a viene dada por la matriz que se obtiene cambiando la fila i por la columna i . Es decir b es la matriz traspuesta de a si se tiene que $b(i, j) = a(j, i)$. Una matriz es simétrica si $a(i, j) = a(j, i)$. Una matriz es antisimétrica si $a(i, j) = -a(j, i)$. La solución viene dada por las tres funciones que reciben como parámetro la matriz a y su dimensión, y en el primer caso devuelve la traspuesta en la propia matriz, y en el segundo y en el tercer caso, devuelven *verdadero* o *falso* dependiendo de que cumplan o no la condición.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#define max 11
void traspuesta(int n, float a[][max]);
int simetrica (int n, float a[][max]);
int antisimetrica (int n, float a[][max]);

void traspuesta(int n, float a[][max])
{
    int i,j;
    float aux;
    for (i = 0; i < n; i++)
        for (j = 0; j < i; j++)
        {
            aux = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = aux;
        }
}

int simetrica (int n, float a[][max])
```



```

{
    int i,j,sime = 1;
    for (i = 0; i <= n; i++)
        for (j = 0; j <= n; j++)
            if (!(a[i][j] == a[j][i]))
                sime = 0;
    return sime;
}
int antisimetrica (int n, float a[][max])
{
    int i,j,sime;
    sime=1;
    for (i = 0; i <= n; i++)
        for (j = 0; j <= n; j++)
            if (!(a[i][j] == -a[j][i]))
                sime = 0;
    return sime;
}

```

9.18. *Escriba una función que encuentre el elemento mayor y menor de una matriz, así como las posiciones que ocupa y se escriban por pantalla.*

Análisis del problema

La solución se presente mediante una función que recibe la dimensión de la matriz y la propia matriz, y calcula ambos elementos sendos bucles *voraces*, que calculan a la vez el mayor y el menor, así como las posiciones.

Codificación

```

#include <stdio.h>
#define max 11
void mayormenor(int n, float a[][max]);

void mayormenor(int n, float a[][max])
//Calcula y escribe el mayor elemento de la matriz y su posición
//Calcula y escribe el menor elemento de la matriz y su posición
{
    int i,j,iM, im, jM,jm ;
    float mayor, menor;
    mayor=a[0][0];
    menor=mayor;
    im=0;
    iM=0;
    jm=0;
    jM=0;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            if (menor > a[i][j])
            {
                menor = a[i][j];
                im = i;
                jm = j;
            }
        }
    }

```

```

    };
    if (mayor < a[i][j])
    {
        mayor = a[i][j];
        iM = i;
        jM = j;}
    }
    printf(" el mayor elemento es %.0f\n", mayor);
    printf(" la posicion del mayor es %d %5d\n", iM + 1,jM +1);
    printf(" el menor elemento es %.0f\n", menor);
    printf(" la posicion del menores %5d %5d\n", im + 1, jm + 1);
}

```

- 9.19.** *Se dice que una matriz tiene un punto de silla si alguna posición de la matriz es el menor valor de su fila, y a la vez el mayor de su columna. Escribir una función que tenga como parámetro una matriz de números reales, y calcule y escriba los puntos de silla que tenga, así como las posiciones correspondientes.*

Análisis del problema

Para resolver el problema se supone que la matriz no tiene elementos repetidos. La función recibe como parámetro un número entero n , y una matriz de reales $a[\text{max}][\text{max}]$ (max es una constante previamente declarada). La codificación se ha planteado de la siguiente forma: un primer bucle recorre las distintas filas de la matriz. Dentro de ese bucle se calcula el elemento menor de la fila y la posición de la columna en la que se encuentra. Posteriormente se comprueba si la columna en la que se ha encontrado el elemento menor de la fila cumple la condición de que es el elemento mayor de la columna. En caso positivo será un punto de silla.

Codificación

```

void puntosdesilla(int n, float a[][max])
{
    int i,j, menor, jm,correcto;
    printf(" puntos de silllas\n");
    for(i = 0; i < n; i++)
    {
        menor = a[i][0];
        jm = 0;
        for(j = 1; j < n; j++)
            if (menor > a[i][j])
            {
                menor =a[i][j];
                jm=j;
            }
        correcto = 1;
        j = 0;
        while(correcto&&(j<n))
        {
            correcto = menor >= a[j][jm];
            j++;
        }
        if (correcto)
            printf("silla en %d %d valor = %f \n", i,jm,a[i][jm]);
    }
}

```

9.20. Escriba un programa que lea en una cadena de caracteres un número entero, y convierta la cadena a número.

Análisis del problema

La variable `cadena` sirve para leer el número introducido por teclado en una variable tipo cadena de caracteres de longitud máxima 80. La función `valor_numerico`, convierte la cadena de caracteres en un número entero con su signo. Esta función se salta todos los blancos y tabuladores que se introduzcan antes del número entero que es dato. Para convertir la cadena en número se usa esta descomposición: $'2345' = 2*1000 + 3*100 + 4*10 + 5$. La realización de estas operaciones es hecha por el método de *Horner*:

$$2345 = ((0*10 + 2)*10 + 3)*10 + 4)*10 + 5$$

La obtención de, por ejemplo el número 3 se hace a partir del carácter '3': $3 = '3' - '0'$

Codificación

```
#include <stdio.h>
#include <string.h>
int valor_numerico(char cadena[]);

void main(void)
{
    char cadena[80];
    int numero;
    printf("dame numero: ");
    gets(cadena);
    numero= valor_numerico(cadena);
    puts(cadena);
    printf(" valor leído %d \n ",numero);
}

int valor_numerico(char cadena[])
{
    int i,n,sign;

    /* salto de blancos y tabuladores */
    for (i = 0; cadena[i] == ' ' || cadena[i] == '\t'; i++);
    /* determinación del signo*/
    sign = 1;
    if(cadena[i] == '+' || cadena[i] == '-')
        sign = cadena[i++] == '+' ? 1:-1;
    /* conversión a número*/
    n = 0;
    for (; cadena[i] >= '0' && cadena[i] <= '9' ; i++)
        n = 10*n + cadena[i] - '0';
    return (sign*n);
}
```

9.21. Escriba un programa en C que lea un número en base 10 en una variable entera. Lea una base y transforme el número leído en base 10 a la otra base como cadena de caracteres. Posteriormente, lea un número como cadena de caracteres en una cierta base y lo transforme al mismo número almacenado en una variable numérica escrito en base 10. Por último el último número leído como cadena en una base, lo transforme a otro número almacenado en una cadena y escrito en otra base que se leerá como dato.

Análisis del problema

La solución se ha estructurado de la siguiente forma:

- Una función *cambio* recibe como dato el número almacenado en una variable numérica y lo transforma en el mismo número almacenado en una cadena de caracteres en otra cierta base que también es recibida como parámetro en otra variable entera. Para poder realizar lo indicado usa una cadena de caracteres llamada *bases* que almacena los distintos caracteres a los que se transformarán los números: así por ejemplo el número 5 se transforma en '5', el número 10 se transforma en 'A', el número 11 en 'B', etc. Se define un vector de enteros *aux* que almacena en cada una de las posiciones (como resultado intermedio) el dígito entero al que corresponde cuando se pasa a la base que se recibe como parámetro. Para hacer esto se usa el esquema siguiente: *mientras* la variable *num* sea distinta de cero *hacer* en la posición *i* del vector *aux* que se almacene el resto de la división entera del número *num* entre la *base*, para posteriormente almacenar en la variable *num* el cociente de la división entera del número entre la *base*. Para terminar se pasa el número a cadena de caracteres usando el array *bases*.
- Una función *valore* recibe como parámetro la cadena de caracteres y la base en que está escrita, y lo transforma a su correspondiente valor numérico devolviéndolo en la propia función, que es declarada como de tipo *entero largo*. Para realizar la operación se usa, al igual que en el ejercicio anterior el método de *Horner* de evaluación de un polinomio escrito en una cierta base:

$$'2345'_{base} = ((0 * base + 2) * base + 3) * base + 4) * base + 5.$$

- Para pasar de un número almacenado en una cadena de caracteres y escrito en una base al mismo número almacenado en otra cadena de caracteres y escrito en otra base se usa la función *cambiodoble* que llama a las dos funciones anteriores.
- Por último el *programa principal* escribe usando el formato correspondiente un número entero almacenado en una variable entera en base octal y base hexadecimal.

Codificación (Consultar la página web del libro)

- 9.22.** *Escriba un programa que lea un texto de la entrada hasta fin de fichero y calcule el número de: letras a, letras e, letras i, letras o, letras u leídas, y cuente el número de palabras y el número de líneas leídas.*

Análisis del problema

Se declara un vector entero de cinco componentes que sirve para contar las vocales correspondientes. Un contador *npal*, cuenta el número de palabras, y otro contador *nlineas* cuenta el número de líneas. Un bucle itera mientras no sea *fin de archivo* (*control + Z*), y otro bucle itera mientras esté en una palabra, y además cuenta las vocales, en caso de que lo sean.

Codificación

```
#include <stdio.h>
#include <string.h>
#define max 5

void main(void)
{
    int npal = 0, nlin = 0, cont[max];
    char c;
    clrscr();
    printf (" \t\t TEXTO \n");
    for (i = 0; i < max; i++)
        cont[i] = 0;
    while ((c = getchar()) != EOF)
    {
        while ((c != '\t') && (c != ' ') && (c != '\n') && (c != EOF))
```

```

    {
        c=getchar();
        if (c == 'a' || c == 'A') cont[0]++;
        if (c == 'e' || c == 'E') cont[1]++;
        if (c == 'i' || c == 'I') cont[2]++;
        if (c == 'o' || c == 'O') cont[3]++;
        if (c == 'u' || c == 'U') cont[4]++;
    }
    npal++;
    if (c == '\n')
nlin++;
}
printf(" palabras %d lineas %d \n", npal, nlin);
printf(" numero de a_es %d \n", cont[0]);
printf(" numero de e_es %d \n", cont[1]);
printf(" numero de i_es %d \n", cont[2]);
printf(" numero de o_es %d \n", cont[3]);
printf(" numero de u_es %d \n", cont[4]);
getch();
}

```

9.23. Escriba un programa que lea una frase, y decida si es palíndroma.

Análisis del problema

Una frase es palíndroma si puede leerse de igual forma de izquierda a derecha, que de derecha a izquierda, después de haber eliminado los blancos. Por ejemplo la frase *dabale arroz a la zorra el abad* es palíndroma. Para resolver el problema, se lee una frase en una cadena de caracteres. Se copia la frase en otra cadena de caracteres, eliminando los blancos. Posteriormente mediante un bucle que llegue hasta la mitad de la frase, se comprueba si la letra que ocupa la posición *i_ésima* coincide con la correspondiente que ocupa la posición *n-i_ésima*, siendo *n* la longitud de la cadena.

Codificación

```

#include <stdio.h>
#include <string.h>

void main(void)
{
    char cadena[80], cadenal[80];
    int i,j, palindroma;
    printf("dame cadena "); gets(cadena);
    j=0;
    for(i = 0; cadena[i]; i++)
        if (cadena[i] != ' ')
        {
            cadenal[j] = cadena[i];
            j++;
        }
    palindroma = 1;
    i = 0;
    while (i <= j / 2 && palindroma)
    {
        palindroma = cadenal[i] == cadenal[j - 1 - i];
    }
}

```

```

    i++;
}
if (palindroma)
    printf(" Es palindroma ");
else
    printf(" No es palindroma");
}

```

9.24. Un número entero es primo si ningún otro número primo más pequeño que él es divisor suyo. A continuación escribir un programa que rellene una tabla con los 100 primeros números primos y los visualice

Análisis del problema

El programa se ha estructurado de la siguiente forma:

- Se declaran la constante `max` y los valores lógicos `TRUE` y `FALSE`.
- El programa principal se encarga de declarar un array para almacenar los `max` números primos, y mediante un bucle `for` se rellena de números primos, llamando a la función `primo`. El número `n` que se pasa a la función `primo(n)` se incrementa en dos unidades (todos los números primos son impares). El primer número primo, que es el 2, es el único que no cumple esta propiedad.
- Función `primo`. Un número entero positivo es primo, si sólo tiene por divisores la unidad y el propio número. El método que se usa, aplica la siguiente propiedad “si un número mayor que la raíz cuadrada de `n` divide al propio `n` es porque hay otro número entero menor que la raíz cuadrada que también lo divide”. Por ejemplo. Si `n` vale 64 su raíz cuadrada es 8. El número 32 divide a 64 que es mayor que 8 pero también lo divide el número 2 que es menor que 8, ya que $2 \cdot 32 = 64$. De esta forma para decidir si un número es primo basta con comprobar si tiene divisores menores o iguales que su raíz cuadrada por supuesto eliminando la unidad.
- Función `escribe`. Se encarga de escribir la lista de los 100 números primos en 10 líneas distintas.

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

9.1. Determinar la salida de cada segmento de programa para el correspondiente archivo de entrada que se indica al final.

```

int i,j,k;
int Primero[21];
for(j = 0; j < 7; )
    scanf("%d",&Primero[j++]);
i = 0
j = 1;
while (( j < 6) && (Primero[j - 1] <
Primero[j]))
{
    i++;
    j++;
}

```

```

for(k = -1; k < j + 2; k = k + 2)
    printf("%d",Primero[ ++ k]);

```

20 60 70 10 0 40 30 90

9.2. Determinar la salida de cada segmento de programa.

```

int i,j,k;
int Tercero[6][12];
for(i = 0; i < 3; i++)
    for(j = 0; j < 12; j++)
        Tercero[i][j] = i + j + 1;
for(i = 0; i < 3; i++)
{
    j = 2;
    while (j < 12)

```

```

    {
        printf("%d \n", i, j, Tercero[i][j]);
        j += 3;
    }
}

```

- 9.3.** El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando letras individuales. Diseñar un programa para jugar al ahorcado. *Sugerencia:* almacenar una lista de palabras en un array y seleccionar palabras aleatoriamente.
- 9.4.** Escribir un programa que lea una colección de cadenas de caracteres de longitud arbitraria. Por cada cadena leída, su programa hará lo siguiente:
- Imprimir la longitud de la cadena.
 - Contar el número de ocurrencia de palabras de cuatro letras.
 - Sustituir cada palabra de cuatro letras por una cadena de cuatro asteriscos e imprimir la nueva cadena.
- 9.5.** Diseñar un programa que determine la frecuencia de aparición de cada letra mayúscula en un texto escrito por el usuario (fin de lectura, el punto o el retorno de carro, ASCII 13).
- 9.6.** Escribir un programa que lea una cadena de caracteres y la visualice en un cuadro.
- 9.7.** Escribir un programa que lea una frase, sustituya todas las secuencias de dos o más blancos por un solo blanco y visualice la frase restante.
- 9.8.** Escribir un programa que lea una frase y a continuación visualice cada palabra de la frase en columna, seguido del número de letras que compone cada palabra.
- 9.9.** Escribir un programa que desplace una palabra leída del teclado desde la izquierda hasta la derecha de la pantalla.
- 9.10.** Escribir un programa que lea una línea de caracteres, y visualice la línea de tal forma que las vocales sean sustituidas por el carácter que más veces se repite en la línea.
- 9.11.** Escribir un programa que lea una serie de cadenas, a continuación determine si la cadena es un identificador válido según la sintaxis de C. *Sugerencias:* utilizar las siguientes funciones: *longitud* (tamaño del identificador en el rango permitido); *primero* (determinar si el nombre comienza con un símbolo permitido); *restantes* (comprueba si los restantes son caracteres permitidos).
- 9.12.** Se introduce una frase por teclado. Se desea imprimir cada palabra de la frase en líneas diferentes y consecutivas.
- 9.13.** Escribir un programa que tenga como entrada una palabra y n líneas. Se quiere determinar el número de veces que se encuentra la palabra en las n líneas.
- 9.14.** Escribir un programa en el que se genere aleatoriamente un vector de 20 números enteros. El vector ha de quedar de tal forma que la suma de los 10 primeros elementos sea mayor que la suma de los 10 últimos elementos. Mostrar el vector original y el vector con la distribución indicada.

Algoritmos de ordenación y búsqueda

Una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación*. El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. El capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en C. De igual modo se estudiará el análisis de los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

10.1 Ordenación

La **ordenación** o **clasificación** de datos es la operación consistente en disponer un conjunto de datos en algún determinado orden con respecto a uno de los *campos* de elementos del conjunto. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina *clave*. Una *lista* dice que *está ordenada por la clave k* si la lista está en orden ascendente o descendente con respecto a esta clave. La lista se dice que está en *orden ascendente* si:

$i < j$ *implica que* $k[i] \leq k[j]$
 y se dice que está en *orden descendente* si: $i > j$ *implica que* $k[i] \leq k[j]$

En todos los métodos de este capítulo, se utiliza el *orden ascendente* sobre vectores o listas (*arrays* unidimensionales). La *eficiencia* es el factor que mide la calidad y rendimiento de un algoritmo. En el caso de la operación de ordenación, dos criterios se suelen seguir a la hora de decidir qué algoritmo (de entre los que resuelven la ordenación: 1) *tiempo menor de ejecución en computadora*; 2) *menor cantidad de memoria utilizada*.

Los métodos de ordenación pueden ser internos o externos según que los elementos a ordenar estén en la memoria principal o en la memoria externa. Los métodos de ordenación se suelen dividir en dos grandes grupos:

- *directos*: burbuja, selección, inserción;
- *indirectos* (avanzados): *shell*, ordenación rápida, ordenación por mezcla, *radixsort*;

10.2 Ordenación por burbuja

En el método de ordenación por burbuja los valores más pequeños “*burbujean*” gradualmente (suben) hacia la cima o parte superior del array de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la

parte inferior del array. La técnica consiste en hacer varias pasadas a través del array. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el array. En el caso de un array (lista) con n elementos, la ordenación por burbuja requiere hasta $n-1$ pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha “burbujeado” hasta la cima de la sublista actual. Por ejemplo, después que la pasada 0 está completa, la cola de la lista $A[n-1]$ está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son :

- En la pasada 0 se comparan elementos adyacentes
 $(A[0], A[1]), (A[1], A[2]), (A[2], A[3]), \dots, (A[n-2], A[n-1])$
- Se realizan $n-1$ comparaciones, por cada pareja $(A[i], A[i+1])$ se intercambian los valores si $A[i+1] < A[i]$.
 Al final de la pasada, el elemento mayor de la lista está situado en $A[n-1]$.
- En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en $A[n-2]$.
- El proceso termina con la pasada $n-1$, en la que el elemento más pequeño se almacena en $A[0]$.

EJEMPLO 10.1. *Funcionamiento del algoritmo de la burbuja con un array de 5 elementos (50, 20, 40, 80, 30). Los movimientos de claves que se realizan son:*

```
50, 20, 40, 80, 30
20, 50, 40, 80, 30
20, 40, 50, 80, 30
20, 40, 50, 30, 80  el mayor se encuentra el último
```

```
20, 40, 30, 50, 80  ordenado
```

10.3 Ordenación por selección

El método de ordenación se basa en seleccionar la posición del elemento más pequeño del array y colocarlo en la posición que le corresponde. El algoritmo de selección se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista. En la primera pasada se busca el elemento más pequeño de la lista y se intercambia con $A[0]$, primer elemento de la lista. Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista $A[1], A[2] \dots A[n-1]$ permanece desordenada. La siguiente pasada busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena entonces en la posición $A[1]$. De este modo los elementos $A[0]$ y $A[1]$ están ordenados y la sublista $A[2], A[3] \dots A[n-1]$ desordenada; entonces, se selecciona el elemento más pequeño y se intercambia con $A[2]$. El proceso continúa $n-1$ pasadas; en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

EJEMPLO 10.2. *Funcionamiento del algoritmo de selección con un array de 5 elementos (50, 20, 40, 80, 30).*

Los movimientos de claves que se realizan son:

```
50, 20, 40, 80, 30
20, 50, 40, 80, 30
20, 30, 40, 80, 50
20, 30, 40, 50, 80
```

10.4 Ordenación por inserción

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, consistente en insertar un nombre en su posición correcta dentro de una lista que ya está ordenada. El algoritmo considera que la lista $a[0], a[1], \dots, a[i-1]$ está ordenada, posteriormente inserta el elemento $a[i]$ en la posición que le corresponda para que la lista $a[0], a[1], \dots, a[i-1], a[i]$ esté ordenada, moviendo hacia la derecha los elementos que sean necesarios. Si un bucle que comienza en la posición $i=1$ y avanza de uno en uno hasta la posición $n-1$ y realiza el proceso anterior, al terminar el proceso el vector estará ordenado.

EJEMPLO 10.3 *Funcionamiento del algoritmo de inserción con un array de 5 elementos (50, 20, 40, 80, 30).*

Los movimientos de claves que se realizan son:

50, 20, 40, 80, 30

20, 50, 40, 80, 30

20, 40, 50, 80, 30

20, 40, 30, 50, 80

10.5 Ordenación *Shell*

La ordenación *Shell* se suele denominar también *ordenación por inserción con incrementos decrecientes*. Es una mejora de los métodos de inserción directa en el que se comparan elementos que pueden ser no contiguos. El algoritmo es el siguiente:

1. Se divide la lista original en $n/2$ grupos de dos elementos, considerando un incremento o salto entre los elementos de $n/2$.
2. Se clasifica cada grupo por separado, comparando las parejas de elementos y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos ($n/4$), con un incremento o salto entre los elementos también mitad ($n/4$), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y después clasificando cada grupo por separado.
5. El algoritmo termina cuando se alcanza el tamaño de salto 1. En este caso puesto que se comparan elementos contiguos cuando termine el proceso el vector estará ordenado.

El método de ordenación *shell* ordena ya que cuando el salto es 1 el método funciona como el burbuja.

EJEMPLO 10.4. *Obtener las secuencias parciales del vector al aplicar el método *Shell* para ordenar en orden creciente la lista:*

6 1 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es $6/2 = 3$. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	Salto	Intercambios	Lista
1	3	(6,2),(5,4),(6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	ninguno	0 1 4 2 3 5 6
salto $3/2 = 1$			
4	1	(4,2),(4,3)	0 1 2 3 4 5 6
5	1	ninguno	0 1 2 3 4 5 6

10.6 Ordenación rápida (*QuickSort*)

El algoritmo conocido como *quicksort* (ordenación rápida) recibe su nombre de su autor, Tony Hoare. Usa la técnica divide y vencerás y normalmente la recursividad en la implementación. El método se basa en dividir los $n > 0$ elementos de la lista a ordenar en dos particiones separadas, de tal manera, que los elementos pequeños están en la izquierda, y los grandes a la derecha. Para realizar esta dos particiones, se elige un *pivote o elemento de partición* y se colocan todos los elementos menores o iguales que él a la parte izquierda del array y los elementos mayores o iguales que él a la derecha. Posteriormente se ordena la izquierda y la derecha recursivamente.

10.7 Búsqueda en listas: búsqueda secuencial y binaria

El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. Las técnicas de búsqueda más utilizadas son : *búsqueda lineal o secuencial*, la técnica más sencilla y *búsqueda binaria o dicotómica*, la técnica más eficiente.

La *búsqueda secuencial* de una clave en un vector se basa en comparar la clave con los sucesivos elementos del vector tratados secuencialmente (de izquierda a derecha o viceversa). Si el vector se encuentra ordenado, la búsqueda secuencial puede interrumpirse sin éxito cuando se esté seguro de que no se puede encontrar la clave en el vector, ya que los elementos que quedan por comparar, se sabe que son mayores que la clave o bien menores.

La *búsqueda binaria* se aplica a vectores ordenados. Para buscar si una clave está en un vector se comprueba si la clave coincide con el valor del elemento central. En caso de que no coincida, entonces se busca a la izquierda del elemento central, si la clave a buscar es más pequeña que el elemento almacenado en la posición central y en otro caso se busca a la derecha.

PROBLEMAS RESUELTOS

10.1. *¿Cuál es la diferencia entre ordenación por selección y ordenación por inserción?*

Solución

El método de ordenación por selección selecciona la posición del elemento más pequeño (más grande) y lo coloca en el lugar que le corresponde. Por lo tanto, elige la posición del elemento más pequeño del vector y lo intercambia con el primero. Posteriormente elige la posición del siguiente más pequeño y lo intercambia con el que ocupa la posición segunda, etc. El método de ordenación por inserción , a partir de una parte del vector ordenado (hasta la posición i-1), decide cual es la posición donde debe colocar el elemento que ocupa la posición i del vector para que quede ordenado el vector hasta la posición i (debe desplazar datos hacia la derecha).

10.2. *Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿ cómo estarán colocados los elementos del vector después de cuatro pasadas más del algoritmo ?*
3 , 13, 8, 25, 45, 23, 98, 58.

Solución

Después de la primera pasada los elementos estarán en el orden: 3 , 8, 13, 25, 45, 23, 98, 58.
Después de la segunda pasada los elementos estarán en el orden: 3 , 8, 13, 25, 45, 23, 98, 58.
Después de la tercera pasada los elementos estarán en el orden: 3 , 8, 13, 25, 45, 23, 98, 58.
Después de la cuarta pasada los elementos estarán en el orden: 3 , 8, 13, 23, 25, 45, 98, 58.

10.3. *Dada la siguiente lista 47 , 3 21, 32, 56, 92. Después de dos pasadas de un algoritmo de ordenación, el array se ha quedado dispuesto así: 3, 21, 47, 32, 56, 92. ¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción) ?. Justifique la respuesta.*

Solución

El método de la burbuja no puede ser el empleado, ya que tras la primera pasada del método, los elementos deberían estar en el orden siguiente: 3, 21, 32, 47, 56, 92. Es decir ya estarían ordenados.

El método de ordenación por selección, tras la primera pasada tendría los elementos colocados de la siguiente forma: 3, 47, 21, 32, 56, 92.

Tras la segunda pasada los elementos del vector estaría colocados : 3, 21, 47, 32, 56, 92. Por lo tanto este método puede haber sido usado.

El método de ordenación por inserción tras la primera pasada los datos estarían: 3, 47, 21, 32, 56, 92. Una vez realizada la segunda pasada la información del vector sería: 3, 21, 47, 32, 56, 92. Es decir, también podría haberse usado este método de ordenación.

10.4. *Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.*
8, 13, 17, 26, 44, 56, 88, 97.

Solución

En la primera iteración del bucle la clave central sería 26, y como no coincide con la clave a buscar que es 88, entonces habría que realizar la búsqueda en: 44, 56, 88, 97. En la segunda iteración del bucle, la clave central sería 26, y como tampoco coincide con la CLAVE a buscar que es 88, entonces habría que realizar la búsqueda en: 88, 97. En la tercera iteración, la clave central es 88 que coincide con la clave que se busca, por lo tanto se terminaría el bucle con éxito.

- 10.5.** *Escriba una función que realice la ordenación interna de un vector de n elementos, por el método de ordenación de burbuja.*

Análisis del problema

La ordenación por burbuja se basa en comparar elementos contiguos del vector e intercambiar sus valores si están desordenados. Si el vector tiene los elementos $a[0], a[1], \dots, a[n-1]$. El método comienza comparando $a[0]$ con $a[1]$; si están desordenados, se intercambian entre sí. A continuación se compara $a[1]$ con $a[2]$. Se continua comparando $a[2]$ con $a[3]$, intercambiándolos si están desordenados,... hasta comparar $a[n-2]$ con $a[n-1]$ intercambiándolos si están desordenados. Estas operaciones constituyen la primera pasada a través de la lista. Al terminar esta pasada el elemento mayor está en la parte superior de la lista. El proceso descrito se repite durante $n-1$ pasadas teniendo en cuenta que en la pasada i el se ha colocado el elemento mayor de las posiciones $0, \dots, n-i$ en la posición $n-i$. De esta forma cuando i toma el valor $n-1$, el vector está ordenado.

Codificación

```
void burbuja( int  n, float  a[max])
{
    int i,j;  float  aux;
    for (i = 0; i < n - 1; i++)
        for(j = 0; j < n - 1 - i; j++)
            if (a[j] > a[j + 1])
            {
                aux = a[j];
                a[j] =a [j + 1];
                a[j + 1] =aux;
            }
}
```

- 10.6.** *Escriba una función que realice la ordenación interna de un vector de n elementos, por el método de ordenación de selección.*

Análisis del problema

Se basa en el siguiente invariante “seleccionar en cada pasada el elemento más pequeño y colocarlo en la posición que le corresponde”. De esta forma en una primera pasada, encuentra el menor de todos los elementos y lo coloca en la posición primera del vector (la cero). En la segunda pasada encuentra el siguiente elemento más pequeño (el segundo elemento más pequeño) y lo coloca en la posición número 2 (la uno). Y así continúa hasta que coloca los $n-1$ elementos más pequeños quedando por tanto el vector ordenado.

Codificación

```
void seleccion( int  n, float  a[max])
{
    int i,j,k;  float  aux;
    for (i = 0; i < n - 1; i++)
```

```

{
    k = i;
    for(j = i + 1; j < n; j++)
        if(a[j] < a[k])
            k = j;
    aux = a[k];
    a[k] = a[i];
    a[i] = aux;
}
}

```

10.7. Escriba funciones para realizar la búsqueda secuencial de un vector.

Análisis del problema

Si se supone declarada previamente una constante `max`, que es el tamaño del vector `a`, y que el elemento que se busca es `x`. La búsqueda, devuelve un valor entero, que es `-1` en el caso de que el elemento a buscar no esté en el vector. La búsqueda secuencial se programa, primeramente de forma descendente (el bucle termina cuando ha encontrado el elemento, o cuando no hay más elementos en el vector) suponiendo que el vector está ordenado, y posteriormente de forma ascendente (el bucle termina cuando se ha encontrado el elemento, o cuando se está seguro de que no se encuentra en el vector), suponiendo que el vector está ordenado.

Codificación

```

int Bsecdes(int n, float a[max], float x)
{
    int enc = 0; int i = n - 1;
    // Búsqueda secuencial descendente
    while ((!enc) && (i >= 0))
    {
        enc = (a[i] == x);
        if (!enc)
            i--;
    }
    // Si se encuentra se devuelve la posición en el vector*
    if (enc)
        return (i);
    else
        return (-1);
}

int Bsecor(int n, float a[max], float x)
{
    int enc = 0; int i = 0;
    // Búsqueda secuencial ascendente.
    while ((!enc) && (i < max))
    {
        enc = (a[i] >= x);
        // enc se hace verdadero cuando lo encuentra o no esta
        if (!enc)
            i++;
    }
    if (i < n)

```

```

    enc = a[i] == x ;
    //Si se encuentra el elemento se devuelve la posición
    if (enc)
        return (i);
    else
        return (-1);
}

```

10.8. *Escriba funciones para realizar la búsqueda binaria de una clave en un vector ordenado ascendentemente.*

Análisis del problema

La búsqueda dicotómica se programa, tomando en cada partición el elemento central *c*. Decidiendo si el elemento buscado se encuentra en esa posición, o bien si hay que mover el índice izquierdo o derecho de la partición. El bucle que realiza la búsqueda termina cuando ha encontrado el elemento o bien cuando los índices de la partición se han cruzado.

Codificación

```

int Bdicotomicacor(int n, float a[max], float x)
{
    int iz, de, c, enc;
    iz = 0;
    de = n - 1;
    enc = 0;
    while((iz <= de) && (!enc))
    {
        c = (iz + de) / 2;
        if (a[c] == x)
            enc = 1;
        else if (x < a[c])
            // debe encontrarse a la izquierda de c. Retrocede de
            de = c - 1;
        else
            // debe encontrarse a la derecha de c. Avanza iz
            iz = c + 1;
    }
    if(enc)
        return (c);
    else
        return (-1);
}

```

10.9. *Escribir una función de búsqueda binaria aplicada a un array ordenado descendientemente.*

Análisis del problema

La búsqueda dicotómica descendente de un vector ordenado descendientemente es análoga a la búsqueda dicotómica en un vector ordenado ascendentemente. Solamente cambian los movimientos de los índices *iz* y *de*; esto se consigue simplemente cambiando la condición de selección del `if(x < a[c])` por esta otra `if (x > a[c])` del ejercicio 10.8.

Codificación

```

int Bdicotomicades(int n, float a[max], float x)

```

```

{
    int  iz, de, c, enc;
    iz = 0;
    de = n - 1;
    enc = 0;
    while((iz <= de) && (!enc))
    {
        c = (iz + de) / 2;
        if (a[c] == x)
            enc = 1;
        else if (x > a[c])
            // debe encontrarse a la izquierda de c. Retrocede de
            de = c - 1;
        else
            // debe encontrarse a la derecha de c. Avanza iz
            iz = c + 1;
    }
    if(enc)
        return (c);
    else
        return (-1);
}

```

10.10. Escriba dos funciones que realicen la ordenación interna de un vector de n elementos, por los métodos de ordenación de inserción.

Análisis del problema

Se basa en realizar un total de $n-1$ iteraciones sobre un vector que almacene n datos. En la iteración número i , se cumple antes de empezar que el vector se encuentra ordenado desde las posiciones $0, 1, \dots, i-2$, y al final de la pasada el vector queda ordenado hasta la posición número $i-1$. Para realizarlo, el método realiza una búsqueda secuencial o binaria (dos métodos distintos) de la posición k donde debe colocarse el elemento que ocupa en el vector la posición $i-1$. Posteriormente intercambia los contenidos de las posiciones k , e $i-1$. En la codificación que se presenta hay que tener en cuenta que si i comienza por cero entonces el número de iteraciones es siempre uno más. Se codifica el método de ordenación por inserción lineal y binaria.

Codificación (Consultar la página web del libro)

10.11. Escriba una función que realice la ordenación interna de un vector de n elementos, por los métodos de ordenación Shell.

Análisis del problema

Se divide el vector original (n elementos) en $n/2$ listas de dos elementos con un intervalo entre los elementos de cada lista de $n/2$ y se clasifica cada lista por separado. Se divide posteriormente el vector en $n/4$ listas de cuatro elementos con un intervalo o salto de $n/4$ y, de nuevo, se clasifica cada lista por separado. Se repite el proceso dividiendo en grupos $n/8, n/16, \dots$ (esta secuencia puede cambiarse) hasta que, en un último paso, se clasifica la lista de n elementos. La clasificación de cada una de las listas puede hacerse por cualquier método en este caso se hará por el método de inserción lineal.

Codificación

```

void shell( int  n, float  a[max])
{
    int  i, j, k, salto;

```

```

float aux;
salto = n / 2;
while (salto > 0)
{
    for (i = salto; i < n; i++)
    {
        j = i - salto;
        while(j >= 0)
        {
            k = j + salto;
            if (a[j] <= a[k])
                j = -1;
            else
            {
                aux = a[j];
                a[j] = a[k];
                a[k] = aux;
                j = j - salto;
            }
        }
    }
    salto = salto / 2;
}

```

- 10.12.** Escriba una función recursiva que implemente el método de ordenación rápida *Quick_Sort* que ordene un array de n elementos.

Análisis del problema

El algoritmo de ordenación rápida divide el array a en dos subarrays (sublistas). Se selecciona un elemento específico del array $a[\text{centro}]$ llamado pivote y se divide el array original en dos subarrays, de tal manera que los elementos menores o iguales que $a[\text{centro}]$ se colocan en la parte izquierda y los mayores o iguales en la parte derecha. Posteriormente se ordenan la parte izquierda y la derecha mediante dos llamadas recursivas.

Codificación (Consultar la página web del libro)

- 10.13.** Escribir una función que acepte como parámetro un vector que puede contener elementos duplicados. La función debe sustituir cada valor repetido por -1 y devolver al punto donde fue llamado el vector modificado y escribir el número de entradas modificadas (puede suponer que el vector dato no contiene el valor -1 cuando se llama a la función).

Análisis del problema

Se supone que el vector es de dimensión n , y que $n \leq \text{max}$ siendo max una constante previamente declarada. El contador nv cuenta el número de veces que se cambia un valor por -1 . El contador nv1 , cuenta el número de elementos que en la iteración i son cambiados por -1 . El problema se ha resuelto de la siguiente forma: un bucle controlado por la variable i , recorre el vector. Para cada posición i cuyo contenido sea distinto de la marca -1 se comprueba, se cuenta y se cambia por el valor predeterminado -1 mediante otro bucle controlado por la variable entera j , aquellos valores que cumplen la condición $a[i] == a[j]$, siempre que $a[i] <> -1$. Al final del bucle j , si se ha cambiado algún elemento por el valor predeterminado, se cambia también el valor de la posición i , y por supuesto se cuenta.

Codificación

```

void duplicados(int n, float a[max])
{
    int i, j;
    float aux, nv, nv1;
    aux = -1;
    nv = 0;
    for (i = 0; i < n - 1; i++)
    {
        nv1 = 0;
        for(j = 0; j < n - 1 - i; j++)
            if (a[i] != -1)
                if (a[j] == a[i] )
                {
                    a[j] = aux;
                    nv1++;
                }
        if (nv1 > 0)
        {
            a[i] = aux;
            nv1++;
            nv+ = nv1;
        }
    }
    printf(" numero de duplicados = %d \n", nv);
}

```

PROBLEMAS PROPUESTOS

- 10.1.** Un array de registros se quiere ordenar según el campo clave *fecha de nacimiento*. Dicho campo consta de tres subcampos: día, mes y año de 2, 2 y 4 dígitos respectivamente. Adaptar el método de la burbuja a esta ordenación.
- 10.2.** Suponga que se tiene una secuencia de n números que deben ser clasificados:
1. Si se utiliza el método de *Shell*, ¿cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si: ya está clasificado; está en orden inverso.
 2. Realizar los mismos cálculos si se utiliza el algoritmo *quicksort*.
- 10.3.** Escriba la función de ordenación correspondiente al método *Shell* para poner en orden alfabético una lista de n nombres.
- 10.4.** Dado un vector x de n elementos reales, donde n es impar, diseñar una función que calcule y devuelva la mediana de ese vector. La mediana es el valor tal que la mitad de los números son mayores que el valor y la otra mitad son menores. Escribir un programa que compruebe la función.
- 10.5.** Se trata de resolver el siguiente problema escolar. Dadas las notas de los alumnos de un colegio en el primer curso de bachillerato, en las diferentes asignaturas (5, por comodidad), se trata de calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por orden decreciente de notas medias individuales. Nota: utilizar como algoritmo de ordenación el método *Shell*.
- 10.6.** Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1000 nombres y números de teléfono de un archivo que contiene los números en orden

aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

10.7. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria.

10.8. Se dispone de dos vectores, *Maestro* y *Esclavo*, del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier acción hecha con *Maestro[i]* debe hacerse a *Esclavo[i]*. Después de realizar la ordenación se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea. Nota: utilizar como algoritmo de ordenación el método quicksort.

10.9. Cada línea de un archivo de datos contiene información sobre una compañía de informática. La línea contiene el nombre del empleado, las ventas efectuadas por el mismo y el número de años de antigüedad del empleado en la compañía. Escribir un programa que lea la información del archivo de datos y a continuación se visualiza. La información debe ser ordenada por ventas de mayor a menor y visualizada de nuevo.

10.10. Se desea realizar un programa que realice las siguientes tareas:

- a) Generar, aleatoriamente, una lista de 999 números reales en el rango de 0 a 2000.
- b) Ordenar en modo creciente por el método de la burbuja.

c) Ordenar en modo creciente por el método Shell.

e) Buscar si existe el número x (leído del teclado) en la lista. Aplicar la búsqueda binaria.

10.11. Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

- t1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
- t2. Tiempo que se emplearía en *ordenar* la lista ya ordenada.
- t3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

10.12. Construir un método que permita ordenar por fechas y de mayor a menor un vector de n elementos que contiene datos de contratos ($n \leq 50$). Cada elemento del vector debe ser un objeto con los campos día, mes, año y número de contrato. Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos.

10.13. Se leen dos listas de números enteros, A y B de 100 y 60 elementos, respectivamente. Se desea resolver mediante funciones las siguientes tareas:

- a) Ordenar, aplicando el método de inserción, cada una de las listas A y B.
- b) Crear una lista C por intercalación o mezcla de las listas A y B.
- c) Visualizar la lista C ordenada.

Estructuras y uniones

Este capítulo examina estructuras, uniones, enumeraciones y tipos definidos por el usuario que permiten a un programador crear nuevos tipos de datos. La capacidad para crear nuevos tipos es una característica importante y potente de C y libera a un programador de restringirse al uso de los tipos ofrecidos por el lenguaje. Una *estructura* contiene múltiples variables, que pueden ser de tipos diferentes. La estructura es importante para la creación de programas potentes, tales como bases de datos u otras aplicaciones que requieran grandes cantidades de datos. Por otra parte, se analizará el concepto de *unión*, otro tipo de dato no tan importante como las estructuras o los array y el concepto de estructura (`struct`), de gran importancia en el tratamiento de la información.

Un tipo de dato enumerado es una colección de miembros con nombre que tienen valores enteros equivalentes. Un `typedef` es de hecho no un nuevo tipo de dato sino simplemente un sinónimo de un tipo existente.

11.1 Estructuras

Una *estructura* es una colección de uno o más tipos de elementos denominados *miembros*, cada uno de los cuales puede ser de un tipo de dato diferente. Una estructura puede contener cualquier número de miembros, cada uno de los cuales tiene un nombre único, denominado *nombre* del miembro.

Una estructura es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar. El formato de la declaración es:

```
struct <nombre de la estructura>
{
    <tipo de dato miembro1> <nombre miembro1>
    <tipo de dato miembro2> <nombre miembro2>
    ...
    <tipo de dato miembron> <nombre miembron>
};
```

Al igual que a los tipos de datos enumerados, a una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura. Del mismo modo que sucede en otras situaciones, en C existen dos conceptos similares a considerar, **declaración** y **definición**. Una declaración especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria; la declaración especifica un nuevo tipo de dato: `struct <nombre_estructura>`. Por consiguiente, cada definición de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado.

Las variables de estructuras se pueden definir de dos formas: (1) listándolas inmediatamente después de la llave de cierre de la declaración de la estructura, o (2) listando el tipo de la estructura creado seguida por las variables correspondientes en cualquier lugar del programa antes de utilizarlas. Se puede asignar una estructura a otra.

Se puede inicializar una estructura de dos formas. Se puede inicializar una estructura dentro de la sección de código de su programa, o bien se puede inicializar la estructura como parte de la definición. Cuando se inicializa una estructura como parte de la definición, se especifican los valores iniciales, entre llaves, después de la definición de variables estructura. El formato general en este caso:

```
struct <tipo> <nombre variable estructura> =
{ valor miembro1,
  valor miembro2,
  ...
  valor miembron
};
```

El operador `sizeof` se puede aplicar para determinar el tamaño que ocupa en memoria una estructura. Cuando se accede a una estructura, o bien se almacena información en la estructura o se recupera la información de la estructura. Se puede acceder a los miembros de una estructura de una de estas dos formas: (1) utilizando el *operador punto* (`.`), o bien (2) utilizando el *operador flecha* (`->`).

La asignación de datos a los miembros de una variable estructura se hace mediante el operador punto. La sintaxis en C es:

```
<nombre variable estructura> . <nombre miembro> = datos;
```

El operador punto proporciona el camino directo al miembro correspondiente. Los datos que se almacenan en un miembro dado deben ser del mismo tipo que el tipo declarado para ese miembro.

El operador puntero, `->`, sirve para acceder a los datos de la estructura a partir de un puntero. Para utilizar este operador se debe definir primero una variable puntero para apuntar a la estructura. A continuación, se utiliza simplemente el operador puntero para apuntar a un miembro dado. La asignación de datos a estructuras utilizando el operador puntero tiene el formato:

```
<puntero estructura> -> <nombre miembro> = datos;
```

Previamente habría que crear espacio de almacenamiento en memoria; por ejemplo, con la función `malloc()`.

Si se desea introducir la información en la estructura basta con acceder a los miembros de la estructura con el operador punto o el operador flecha(puntero). Se puede introducir la información desde el teclado o desde un archivo, o asignar valores calculados.

Se recupera información de una estructura utilizando el operador de asignación o una sentencia de salida (`printf()`, `puts()`...). Igual que antes, se puede emplear el operador punto o el operador flecha(puntero). El formato general toma uno de estos formatos:

1. `<nombre variable> = <nombre variable estructura>.<nombre miembro>;`
o bien
`<nombre variable> = <puntero de estructura> -> <nombre miembro>;`
2. para salida:
`printf(" ", <nombre variable estructura>.<nombre miembro>);`
o bien
`printf(" ", <puntero de estructura>-> <nombre miembro>);`

Una estructura puede contener otras estructuras llamadas *estructuras anidadas*. Las estructuras anidadas ahorran tiempo en la escritura de programas que utilizan estructuras similares. Se han de definir los miembros comunes sólo una vez en su propia estructura y a continuación utilizar esa estructura como un miembro de otra estructura. El acceso a miembros dato de estructuras anidadas requiere el uso de múltiples operadores punto. Las estructuras se pueden anidar a cualquier grado. También es posible inicializar estructuras anidadas en la definición.

Se puede crear un *array de estructuras* tal como se crea un array de otros tipos. Muchos programadores de C utilizan arrays de estructuras como un método para almacenar datos en un archivo de disco. Se pueden introducir y calcular sus datos de disco en arrays de estructuras y a continuación almacenar esas estructuras en memoria. Los arrays de estructuras proporcionan también un medio de guardar datos que se leen del disco. Los miembros de las estructuras pueden ser asimismo arrays.

C permite pasar estructuras a funciones, bien por valor o bien por referencia, utilizando el operador &. Si la estructura es grande, el tiempo necesario para copiar un parámetro struct a la pila puede ser prohibitivo. En tales casos, se debe considerar el método de pasar la dirección de la estructura.

EJEMPLO 11.1 Declaración de diferentes estructuras

```
struct complejo
{
    float x,y;
};

struct racional
{
    int numerador;
    int denominador;
};

struct fecha
{
    unsigned int mes, dia, anyo;
};

struct tiempo
{
    unsigned int horas, minutos;
};

struct direccion
{
    char calle[40];
    int num;
    int codpost;
    char ciudad [20];
}

struct entrada
{
    char nombre[50];
    struct direccion dir;
    char telefonos [5][15];
};
```

11.2 Uniones

Las uniones son similares a las estructuras en cuanto que agrupan a una serie de variables, pero la forma de almacenamiento es diferente y por consiguiente tiene efectos diferentes. Una estructura (struct) permite almacenar variables relacionadas juntas y almacenadas en posiciones contiguas en memoria. Las uniones, declaradas con la palabra reservada union, almacenan también miembros múltiples en un paquete; sin embargo, en lugar de situar sus miembros unos detrás de otros, en una unión, todos los miembros se solapan entre sí en la misma posición. El tamaño ocupado por una unión se determina así: se analiza el tamaño de cada variable de la unión; el mayor tamaño de variable será el tamaño de la unión. La sintaxis de una unión es la siguiente:

```
union nombre {
    tipo1 miembro1;
    tipo2 miembro2;
    ...
};
```

La cantidad de memoria reservada para una unión es igual a la anchura de la variable más grande. En el tipo `union`, cada uno de los miembros dato comparten memoria con los otros miembros de la unión.

Una razón para utilizar una unión es ahorrar memoria. En muchos programas se deben tener varias variables, pero no necesitan utilizarse todas al mismo tiempo. Para referirse a los miembros de una unión, se utiliza el operador punto (`.`), o bien el operador `->` si se hace desde un puntero a unión.

EJEMPLO 11.2 Declaración de una unión

```
union arg
{
    int v;
    char c[2];
} n;

printf ("Introduzca un número entero:");
scanf ("%d", &n.v);
printf ("La mitad más significativa del número es %i \n", c[1]);
printf ("La mitad menos significativa del número es %i \n", c[0]);
/* En algunos sistemas puede ser al revés */
```

11.3 Enumeraciones

Una enumeración es un tipo definido por el usuario con constantes de nombre de tipo entero. En la declaración de un tipo `enum` se escribe una lista de identificadores que internamente se asocian con las constantes enteras 0, 1, 2

Formato

1. `enum`

```
{
    enumerador1, enumerador2, ...enumeradorn.
};
```
2. `enum nombre`

```
{
    enumerador1, enumerador2, ...enumeradorn.
};
```

En la declaración del tipo `enum` pueden asociarse a los identificadores valores constantes en vez de la asociación que por defecto se hace (0, 1, 2 ...). Para ello se utiliza este formato:

3. `enum nombre`

```
{
    enumerador1 = expresión_constante1,
    enumerador2 = expresión_constante2,
    ...
    enumeradorn = expresión_constanten
};
```

El tamaño en bytes de una estructura, de una unión o de un tipo enumerado se puede determinar con el operador `sizeof`.

EJEMPLO 11.3 Diferentes tipos de enumeraciones

```
enum tipo_operacion {deposito, retirada, aldia, estado};

enum tipo_operación op;
...
```

```

switch (op)
{
    case deposito: realizar_deposito (args);
        break;
    case aldia: poner_al_dia (args);
        break;
    case retirada: retirar_fondos (args);
        break;
    case estado: imprimir_estado (args);
        break;
    default: imprimir_error (args);
}

```

11.4 Sinónimo de un tipo de datos: Typedef

La sentencia `typedef` permite a un programador crear un sinónimo de un tipo de dato definido por el usuario o de un tipo ya existente.

EJEMPLO 11.4 *Uso de typedef*

Uso de typedef para declarar un nuevo nombre, Longitud, de tipo de dato double.

```
typedef double Longitud;
```

A partir de la sentencia anterior, `Longitud` se puede utilizar como un tipo de dato, en este ejemplo sinónimo de `double`. La función `Distancia()`, escrita a continuación, es de tipo `Longitud`:

```

Longitud Distancia (const Punto& p, const Punto& p2)
{
    ...
    Longitud longitud = sqrt(rcua);
    return longitud;
}

```

Otros ejemplos:

```

typedef char* String;
typedef const char* string;

```

A continuación se pueden hacer las siguientes declaraciones con la palabra `String` o `string`:

```

String nombre = "Jesus Lopez Arrollo";
string concatena(string apell1, string apell2);

```

Sintaxis:

```
typedef tipo_dato_definido nuevo_nombre;
```

Puede declararse un tipo estructura o un tipo unión y a continuación asociar el tipo estructura a un nombre con `typedef`.

```
typedef struct complejo complex;
```

```
/* definición de un array de complejos */
```



```

complex v[12];

typedef struct racional
{
    int numerador;
    int denominador;
} Racional;

```

Ahora se puede declarar la estructura numero utilizando el tipo complex y el tipo Racional:

```

typedef struct numero
{
    complex a;
    Racional r;
};

```

EJEMPLO 11.5 *Uso de typedef*

```

typedef struct fecha Fecha;
typedef struct tiempo Tiempo;
typedef enum tipo_operacion TipOperacion;

struct registro_operacion
{
    long numero_cuenta;
    float cantidad;
    TipOperacion operacion;
    Fecha f;
    Tiempo t;
};

typedef struct registro_operacion RegistrOperacion;

RegistrOperacion entrada(void);

int main()
{
    RegistrOperacion w;
    w = entrada();

    printf("\n Operación realizada\n\n");
    printf("\t%d\n",w.numero_cuenta);
    printf("\t%d-%d-%d\n",w.f.dia,w.f.mes,w.f.anyo);
    printf("\t%d:%d\n",w.t.horas,w.t.minutos);

    return 0;
}

RegistrOperacion entrada(void)
{
    int x, y, z;
    RegistrOperacion una;

    printf("\nNúmero de cuenta: ");
    scanf("%ld",&una.numero_cuenta);
    puts("\tTipo de operación");
    puts("Deposito(0)");
}

```

```

puts("Retirada de fondos(1)");
puts("Puesta al día(2)");
puts("Estado de la cuenta(3)");
scanf("%d",&una.operacion);

printf("\nFecha (día mes año): ");
scanf("%d %d %d",&una.f.día,&una.f.mes,&una.f.año);

printf("Hora de la operación(hora minuto): ");
scanf("%d %d",&una.t.horas,&una.t.minutos);

return una;
}

```

11.5 Campos de bit

El lenguaje C permite realizar operaciones con los bits de una palabra. Ya se han estudiado los operadores de manejo de bits: `>>`, `<<`, `...`. Con los **campos de bit**, C permite acceder a un número de bits de una palabra entera. Un campo de bits es un conjunto de bits adyacentes dentro de una palabra entera.

La sintaxis para declarar campos de bits se basa en la declaración de estructuras. El formato general:

```

struct identificador_campo {
    tipo nombre1: longitud1;
    tipo nombre2: longitud2;
    tipo nombre3: longitud3;
    .
    .
    .
    tipo nombren: longitudn;
};

```

tipo ha de ser entero, int; generalmente unsigned int.
longitud es el número de bits consecutivos que se toman.

Al declarar campos de bits, la suma de los bits declarados puede exceder el tamaño de un entero; en ese caso se emplea la siguiente posición de almacenamiento entero. No está permitido que un campo de bits solape los límites entre dos int.

En la declaración de una estructura puede haber miembros que sean variables y otros campos de bits. Los campos de bits se utilizan para rutinas de encriptación de datos y fundamentalmente para ciertos interfaces de dispositivos externos. Los campos de bits tienen ciertas restricciones. Así, no se puede tomar la dirección de una variable campo de bits; no puede haber arrays de campos de bits; no se puede solapar fronteras de int. Depende del procesador el que los campos de bits se alineen de izquierda a derecha o de derecha a izquierda (conviene hacer una comprobación para cada procesador, utilizando para ello un union con variable entera y campos de bits).

EJEMPLO 11.6 Estructuras con campos de bit

```

struct entrada
{
    char nombre[50];
    struct direccion dir;
    char telefonos [5][15];
    int edad;
    int sexo:1; /* H: 1 - M: 0 */
    int departamento:3; /* código <8 */
    int contrato:3;
};

```

PROBLEMAS RESUELTOS

11.1. Encuentre los errores en la siguiente declaración de estructura y posterior definición de variable.

```
struct hormiga
{
    int patas;
    char especie[41];
    float tiempo;
};
hormiga colonia[100];
```

Es necesario conservar la palabra `struct` en la declaraciones de variables, a no ser que se añada una sentencia de tipo `typedef` como la siguiente:

```
typedef struct hormiga hormiga;
```

11.2. Declare una tipo de datos para representar las estaciones del año.

```
enum estaciones {PRIMAVERA =1, VERANO=2, OTONO=3, INVIERNO=4};
```

11.3. Escriba un función que devuelva la estación del año que se ha leído del teclado. La función debe de ser del tipo declarado en el ejercicio 2.

Análisis del problema

El tipo enumerado asocia enteros a nombres simbólicos, pero estos nombres simbólicos no pueden ser leídos desde una función estándar como `scanf`. Por consiguiente el programa tiene que leer los valores enteros y traducirlos a los nombres simbólicos que les corresponden según la definición del tipo enumerado.

Codificación

```
enum estaciones leerEstacion ( )
{
    int e;
    printf("Introduzca el número de la estación del año:\n");
    printf(" 1 - Primavera\n");
    printf(" 2 - Verano\n");
    printf(" 3 - Otoño\n");
    printf(" 4 - Invierno\n");
    scanf("%d", &e);
    switch (e)
    {
        case 1: return (PRIMAVERA); break;
        case 2: return (VERANO); break;
        case 3: return (OTOÑO); break;
        case 4: return (INVIERNO); break;
        default: printf ("Entrada errónea \n"); return;
    }
}
```

11.4. Declara un tipo de dato enumerado para representar los meses del año; el mes enero debe estar asociado al dato entero 1, y así sucesivamente los demás meses.

```
enum meses {ENERO=1,    FEBRERO=2,    MARZO=3,    ABRIL=4,    MAYO=5,
            JUNIO=6,    JULIO=7,    AGOSTO=8,    SEPTIEMBRE=9,
            OCTUBRE=10, NOVIEMBRE=11, DICIEMBRE=12};
```

11.5. Encuentra los errores del siguiente código

```
#include <stdio.h>
void escribe(struct fecha f);
int main( )
{
    struct fecha
    {
        int dia;
        int mes;
        int anyo;
        char mes[];
    } ff;
    ff = {1,1,2000,"ENERO"};
    escribe(ff);
    return 1;
}
```

Análisis del problema

La inicialización de una estructura puede hacerse solamente cuando es estática o global. No se puede definir un array de caracteres sin especificar el tamaño. La mayor parte de los compiladores tampoco permiten inicializar las estructuras de la manera que aparece en el ejemplo fuera de la inicialización de variables globales. Para estar seguros habría que inicializar la estructura `fecha` miembro a miembro.

11.6. ¿Con typedef se declaran nuevos tipos de datos, o bien permite cambiar el nombre de tipos de datos ya declarados?

Solución

La sentencia no añade ningún tipo de datos nuevo a los ya definidos en el lenguaje C. Simplemente permite renombrar un tipo ya existente, incluso aunque sea un nuevo nombre para un tipo de datos básico del lenguaje como `int` o `char`.

11.7. Declara un tipo de dato estructura para representar un alumno; los campos que tiene que tener son: nombre, curso, edad, dirección y notas de las 10 asignaturas. Declara otro tipo estructura para representar un profesor; los campos que debe tener son: nombre, asignaturas que imparte y dirección. Por último declara una estructura que pueda representar un profesor o a un alumno.

Solución

```
struct alumno
{
    char nombre [40];
    int curso;
    int edad;
    char direccion[40];
    int notas[10];
};
struct profesor {
    char nombre [40];
    char direccion[40];
    char asignaturas[10][20];
};
```

```
};
union univ {
    struct alumno al;
    struct profesor prof;
};
```

11.8. Definir tres variables correspondientes a los tres tipos de datos declarados en el ejercicio anterior y asignarles un nombre.

Solución

```
struct alumno a;
struct profesor p;
union univ un;
```

11.9. Escribe una función que devuelva un profesor o un alumno cuyos datos se introducen por teclado.

Análisis del problema

La estructura es devuelta por valor y la función ha de leer por separado cada uno de sus campos, accediendo a ellos por medio del operador punto. Para algunos compiladores no está permitido devolver una estructura por valor. En estos casos habría que devolver un puntero que contenga la dirección de la estructura creada o, aun mejor, pasar la estructura creada a la propia función por referencia y que sea ésta la que modifique los miembros de la estructura, como se hace en el ejercicio siguiente.

Codificación

```
struct alumno leerAlumno( )
{
    struct alumno aa;
    int i;
    printf(" Introduzca el nombre \n");
    scanf("%s", aa.nombre);
    printf(" Introduzca la edad \n");
    scanf("%d", &aa.edad);
    printf(" Introduzca su curso \n");
    scanf("%d", &aa.curso);
    printf(" Introduzca la dirección\n");
    scanf("%s", aa.direccion);
    for (i=0; i<10; i++)
    {
        printf(" Introduzca la nota de la asignatura %d \n", i);
        scanf("%d", &aa.notas[i]);
    }
    return aa;
}
```

11.10. Escribe la misma función que en el ejercicio anterior, 11.9, pero pasando la estructura como argumento a la función.

Análisis del problema

Al pasar la estructura por referencia, por medio de un puntero a la misma, es necesario utilizar el operador flecha para acceder a cada uno de sus campos. La ventaja es que la estructura en este ejemplo no es una variable local, cuyo espacio de memoria puede dar problemas al terminar de ejecutarse la función.

Codificación

```
leerAlumno(struct alumno *aa)
{
    int i;
    printf(" Introduzca el nombre \n");
    scanf("%s", aa->nombre);
    printf(" Introduzca la edad \n");
    scanf("%d", &aa->edad);
    printf(" Introduzca su curso \n");
    scanf("%d", &aa->curso);
    printf(" Introduzca la dirección\n");
    scanf("%s", aa->direccion);
    for (i=0; i<10; i++)
    {
        printf(" Introduzca la nota de la asignatura %d \n", i);
        scanf("%d", &aa->notas[i]);
    }
}
```

11.11. *Escribe una función que tenga como entrada una estructura, profesor o alumno, y escriba sus campos por pantalla.*

Análisis del problema

Es la operación inversa a la entrada. Se trata de acceder a cada uno de los campos de la estructura que se pasa como argumento a la función, puesto que una función estándar como `printf()` no es capaz de mostrar correctamente los campos de una estructura definida por el usuario en un programa; hay que recorrer cada uno de los campos miembro.

Codificación

```
mostrarAlumno(struct alumno *aa)
{
    int i;
    printf(" Nombre: %s\n", aa->nombre );
    printf(" Edad: %d \n", aa->edad);
    printf(" Curso: %d\n", aa->curso);
    printf(" Dirección: %s\n", aa->direccion);
    for (i=0; i<10; i++)
        printf(" Nota de la asignatura %d: %d \n", i, aa->notas[i]);
}
```

11.12. *Escribir un programa de facturación de clientes. Los clientes tienen un nombre, el número de unidades solicitadas, el precio de cada unidad y el estado en que se encuentra: moroso, atrasado, pagado. El programa debe generar los diversos clientes.*

Análisis del problema

Se ha definido una estructura siguiendo la especificación del problema y con la misma se ha declarado un array que va a guardar la información de cada cliente. Es decir, el array de estructuras funciona como una base de datos relacional, en la que cada campo de la estructura corresponde a una columna de la base de datos y cada estructura corresponde a una línea o registro de dicha base.

Codificación

```

struct cliente
{
    char nombre[100];
    int numUnidades;
    float precio;
    char estado;
};

main( )
{
    struct cliente listado [100];
    for (i=0; i<100; i++)
    {
        printf("Introduzca nombre del cliente: ");
        scanf("%s",listado[i].nombre);
        printf("\n Introduzca el número de unidades solicitadas: ");
        scanf("%d",&listado[i].numUnidades);
        printf("\n Introduzca el precio de cada unidad:");
        scanf("%f",&listado[i].precio);
        printf("\n Introduzca el estado del cliente (m\a\p)");
        scanf("%c",&listado[i].estado);
    }
}

```

11.13. *Modifique el programa de facturación de clientes de tal modo que se puedan obtener los siguientes listados.*

- Clientes en estado moroso.
- Clientes en estado pagado con factura mayor de una determinada cantidad.

Análisis del problema

La información requerida está en el array de estructuras que se considera como una tabla de la base de datos. Para realizar el procesamiento del array hay que recorrerlo, por eso la sentencia de C que más se adecua a esta labor es la sentencia `for` con una variable entera de control que recorre todos los posibles valores del índice de la tabla y para cada registro lee y procesa los campos que contienen la información relevante.

Para cumplir los requisitos del programa se supone que el estado del cliente está representado en la base de datos por un carácter que lo simboliza. Así, por ejemplo, si el carácter que aparece es 'm', esto va a significar que el cliente es moroso y si contienen una 'p' son clientes que han pagado ya sus facturas. Se supone también que lo que debe pagar cada cliente en la factura se calcula multiplicando el precio de lo que adquirieron por el número de unidades adquiridas.

Codificación

Se muestran únicamente los dos bucles que presentan el modelo en que ha de procesar el array de estructuras.

```

for (i=0; i<100; i++)
{
    if (listado[i].estado == 'm')
        printf ("%s\n",listado[i].nombre);
}
for (i=0; i<100; i++)
{
    if (listado[i].estado == 'p')

```

```

        if (listado[i].precio * listado[i].numUnidades > maximo)
            printf ("%s\n", listado[i].nombre);
    }

```

11.14. *Escribir un programa que permita hacer las operaciones de suma, resta y multiplicación de números complejos. El tipo complejo ha de definirse como una estructura.*

Análisis del problema

Un número complejo está formado por dos números reales, uno de los cuales se denomina parte real y el otro parte imaginaria. La forma normal de representar en matemáticas un número complejo es la siguiente: $\text{real} + i * \text{imaginario}$. Donde el símbolo i se denomina «unidad imaginaria» y simboliza la raíz cuadrada de -1 . Debido a su naturaleza compuesta, un número complejo se representa de forma natural por una estructura con dos campos de tipo real que contendrán la parte real y la imaginaria del número concreto. Las funciones que siguen, traducen las operaciones matemáticas tal y como se definen en la aritmética de números complejos.

Codificación

```

struct complejo
{
    float r;
    float i;
};

struct complejo suma (struct complejo a, struct complejo b)
{
    struct complejo c;
    c.r = a.r + b.r;
    c.i = a.i + b.i;
    return c;
}

struct complejo resta (struct complejo a, struct complejo b)
{
    struct complejo c;
    c.r = a.r - b.r;
    c.i = a.i - b.i;
    return c;
}

struct complejo multiplicacion (struct complejo a, struct complejo b)
{
    struct complejo c;
    c.r = a.r*b.r - a.i*b.i;
    c.i = a.r*b.i + a.i*b.r;
    return c;
}

```

11.15. *Un número racional se caracteriza por el numerador y denominador. Escriba un programa para operar con números racionales. Las operaciones a definir son la suma, resta, multiplicación y división; además de una función para simplificar cada número racional.*

Análisis del problema

Un número racional posee entonces dos componentes, por lo cual será representado por una estructura con dos campos miembro que corresponderán respectivamente al numerador y al denominador del número. Las operaciones aritméticas se

traducen a la manipulación de sendas estructuras para cada uno de los operandos. Como el resultado debe ser otra estructura se añade un tercer parámetro a las funciones para recibir el resultado. Como el parámetro de salida ha de ser modificado, la única posibilidad es que sea pasado por referencia por medio de un puntero teniendo en cuenta que ha de ser manipulado de forma adecuada con el operador flecha para acceder a sus miembros.

Codificación (Consultar la página web del libro)

11.16. *Se quiere informatizar los resultados obtenidos por los equipos de baloncesto y de fútbol de la localidad alcarreña Lupiana.*

La información de cada equipo:

- *Nombre del equipo.*
- *Número de victorias.*
- *Número de derrotas.*

Para los equipos de baloncesto añadir la información:

- *Número de pérdidas de balón.*
- *Número de rebotes cogidos.*
- *Nombre del mejor anotador de triples.*
- *Número de triples del mejor triplista.*

Para los equipos de fútbol añadir la información:

- *Número de empates.*
- *Número de goles a favor.*
- *Número de goles en contra.*
- *Nombre del goleador del equipo.*
- *Número de goles del goleador.*

Escribir un programa para introducir la información para todos los equipos integrantes en ambas ligas.

Análisis del problema

Cada equipo ha de corresponder a solamente una estructura, de ahí que se defina un array de estructuras para contener la información de cada uno y de todos los equipos. A la hora de leer los datos de cada equipo, así como en la salida de esos datos, hay que tener en cuenta realizar ordenadamente dos movimientos. Por un lado hay que recorrer iterativamente por medio de un bucle las posiciones del array de equipos. Al mismo tiempo y por cada posición del array habrá que acceder a cada uno de los campos de la estructura correspondiente, teniendo en cuenta el tipo de datos que contiene para que la operación de entrada o salida se realice correctamente.

Codificación

```
struct baloncesto
{
    char nombre[20];
    int victorias;
    int derrotas;
    int perdidas;
    int rebotes;
    char mejorTriples [40];
    int triples;
};
struct futbol
{
    char nombre[20];
```

```
    int victorias;
    int derrotas;
    int empates;
    int golesAFavor;
    int golesContra;
    int golesGoleador;
    char goleador[40];
};

main( )
{
    struct baloncesto equiposB [10];
    struct futbol equiposF [10];
    int i;
    for (i=0, i<10; i++) leerEquipoBaloncesto (&equiposB[i]);
    for (i=0, i<10; i++) leerEquipoFutbol (&equiposF[i]);
}

void leerEquipoBaloncesto ( struct baloncesto *bal)
{
    printf ("\n\tIntroducir nombre del equipo :");
    scanf("%s",bal->nombre);
    printf ("\n\tIntroducir el número de victorias conseguidas :");
    scanf ("%d", &bal->victorias);
    printf ("\n\tIntroducir el número de derrotas :");
    scanf ("%d", &bal->derrotas);
    printf ("\n\tIntroducir el número de pérdidas de balón :");
    scanf ("%d", &bal->perdidas);
    printf ("\n\tIntroducir el número de rebotes cogidos :");
    scanf ("%d", &bal->rebotes);
    printf ("\n\tIntroducir nombre del mejor triplista :");
    scanf("%s", bal->mejorTriples);
    printf ("\n\tIntroducir el número de rebotes cogidos :");
    scanf ("%d", &bal->triples);
}

void leerEquipoFutbol ( struct futbol *fut)
{
    printf ("\n\tIntroducir nombre del equipo :");
    scanf("%s", fut->nombre);
    printf ("\n\tIntroducir el número de victorias conseguidas :");
    scanf ("%d", &fut->victorias);
    printf ("\n\tIntroducir el número de derrotas :");
    scanf ("%d", &fut->derrotas);
    printf ("\n\tIntroducir el número de empates :");
    scanf ("%d", &fut->empates);
    printf ("\n\tIntroducir el número de goles a favor :");
    scanf ("%d", &fut->golesAFavor);
    printf ("\n\tIntroducir el número de goles en contra :");
    scanf ("%d", &fut->golesContra);
    printf ("\n\tIntroducir nombre del mejor goleador :");
    scanf("%s", fut->goleador);
    printf ("\n\tIntroducir el número de goles del goleador :");
    scanf ("%d", &fut->golesGoleador);
}
```

11.17. *Modificar el programa 16 para obtener los siguientes informes o datos.*

- *Listado de los mejores triplistas de cada equipo.*
- *Máximo goleador de la liga de fútbol.*
- *Suponiendo que el partido ganado son tres puntos y el empate 1 punto: equipo ganador de la liga de fútbol.*
- *Equipo ganador de la liga de baloncesto.*

Análisis del problema

El procesamiento de la información de un array de estructuras se basa en determinar primero en qué campos de cada registro está la información que se precisa. Una vez determinado esto se trata de recorrer todo el array acumulando en una variable, en este caso, la indicación del registro que contiene el valor buscado. La búsqueda así realizada se denomina lineal, porque recorre la totalidad del array para determinar el elemento que cumple las condiciones predeterminadas.

Se muestra el código de las funciones puesto que las llamadas a las mismas, que habría que realizar dentro del programa solamente necesitan un parámetro y es el nombre del array que contiene los registros de todos los equipos tanto de fútbol, como de baloncesto:

```
mejoresTriplistas (equiposB);
maximoGoleador (equiposF);
equipoGanadorFutbol (equiposF);
equipoGanadorBaloncesto (equiposB);
```

Codificación (Consultar la página web del libro)

11.18. *Un punto en el plano se puede representar mediante una estructura con dos campos. Escribir un programa que realice las siguientes operaciones con puntos en el plano.*

- *Dados dos puntos calcular la distancia entre ellos.*
- *Dados dos puntos determinar el punto medio de la línea que los une.*

Análisis del problema

Las coordenadas de un punto en el plano de dos dimensiones son dos números reales que se representan en forma de estructura. Es decir, por cada punto se define una estructura. Cuando una función retorne como salida una estructura, un método adecuado de hacerlo suele ser proporcionar una estructura para la salida pasándola por referencia al realizar la llamada de la función. Esto es lo que se hace para la función que calcula el punto medio entre dos puntos dados.

Es importante recordar que la función `scanf()`, como cualquier función estándar de entrada y salida de C, no reconoce las estructuras definidas por los programas. Por consiguiente la salida y la entrada de los valores de una estructura ha de realizarse siempre miembro a miembro, teniendo en cuenta exactamente cuál es el tipo de cada uno de ellos y si hay que referenciarlos con el operador punto, porque se accede desde una variable creada desde la definición, o si es necesario usar el operador flecha, porque sólo se posea un puntero a la estructura.

Codificación

```
struct punto
{
    float x;
    float y;
};
main( )
{
    struct punto p, q, r;
    printf ("Introduzca las coordenadas de dos puntos. \n");
    printf ("Primer punto. Coordenada x:");
```

```

scanf ("%f", &p.x);
printf ("\n Primer punto. Coordenada y:");
scanf ("%f", &p.y);
printf ("\n Segundo punto. Coordenada x:");
scanf ("%f", &q.x);
printf ("\n Segundo punto. Coordenada y:");
scanf ("%f", &q.y);
printf ("\n La distancia entre ambos puntos es de %f.\n, distancia (p, q));
medio (p,q, &r);
printf (El punto medio de la recta que une ambos puntos es: (%f, %f). \n", r.x, r.y);
}
float distancia (struct punto p1, struct punto p2)
{
    return (sqrt( sqr( p1.x-p2.x) + sqr( p1.y-p2.y)));
}
medio (struct punto p1, struct punto p2, struct punto * p3)
{
    p3->x = (p1.x + p2.x) /2 ;
    p3->y = (p1.y + p2.y) /2 ;
}

```

11.19. Este programa busca en un archivo una secuencia de bytes. Si los encuentra, los sustituye por ceros. El programa está pensado para ser llamado desde otros programas, por lo que debe devolver su resultado como código de error. Asimismo no debe modificar la longitud total del fichero.

La invocación del programa debe tomar como argumentos el nombre del fichero y una lista de patrones separados por comas, cada uno de los cuales debe ser de la forma siguiente: Puede contener caracteres sueltos que se interpretan literalmente, puede tener números hexadecimales de hasta dos dígitos terminados con el sufijo *h* y puede constar de un número de hasta tres dígitos en decimal seguido por el sufijo *d*.

Por ejemplo:

```
cambiabytes datos.dat 12d 1bh,[,3,x
```

Convierte a cero los códigos *12 1B* y la secuencia de escape «*[3x*» en el fichero *datos.dat*.

Devuelve los siguientes códigos:

0 – Éxito.

1 – No existe el fichero.

2 - Faltan argumentos.

3 – No se puede abrir el fichero.

4 – No hay memoria suficiente.

5 – Argumento no válido.

6 - Error al leer el fichero.

7 - Error al escribir el fichero.

Análisis del problema

Este programa rellena un *array* de estructuras a partir de los argumentos de la línea de órdenes que le indican qué patrones debe buscar en el fichero para ser reemplazados posteriormente. Cada estructura posee un campo para guardar el patrón, de tipo cadena, y su longitud para facilitar su comparación.

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

- 11.1.** Escribir un programa que gestione una agenda de direcciones. Los datos de la agenda se almacenan en memoria en un array de estructuras, cada una de las cuales tiene los siguientes campos:

nombre
dirección
teléfono fijo
teléfono móvil
dirección de correo electrónico

El programa debe permitir añadir una nueva entrada a la agenda, borrar una entrada, buscar por nombre y eliminar una entrada determinada por el nombre.

- 11.2.** Escribir un programa que permita ordenar el array de estructuras definido en el programa anterior por el campo nombre de cada estructura. Utilizar los algoritmos de ordenación por el método de la burbuja, ordenación por inserción y ordenación por selección.

- 11.3.** A menudo, en el tratamiento de bases de datos es necesario unir los datos de dos bases distintas cuyos registros tienen la misma estructura. Para estudiar los aspectos involucrados en tal operación de mezcla de bases de datos, suponga que tiene dos arrays del tipo descrito en el ejercicio propuesto 1 y codifique el programa en C que los una en uno solo, eliminando los duplicados que puedan existir entre los dos.

- 11.4.** Diseñe una estructura de registro para una base de empleados que contenga campos que codifiquen el estado civil del empleado, el sexo y el tipo de contrato utilizando la menor cantidad de memoria posible, es decir, utilizando campos de bits.

- 11.5.** En la base de datos anterior cree un campo de tipo enumerado que permita determinar el departamento al que pertenece un empleado, utilizando un nombre simbólico.

- 11.6.** Escriba un programa para calcular el número de días que hay entre dos fechas; declarar fecha como una estructura.

PROBLEMAS DE PROGRAMACIÓN DE GESTIÓN

- 11.1.** Suponga que tiene un array que guarda la información de los empleados de una gran empresa. De cada empleado se guarda el nombre, los dos apellidos, el número de la Seguridad Social, el NIF, la edad, el departamento en el que trabaja y la antigüedad en la empresa. Escriba un programa en el que se ordene el array por el campo primer apellido y en caso de que el primer apellido coincida por el segundo apellido. Si ambos apellidos coinciden para algún registro ordenar entonces por el nombre.

- 11.2.** Utilizando el array del ejercicio anterior escriba un programa que permita a un usuario por medio de un menú elegir uno de los campos para realizar una búsqueda por dicho campo en el array de registros.

- 11.3.** Escriba un programa auxiliar que permita añadir nuevos campos a la tabla de empleados, como por ejemplo, suel-

do anual y porcentaje de retenciones de impuestos. Una vez modificado el array de estructuras, escriba un programa que permita a un usuario elegir un rango de registros de empleados especificando un apellido inicial y otro final, o un departamento concreto, y produzca en la salida la suma total de los sueldos que se les pagan a los empleados seleccionados.

- 11.4.** Escribir un programa que permita elaborar un informe a partir del array de estructuras anterior con el siguiente formato. Cada página contendrá los empleados de un solo departamento. Al comienzo de cada página se indicará por medio de una cabecera cada uno de los campos que se listan y al departamento que corresponde el listado. Los campos aparecen justificados a la derecha en cada columna.

Punteros (apuntadores)

Los punteros en C tienen fama, en el mundo de la programación, de dificultad, tanto en el aprendizaje como en su uso. En este capítulo se tratará de mostrar que los punteros no son más difíciles de aprender que cualquier otra herramienta de programación ya examinada o por examinar a lo largo de este libro. El *puntero*, no es más que una herramienta muy potente que puede utilizar en sus programas para hacerlos más eficientes y flexibles. Los punteros son, sin género de dudas, una de las razones fundamentales para que el lenguaje C sea tan potente y tan utilizado.

Una *variable puntero* (o *puntero*, como se llama normalmente) es una variable que contiene direcciones de otras variables. Todas las variables vistas hasta este momento contienen valores de datos; por el contrario las variables puntero contienen valores que son direcciones de memoria donde se almacenan datos. En resumen, un puntero es una variable que contiene una dirección de memoria, y utilizando punteros su programa puede realizar muchas tareas que no serían posibles utilizando tipos de datos estándar.

En este capítulo se estudiarán los diferentes aspectos de los punteros:

- Concepto y características
- Utilización de punteros.
- Asignación dinámica de memoria.
- Aritmética de punteros.
- *Arrays* de punteros.
- Punteros a punteros, funciones y estructuras.

12.1 Concepto de puntero (apuntador)

Cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su nombre, su tipo y su dirección en memoria. Al valor de una variable se accede por medio de su nombre. A la dirección de la variable se accede mediante el operador de dirección `&`. Un puntero es una variable que contiene una dirección de una posición de memoria que puede corresponder o no a una variable declarada en el programa. La declaración de una variable puntero debe indicar el tipo de dato al que apunta; para ello se hace preceder a su nombre con un asterisco (`*`):

```
<tipo de dato apuntado> *<identificador de puntero>
```

EJEMPLO 12.1 *Inicialización de punteros*

C no inicializa los punteros cuando se declaran y es preciso inicializarlos antes de su uso. Después de la inicialización, se puede utilizar el puntero para referenciar los datos direccionados. Para asignar una dirección de memoria a un puntero se utiliza el operador `&`. Este método de inicialización, denominado *estático*, requiere:

Asignar memoria estáticamente definiendo una variable y a continuación hacer que el puntero apunte al valor de la variable.

```
int i; /*define una variable i*/
int *p; /*define un puntero a un entero p*/
p = &i; /*asigna la dirección de i a p */
```

El operador `&` devuelve la dirección de la variable a la cual se aplica.

Es un error asignar un valor a una variable puntero si previamente no se ha inicializado con la dirección de una variable o se le ha asignado memoria dinámicamente. El uso de un puntero para obtener el valor al que apunta, es decir, su dato apuntado se denomina *indireccionar el puntero* (“desreferenciar el puntero”); para ello, se utiliza el operador de indirección `*`. La Tabla 12.1 resume los operadores de punteros.

Tabla 12.1 Operadores de punteros

Operador	Propósito
<code>&</code>	Obtiene la dirección de una variable.
<code>*</code>	Define una variable como puntero.
<code>*</code>	Obtiene el contenido de una variable puntero.

12.2 Punteros NULL y VOID

Un *puntero nulo* no apunta a ningún dato válido; se utiliza para proporcionar a un programa un medio de conocer cuándo una variable puntero no direcciona a un dato válido. Para declarar un puntero nulo se utiliza la macro `NULL`.

Nota:

- Un puntero nulo no direcciona ningún dato válido. Un puntero `void` direcciona datos de un tipo no especificado. Un puntero `void` se puede igualar a nulo si no se direcciona ningún dato válido. `NULL` es un valor; `void` es un tipo de dato.
- Un puntero puede apuntar a otra variable puntero. Para declarar un puntero a un puntero se hace preceder a la variable con dos asteriscos (`**`).

12.3 Punteros y arrays

Los arrays y los punteros están fuertemente relacionados en el lenguaje C. El nombre de un array es un puntero, contiene la dirección en memoria de comienzo de la secuencia de elementos que forma el array. Es un puntero constante ya que no se puede modificar, sólo se puede acceder para indexar a los elementos del array. Si se tiene la siguiente declaración de un array, la Figura 12.1 representa un array almacenado en memoria.

```
int lista[5] = {10, 20, 30, 40, 50};
```

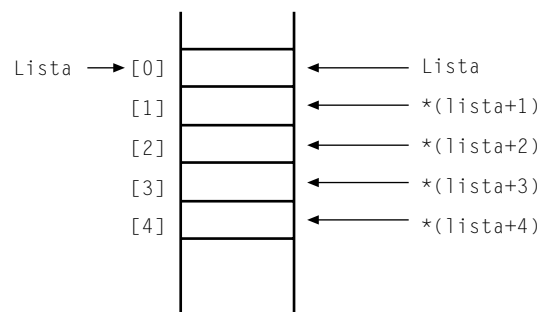


Figura 12.1 Un array almacenado en memoria.

EJEMPLO 12.2. Acceso a arrays mediante punteros

Se puede utilizar indistintamente notación de subíndices o notación de punteros. Dado que un nombre de un array contiene la dirección del primer elemento del array, se debe indireccionar el puntero para obtener el valor del elemento. En este ejemplo se ponen de manifiesto operaciones correctas y erróneas con nombres de array.

```
float v[10];
float *p;
float x = 100.5;
int j;

/* se indexa a partir de v */
for (j= 0; j<10; j++)
    *(v+j) = j*10.0;
p = v+4; /* se asigna la dirección del quinto elemento */
v = &x; /* error: intento de modificar un puntero constante */
```

Se puede declarar un *array de punteros*, como un array que contiene como elementos punteros, cada uno de los cuales apuntará a otro dato específico. La línea siguiente reserva un array de diez variables punteros a enteros:

```
int *ptr[10];
```

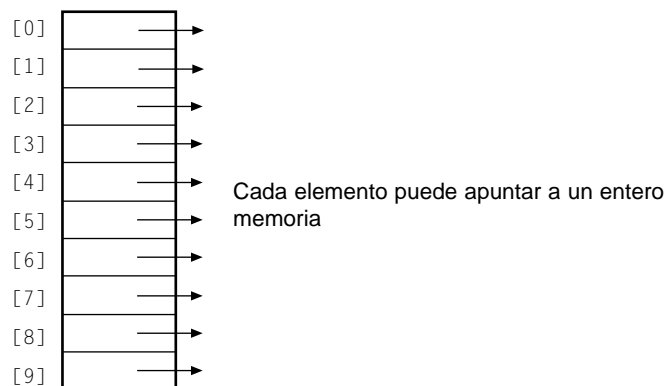


Figura 12.2 Un array de 10 punteros a enteros.

La Figura 12.2 muestra cómo C organiza este array. Cada elemento contiene una dirección que *apunta* a otros valores de la memoria. Cada valor apuntado debe ser un entero. Se puede asignar a un elemento de `ptr` una dirección, tal como para variables puntero o arrays. Así por ejemplo,

```
ptr[5] = &edad; /* ptr[5] apunta a la dirección de edad */
ptr[4] = NULL; /* ptr[4] no contiene dirección alguna */
```


De igual forma, se podría declarar un puntero a un array de punteros a enteros.

```
int *(*ptr10)[];
```

paso a paso:

<code>(*ptr10)</code>	es un puntero; ptr10 es un nombre de variable.
<code>(*ptr10)[]</code>	es un puntero a un array
<code>*(ptr10)[]</code>	es un puntero a un array de punteros
<code>int *(*ptr10)[]</code>	es un puntero a un array de punteros de variables int

Una matriz de números enteros, o reales, puede verse como un array de punteros; de tantos elementos como filas tenga la matriz, apuntando cada elemento del array a un array de enteros, reales, de tantos elementos como columnas.

La inicialización de un array de punteros a cadenas se puede realizar con una declaración similar a ésta:

```
char *nombres_meses[12] = { "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
                           "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre" };
```

EJEMPLO 12.3 *Uso de los punteros a cadenas*

Considérese la siguiente declaración de un array de caracteres que contiene las veintiséis letras del alfabeto internacional.

```
char alfabeto[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Si `p` es un puntero a `char`. Se establece que `p` apunta al primer carácter de `alfabeto` escribiendo

```
p = &alfabeto[0];          /* o también p = alfabeto */
```

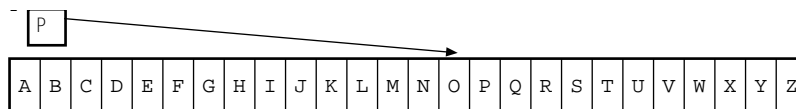


Figura 12.3 Un puntero `alfabeto` [15].

Es posible, entonces, considerar dos tipos de definiciones de cadena:

```
char cadena1[]="Hola viejo mundo";
/*array contiene una cadena */
char *cptr = "C a su alcance";
/*puntero a cadena, el sistema reserva memoria para la cadena*/
```

12.4 Aritmética de punteros

A un puntero se le puede sumar o restar un entero n ; esto hace que apunte n posiciones adelante o atrás de la actual. A una variable puntero se le puede aplicar el operador `++`, o el operador `--`. Esto hace que contenga la dirección del siguiente, o anterior elemento. No tiene sentido, por ejemplo, sumar o restar una constante de coma flotante.

Operaciones no válidas con punteros:

- No se pueden sumar dos punteros.
- No se pueden multiplicar dos punteros.
- No se pueden dividir dos punteros.

Para crear un puntero constante se debe utilizar el siguiente formato:

```
<tipo de dato> *const <nombre puntero> = <dirección de variable>;
```

El formato para definir un puntero a una constante es:

```
const <tipo de dato elemento> *<nombre puntero> = <dirección de constante>;
```

Cualquier intento de cambiar el contenido almacenado en la posición de memoria a donde apunta creará un error de compilación.

Nota:

Una definición de un puntero constante tiene la palabra reservada `const` delante del nombre del puntero, mientras que el puntero a una definición constante requiere que la palabra reservada `const` se sitúe antes del tipo de dato. Así, la definición en el primer caso se puede leer como «punteros constante o de constante», mientras que en el segundo caso la definición se lee «puntero a tipo constante de dato».

El último caso a considerar es crear punteros constantes a constantes utilizando el formato siguiente:

```
const <tipo de dato elemento> *const <nombre puntero> = <dirección de constante>;
```

Regla:

- Si conoce que un puntero siempre apuntará a la misma posición y nunca necesita ser reubicado (recolocado), defínalo como un puntero constante.
- Si conoce que el dato apuntado por el puntero nunca necesitará cambiar, defina el puntero como un puntero a una constante.

12.5 Punteros como argumentos de funciones

Cuando se pasa una variable a una función (*paso por valor*) no se puede cambiar el valor de esa variable. Sin embargo, si se pasa un puntero a una función (*paso por dirección*) se puede cambiar el valor de la variable a la que el puntero apunte.

El paso de un nombre de array a una función es lo mismo que pasar un puntero al array. Se pueden cambiar cualquiera de los elementos del array. Sin embargo, cuando se pasa un elemento a una función, el elemento se pasa por valor.

EJEMPLO 12.4 Paso de argumentos por referencia

Hay que recordar que el paso de los parámetros en las llamadas a las funciones en C siempre se hace por valor. Si se precisa pasar una variable por referencia, se pasará un puntero a dicha variable, como por ejemplo en la función que intercambia el valor de los variables:

```
intercambia (int *a, int *b)
{
    int aux;

    aux = *a;
    *a = *b;
    *b = aux;
}
```

12.6 Punteros a funciones

Es posible también crear punteros que apunten a funciones. En lugar de direccionar datos, los punteros de funciones apunten a código ejecutable. Al igual que los datos, las funciones se almacenan en memoria y tienen direcciones iniciales. Tales funciones se pueden llamar de un modo indirecto, es decir, mediante un puntero cuyo valor es igual a la dirección inicial de la función en cuestión.

La sintaxis general para la declaración de un puntero a una función es:

```
Tipo_de_retorno (*PunteroFuncion) (<lista de parámetros>);
```

Este formato indica al compilador que *PunteroFuncion* es un puntero a una función que devuelve el tipo *Tipo_de_retorno* y tiene una lista de parámetros.

Un puntero a una función es simplemente un puntero cuyo valor es la dirección del nombre de la función. Dado que el nombre es, en sí mismo, un puntero, un puntero a una función es un puntero a un puntero constante.

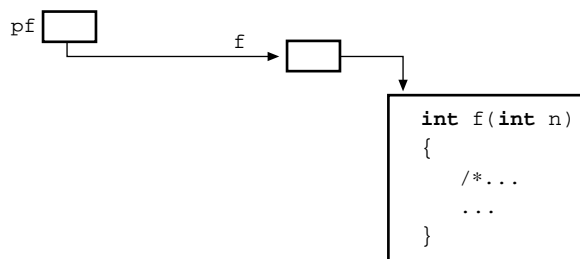


Figura 12.4 Puntero a función.

La función asignada debe tener el mismo tipo de retorno y lista de parámetros que el puntero a función; en caso contrario, se producirá un error de compilación. *Los punteros a funciones también permiten pasar una función como un argumento a otra función.* Para pasar el nombre de una función como un argumento función, se especifica el nombre de la función como argumento.

Recuerde:

var, nombre de una variable.
 var[] es un array.
 (*var[]) es un array de punteros.
 (*var[])() es un array de punteros a funciones.
 int (*var[])() es un array de punteros a funciones que devuelven valores de tipo **int**.

EJEMPLO 12.5. Uso de punteros a estructuras

```

struct punto {
    float x;
    float y;
} p, *puntp;

printf ("Introduzca las coordenadas: ");
scanf ("%f", &p.x);
scanf ("%f", &p.y);
puntp = &p;
printf ("Las coordenadas introducidas son (%f, %f)", puntp->x, puntp->y);
  
```

Se puede declarar un puntero a una estructura tal como se declara un puntero a cualquier otro objeto. Cuando se referencia una estructura utilizando el puntero estructura, se emplea el operador `->` para acceder a un miembro de ella.

PROBLEMAS RESUELTOS

12.1. Encuentre los errores en la siguiente declaración de punteros:

```
int x, *p, &y;  
char* b= "Cadena larga";  
char* c= 'C';  
float x;  
void* r = &x;
```

Es incorrecta sintácticamente la declaración `int &y`. No tiene ningún sentido en C.

Cuando un carácter está rodeado por comillas simples es considerado como una constante de tipo `char` no como una cadena, para lo cual debería estar rodeado de dobles comillas:

```
char* c= "C";
```

No se puede declarar un puntero a tipo `void`.

12.2. Dada la siguiente declaración, escribir una función que tenga como argumento un puntero al tipo de dato y muestre por pantalla los campos.

```
struct boton  
{  
    char* rotulo;  
    int codigo;  
};
```

La función puede ser la siguiente:

```
void mostrarBoton (struct boton *pb)  
{  
    printf ("Rotulo del botón : %s\n", pb->rotulo);  
    printf ("Código asociado al boton : %d\n", pb->codigo);  
}
```

12.3. ¿Qué diferencias se pueden encontrar entre un puntero a constante y una constante puntero?

Por medio de un puntero a constante se puede acceder a la constante apuntada, pero obviamente no está permitido modificar su valor por medio del puntero. Un puntero declarado como constante no puede modificarse su valor, es decir la dirección que contiene y a la que apunta.

12.4. Un array unidimensional se puede indexar con la aritmética de punteros. ¿Qué tipo de puntero habría que definir para indexar un array bidimensional?

El tipo de puntero que se vaya a utilizar para recorrer un array unidimensional tendrá que ser el mismo tipo que el de los elementos que compongan el array, puesto que va a ir apuntando a cada uno de ellos según los recorra. Para apuntar al array bidimensional como tal, o lo que es lo mismo, para apuntar a su inicio, el compilador de C considera que un array bidimensional es en realidad un array de punteros a los arrays que forman sus filas. Por tanto, será necesario un puntero doble o puntero a puntero, que contendrá la dirección del primer puntero del array de punteros a cada una de las filas del array bidimensional o matriz.

Una array bidimensional se guarda en memoria linealmente, porque la memoria es lineal, fila a fila. Por consiguiente para acceder a un elemento concreto de una fila y columna determinadas habrá que calcular primero en qué fila está y dentro de

esa fila según su columna se calculará su posición dentro de la memoria. Para realizar esta operación no es necesario saber cuántas filas contiene la matriz bidimensional pero sí cuántas columnas, para saber cuántos bytes ocupa cada fila. Esto se verá en los ejercicios siguientes.

12.5. *En el siguiente código se accede a los elementos de una matriz. Acceder a los mismos elementos con aritmética de punteros.*

```
#define N 4
#define M 5
int f,c;
double mt[N][M];

. . .
for (f = 0; f<N; f++)
{
    for (c = 0; c<M; c++)
        printf("%lf ", mt[f][c]);
    printf("\n");
}
```

Análisis del problema

Se define un puntero que apunte a la primera posición de la matriz y se calcula la posición de memoria donde se van encontrando cada uno de los elementos de la matriz, a base de sumar la longitud de las filas desde el comienzo de la matriz y los elementos desde el comienzo de la fila donde está situado el elemento al que se desea acceder. Si un elemento está en la fila 5, habrá que saltar 5 filas enteras de elementos del tipo de la matriz, para situarse al comienzo de su fila en la memoria. Recordar que en C los arrays siempre se numeran desde 0. Si el elemento que se busca está en la columna 3, hay que calcular tres posiciones desde el comienzo de su fila calculado antes. Así se llega a la dirección donde se encuentra el elemento buscado. Para hacer estos cálculos es imprescindible conocer el número de columnas de la matriz, que es igual al tamaño de cada fila. Sin este dato sería imposible reconstruir la estructura de la matriz, partiendo sólo del espacio que ocupa en la memoria, ya que éste es puramente lineal. Una vez que se tiene dicha dirección se accede a su contenido. Esta expresión es la misma que sustituye el compilador de C, cuando compila la indirección que representan los operadores corchetes de los arrays.

Codificación

```
#define N 4
#define M 5
int f,c;
double mt[N][M], *pmt=mt;

. . .
for (f = 0; f<N; f++)
{
    for (c = 0; c<M; c++)
        printf("%lf ", *(pmt + f*M + c));
    printf("\n");
}
```

Otra opción podría haber sido hacer que el puntero recorra la matriz, sabiendo que la matriz está almacenada en memoria fila a fila de forma lineal.

```
#define N 4
#define M 5
int f,c;
double mt[N][M], *pmt=mt;

. . .
for (f = 0; f<N; f++)
{
```

```

    for (c = 0; c<M; c++)
        printf("%lf ", *pmt++);
    printf("\n");
}

```

- 12.6.** *Escriba una función con un argumento de tipo puntero a `double` y otro argumento de tipo `int`. El primer argumento se debe corresponder con un array y el segundo con el número de elementos del array. La función ha de ser de tipo puntero a `double` para devolver la dirección del elemento menor.*

Análisis del problema

Al hacer la llamada a la función se pasa el nombre del array a recorrer o un puntero con la dirección del comienzo del array. De la misma manera el puntero que recibe la función puede ser tratado dentro de su código como si fuera un array, usando el operador corchete, o como un simple puntero que se va a mover por la memoria. En ambos casos el segundo parámetro de la función es imprescindible para no acceder fuera de la región de la memoria donde están los datos válidos del array original almacenados.

Codificación

```

double *menorArray (double *v, int n)
{
    int i, min = -1;
    /* suponer que los elementos del array v son positivos */
    double *menor;
    for (i=0; i<n; i++)
        if (v[i] < min) menor = &v[i];
    /* o if (*v++ < min) menor = v-1; */
    return menor;
}

```

- 12.7.** *¿Qué diferencias se pueden encontrar entre estas dos declaraciones?*

```

float mt[5][5];
float *m[5];

```

Análisis del problema

En la primera declaración se reserva memoria para una matriz de 25 números reales. En la segunda, sólo se reserva memoria para un array de cinco punteros a reales.

¿Se podrían hacer estas asignaciones?:

```

m = mt;
m[1] = mt[1];
m[2] = &mt[2][0];

```

Ambas variables son del mismo tipo, pero son consideradas como constantes por ser nombres de arrays, por lo cual no se puede modificar su contenido. La segunda asignación es correcta porque el compilador interpreta la expresión `mt[1]` como conteniendo la dirección de la segunda fila de la matriz y por lo tanto su valor es del mismo tipo que `m[1]` en el lado de la izquierda. La expresión de la derecha en la tercera asignación proporciona la dirección de la tercera fila de la matriz; por consiguiente, es también de tipo puntero a real al igual que el lado derecho de la matriz.

- 12.8.** *Dadas las siguientes declaraciones de estructuras, escribe cómo acceder al campo `x` de la variable estructura `t`.*

```

struct fecha
{
    int d, m, a;
    float x;
};
struct dato
{
    char* mes;
    struct fecha* f;
} t;

```

Análisis del problema

La variable `t` es una variable de tipo estructura, por lo que se usa el operador punto para acceder al miembro `f`. Desde el dato miembro `f` es necesario usar el operador flecha para acceder al campo `x` de la estructura `fecha` a la que apunta.

```
t.f->x;
```

¿Qué problemas habría en la siguiente sentencia?

```
gets(t.mes);
```

El campo `mes` de la estructura `fecha` no apunta a ningún sitio, por lo cual dará problemas de asignación de memoria cuando la función `gets()` intente colocar el resultado en el puntero que se le pasa como argumento. Para evitar esto sería necesario reservar memoria antes de llamar a `gets()`.

12.9. El prototipo de una función es

```
void escribe_mat(int** t, int nf, int nc);
```

La función tiene como propósito mostrar por pantalla la matriz. El primer argumento se corresponde con una matriz entera, el segundo y tercero es el número de filas y columnas de la matriz. Escriba la implementación de la función aplicando la aritmética de punteros.

```

void escribe_mat(int** t, int nf, int nc)
{
    int f,c;
    for (f=0; f<nf; f++)
        for (c=0; c<nc; c++)
            printf("Elemento de la fila %d y columna %d: %d\n", f, c, *(t + f * nc + c));
}

```

12.10. Escriba un programa en el que se lean 20 líneas de texto, cada línea con un máximo de 80 caracteres. Mostrar por pantalla el número de vocales que tiene cada línea.

Análisis del problema

Una forma de almacenar texto compuesto de líneas que son cadenas de caracteres consiste en partir de un puntero a cadenas, es decir de un puntero de indirección doble que apunte a un array de punteros a cadenas de caracteres. Ese puntero, que el programa denomina `texto`, apuntará a un array de punteros a cadenas que apuntarán cada uno de ellos a su vez a cada una de las líneas de texto que el usuario vaya introduciendo por teclado. Como cada cadena es un array de caracteres, cada línea funcionará como si fuera la fila de una matriz bidimensional, con la ventaja de que cada una tendrá sólo la longitud necesaria

para almacenar su contenido, sin que se desperdicie espacio de memoria sin usar como ocurriría si se hubiese definido una matriz con todas las filas de la misma longitud.

Codificación

ENTRADA: 20 cadenas de caracteres
SALIDA: Número de vocales que contiene cada línea.

Para ajustar el tamaño de la memoria que ocupa cada cadena, en vez de reservar directamente el tamaño máximo para cada línea, se lee en un array cada línea desde teclado y después se usan las funciones de manejo de cadenas para reservar espacio para los caracteres leídos y para copiar la línea leída en ellos.

Para comprobar el número de vocales se accede a cada uno de los caracteres de la línea leída y se comparan por medio de una sentencia `switch` con los caracteres correspondientes a las vocales, tanto mayúsculas como minúsculas.

```
main( )
{
    char **texto, *línea;
    int i, vocales;
    texto = (char **) malloc (20 * sizeof (char *));
    línea = (char *) malloc (80 * sizeof (char));
    for (i=0; i<20; i++)
    {
        gets(línea);
        texto [i] = (char *) malloc (strlen (línea)+1);
        strcpy (texto[i], línea);
        /* Ahora comprobar las vocales que contiene la línea */
        vocales = 0;
        for (i=0; i<strlen(línea); i++)
        {
            switch (línea [i])
            {
                case 'a': case 'A':
                case 'e': case 'E':
                case 'i': case 'I':
                case 'o': case 'O':
                case 'u': case 'U': vocales++;
            }
        }
        printf ("La línea contiene %d vocales. \n", vocales);
    } /* for interno */
} /* for externo */
```

- 12.11.** *Escribir un programa que encuentre una matriz de números reales simétrica. Para ello, una función entera con entrada la matriz determinará si ésta es simétrica. En otra función se generará la matriz con números aleatorios de 1 a 112. Utilizar la aritmética de punteros en la primera función, en la segunda indexación.*

Análisis del problema

Una matriz es simétrica si cada uno de sus elementos es igual a su simétrico con respecto a la diagonal principal. Una manera de comprobarlo es recorrer la matriz y preguntar si cada elemento es idéntico al que se obtiene cambiando su fila por su columna y su columna por su fila.

Codificación

ENTRADA: Matriz generada aleatoriamente

SALIDA: Un valor entero que es cero si la matriz no es simétrica, uno si lo es.

Para recorrer una matriz utilizando punteros hay que recordar siempre que la matriz está almacenada linealmente en la memoria línea a línea. Por consiguiente para situarse en un elemento concreto posicionarse primero en la fila correspondiente sumando a la dirección del primer elemento el número de filas a saltar y para calcular la dirección en la fila requerida, sumar tantas unidades al puntero como columnas haya que saltar para llegar. No hay que preocuparse de cuántos bytes hay que ir sumando cada vez a la dirección puesto que eso lo realiza el compilador de forma automática a partir de la información del tipo de datos a los que apunta el puntero. En este problema a reales (float).

```
main ( )
{
    float matriz [10][10];
    crearMatriz (matriz);
    if ((resp = simetrica (matriz))==1)
        printf ("La matriz generada es simétrica. \n");
    else
        printf ("La matriz generada no es simétrica. \n");
}

crearMatriz ( float m[][10])
{
    int i,j;
    randomize( );
    for (i=0; i<10; i++)
        for (j=0; j<10; j++)
            m[i][j] = rand ( ) % 10;
}

int simetrica (float **mat)
{
    int i,j, resp = 1;
    for (i=0; i<10; i++)
        for (j=0; j<10; j++)
            if ( i!= j)
                if ( *(mat + 10 *i + j) != *(mat + 10 *j +i))
                {
                    resp =0;
                    return (resp);
                }
    return (resp);
}
```

12.12. *En una competición de natación se presentan 16 nadadores. Cada nadador se caracteriza por su nombre, edad, prueba en la que participa y tiempo(minutos, segundos) de la prueba. Escribir un programa que realice la entrada de los datos y calcule la desviación estándar respecto al tiempo. Para la entrada de datos, definir una función que lea del dispositivo estándar el nombre, edad, prueba y tiempo.*

Análisis del problema

El primer paso es siempre definir una estructura con los datos que se necesitan para cada elemento. Para que haya espacio para varios participantes, se crea también un array de estructuras del tipo definido. Aunque de antemano se sabe el número de participantes se trabajará con un array de punteros a las estructuras. Así lo único que se crea en principio es un array de punteros sin inicializar. Según se vayan leyendo los datos se irá reservando memoria para cada estructura y haciendo que el puntero correspondiente del array apunte a ellas.

Otro uso común consiste en que cuando en una estructura un miembro es de tipo cadena, en vez de definirlo con una longitud determinada, se declara como puntero a carácter, pues de esta manera sólo se reserva la memoria que se necesita una vez que se sepa qué cadena se ha de introducir en dicho campo. El ahorro en espacio de memoria es significativo cuando el número de registros, en este caso estructuras, es grande.

Codificación

```
struct nadador
{
    char * nombre;
    int edad;
    char *prueba;
    struct tt
    {
        int min;
        int seg;
    } tiempo;
};

main( )
{
    struct nadador *participantes[16];
    leerParticipantes (participantes);
    desviacionEstandar (participantes);
}

leerParticipantes ( struct nadador *participantes[])
{
    int i;
    for (i=0; i<16; i++)
        leerNadador(participantes, i);
}

leerNadador (struct nadador ** p, int num)
{
    char cadena[40];
    *(p+i) = (struct nadador *) malloc (sizeof (struct nadador));
    printf ("\n Introduzca el nombre : ");
    gets(cadena);
    p[i]->nombre = (char*) malloc (strlen(cadena)+1);
    strcpy (p[i]->nombre, cadena);
    printf ("\n Introduzca la edad : ");
    scanf ("%d", &p[i]->edad);
    printf ("\n Introduzca la prueba en la que participa : ");
    gets(cadena);
    p[i]->prueba = (char*) malloc (strlen(cadena)+1);
    strcpy (p[i]->prueba, cadena);
    printf ("\n Introduzca el tiempo de la prueba. Minutos y segundos : ");
    scanf ("%d %d", &p[i]->tiempo.min, &p[i]->tiempo.seg);
}

desviacionEstandar (struct nadador *participantes[])
{
    float media, desv, d;
    int i;
    for (i=0; i<16; i++)
        media += participantes[i]->tiempo.min * 60 +
```

```

        participantes[i]->tiempo.seg;
media /= 16;
for (i=0; i<16; i++)
{
    d = participantes[i]->tiempo.min * 60 +
        participantes[i]->tiempo.seg - media;
    desv += d * d;
}
desv /= 16;
printf ("La desviación estándar es: %f/n", sqrt(desv));
}

```

12.13. *Se quiere evaluar las funciones $f(x)$, $g(x)$ y $z(x)$ para todos los valores de x en el intervalo $0 \leq x < 3.5$ con incremento de 0.2. Escribir un programa que evalúe dichas funciones. Utilizar un array de punteros a función. Las funciones son las siguientes:*

$$f(x) = 3e^x - 2x$$

$$g(x) = -x \sin(x) + 1.5$$

$$z(x) = x^3 - 2x + 1$$

Análisis del problema

Una vez que se entiende el mecanismo de los punteros a funciones su uso es muy sencillo. Se trata de definir las funciones al estilo de C y después definir un array de punteros, a los que se asignan cada una de las funciones creadas. El acceso a dichas funciones para su ejecución es similar a cuando se accede a cualquier otro dato por medio de punteros.

Codificación

```

float f (float x)
{
    return (3 * exp(x) - 2*x);
}
float g (float x)
{
    return (-x * sin (x) + 1.5);
}
float z (float x)
{
    return ( x *x * x - 2 *x + 1);
}
main ( )
{
    float (*func[3]) (float);
    func [0] = f;
    func [1] = g;
    func [2] = z;
    for ( i=0; i <=2; i++)
        for ( x = 0.0; x < 3.5;
            x += 0.2)
            printf ( " x = %f = %f \n", x, (*func [i])(x) );
}

```

12.14. *Se quiere sumar enteros largos, con un número de dígitos que supera el máximo entero largo. Los enteros tienen un máximo de 40 dígitos. Para solventar el problema se utilizan cadenas de caracteres para guardar cada entero y realizar la suma. Escribir un programa que lea dos enteros largos y realice la suma.*

Análisis del problema

Como indica el enunciado el programa recoge los dos números como cadenas. De esta manera sus dígitos se guardan según su código ASCII y no según su valor numérico. No es ningún problema la conversión entre códigos ASCII y valores numéricos dado que C permite operar con caracteres y enteros convirtiéndolos unos en otros según se necesite.

Una vez que se tienen las dos cadenas hay que fijarse en que los dígitos menos significativos, que son por los que se empieza a sumar, están al final de las cadenas, por lo cual o se da la vuelta a las cadenas (procedimiento que se verá en un capítulo posterior) o, como en este caso, usar dos punteros para recorrer las cadenas con los números desde atrás adelante.

Codificación

```
main( )
{
    char num1[40], num2[40], res[40];
    char *pnum1, *pnum2;
    printf ("Introduzca el primer sumando: ");
    gets (num1);
    printf ("\n Introduzca el segundo sumando: ");
    gets (num2);
    long1 = strlen (num1);
    long2 = strlen (num2);
    pnum1 = num1[long1];
    /* los punteros apuntan al final de cada cadena que es el primer dígito de número */
    pnum2 = num2[long2];
    i = long1 > long2 ? long1 : long2; /* longitud cadena más larga */
    res[i+1] = '\0';
    do
    {
        suma = res[i] + (*pnum1-- - '0') + (*pnum2-- - '0');
        if (suma < 9)
            res[i--] = suma + '0';
        else
        {
            res[i--] = (suma % 10) + '0'; /* hay arrastre */
            res[i] ++;
        }
    }
    /* las conversiones de char a int y viceversa suponen -'0'-'0'+ '0' */
    } while (pnum1 >= num1 && pnum2 >= num2);
    /* Hasta que se acabe el número con menos dígitos. */
    /* Queda sumar el resto del número más largo. */
    if (pnum1 > num1)
        do
        {
            res[i--] += num1[i]
        } while (i);
    else if (pnum2 > num2)
        do
        {
            res[i--] += num2[i]
```

```

    } while (i);
    printf ( "\nEl resultado es : %s\n", res);
}

```

12.15. *Escribir una función que tenga como entrada una cadena y devuelva un número real. La cadena contiene los caracteres de un número real en formato decimal (por ejemplo, la cadena «25.56» se ha de convertir en el correspondiente valor real).*

Análisis del problema

Fundamentalmente se trata de ir recorriendo la cadena, utilizando un puntero, para calcular primero la parte entera del número hasta encontrar el punto decimal y luego seguir por los dígitos después del punto para calcular los decimales del número real.

Codificación

```

float cadenaanum (char * cadena)
{
    char *pcad; int d = 1, signo;
    float n;
    /* saltar espacios en blanco iniciales */
    for (pcad = cadena [0]; *pcad == ' '; pcad++);
    /* averiguar el signo del número */
    switch (*pcad)
    {
        case '-': signo = -1; pcad++; break;
        case '+': pcad++;
        default : signo = +1;
    }
    /* calcular la parte entera del número */
    n = *pcad - '0'; /* convierte el dígito en decimal */
    while (*++pcad >='0' && *pcad <= '9')
        /* ¿el siguiente carácter es un dígito? */
        n = n*10 + *pcad - '0';
    /* convertir y añadir a parte entera del número */
    if (*pcad++ != '.' )
        return (signo * n);
    /* el número sólo tiene parte entera así que devolver */
    while ((*pcad >='0' && *pcad <= '9')
    {
        d *= 10;
        n += (*pcad++ - '0') / d;
    }
    return (signo * n);
}

```

12.16. *Escribir un programa para generar una matriz de 4 x 5 elementos reales, multiplicar la primera columna por cualquier otra y mostrar la suma de los productos. El programa debe descomponerse en subproblemas y utilizar punteros para acceder a los elementos de la matriz.*

```

main( )
{
    float **mat, *productos;
    randomize( );

```

```

    generarmatriz (mat);
    multiplicacolumnas (mat, rand( ) % n, productos);
    mostrarsumaproductos (productos);
}
generarmatriz (float **m)
{
    m = (float **) malloc (4*5 * sizeof (float));
    randomize( );
    for (i = 0; i <4; i++)
        for (j = 0; j <5; j++)
            *(m + i*5 + j) = rand( ) + rand( )/ 1000 ;
}
multiplicacolumnas (float **m, int ncol, float *vp)
{
    vp = (float *) malloc(5 * sizeof (float));
    for (i = 0; i <4; i++)
        *vp++ = *(m + i*5) * *(m + i*5 + ncol);
        /* vp[i] = m[i][0] * m[i][ncol]; */
}
mostrarsumaproductos ( float *v)
{
    int suma = 0;
    for (i = 0; i < 4; i++) suma += *v++;
    printf ("%f \n", suma);
}

```

12.17. *Desarrolle un programa en C que use una estructura para la siguiente información sobre un paciente de un hospital: nombre, dirección, fecha de nacimiento, sexo, día de visita y problema médico. El programa debe tener una función para entrada de los datos de un paciente, guardar los diversos pacientes en un array y mostrar los pacientes cuyo día de visita sea uno determinado.*

```

#define TAM 200
struct paciente
{
    char *nombre;
    char *direccion;
    char fecha [10];
    char sexo; /* V o H */
    char diavisita [10];
    char *problema;
};
main ( )
{
    char dia [10];
    struct paciente *lista;
    tomadatos(lista);
    printf ("Introduzca la fecha de la consulta: (dd/mm/aaaa) ");
    gets(dia);
    mostrarconsulta (dia, lista);
}
tomadatos(struct paciente *lista)
{
    int i;

```

```

char buffer [80];
lista = (struct paciente*) malloc(TAM*sizeof(struct paciente));
printf ("  Entrada de datos de los pacientes\n");
printf("===== \n");
for (i = 0; i < TAM; i++)
{
    printf ("Nombre del paciente :\n");
    gets (buffer);
    lista[i]->nombre = malloc (strlen (buffer) +1);
    strcpy (lista[i]->nombre, buffer);
    printf ("Direccion del paciente :\n");
    gets (buffer);
    lista[i]->direccion = malloc (strlen (buffer) +1);
    strcpy (lista[i]->direccion, buffer);
    printf ("Fecha de nacimiento  (dd/mm/aaaa) \n");
    gets(lista[i]->fecha);
    printf ("Sexo del paciente\n");
    lista[i]->sexo = getche( );
    printf ("Dia de visita  (dd/mm/aaaa) \n");
    gets(lista[i]->diavisita);
    printf ("Problema médico del paciente :\n");
    gets (buffer);
    lista[i]->problema = malloc (strlen (buffer) +1);
    strcpy (lista[i]->problema, buffer);
    printf ("¿Desea continuar? (S/N) \n");
    if (getchar( ) == 'N') return;
}
}
mostrarconsulta (char * dia, struct paciente *lista)
{
    int i;
    printf ("  Pacientes con visita el dia %s\n\n", dia);
    printf (" ===== \n");
    for (i = 0; i < TAM; i++)
        if ( !strcmp (dia, lista->diavisita))
            printf ("\t%s\n", lista ->nombre);
}

```

12.18. *Escribir un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapecio y un círculo. Utilizar un array de punteros de funciones, siendo las funciones las que permiten calcular el área.*

```

main( )
{
    int i;
    float (*area[])( ) = {areaTriRect, areaTriIsoc, areaCuad, areaCirc };
    printf ("Elija el tipo de figura : \n");
    printf ("    1. Triángulo Rectángulo.\n");
    printf ("    2. Triángulo Isósceles. \n");
    printf ("    3. Cuadrado. \n");
    printf ("    4. Círculo. \n");
    scanf ("%d", &i);
    printf ("El área de la figura es %f\n ", (*area[i])( ));
}

```

```
}
float areaTriRect( )
{
    float c1, c2;
    printf ("Introduzca longitudes de catetos del triángulo: ");
    scanf ("%f %f" &c1, &c2);
    return (c1 * c2 / 2);
}
float areaTriIsoc( )
{
    float l1, l2;
    printf ("Introduzca las longitudes de dos lados desiguales del triángulo: ");
    scanf ("%f %f" &l1, &l2);
    return (l1 * l2 / 2);
}
float areaCuad( )
{
    float l;
    printf ("Introduzca la longitud de lado del cuadrado: ");
    scanf ("%f" &l);
    return (l * l);
}
float areaCirc
{
    float r;
    printf ("Introduzca el radio del círculo: ");
    scanf ("%f" &r);
    return (r * r * 3.1415);
}
```

PROBLEMAS PROPUESTOS

- 12.1.** Se tiene la ecuación $3e^x - 7x = 0$; para encontrar una raíz (una solución) escribir tres funciones que implementen respectivamente el método de Newton, Regula-Falsi y Bisección. Mediante un puntero de función aplicar uno de estos métodos para encontrar una raíz de dicha ecuación.
- 12.2.** Se parte de una tabla de claves enteras desordenadas. Se trata de mostrar las claves ordenadas sin modificar la tabla, utilizando para ello un array paralelo de punteros que apunten a cada una de las claves.
- 12.3.** Escribir un programa que calcule el determinante de una matriz cuadrada utilizando la expresión recursiva que lo obtiene sumando los determinantes de las submatrices

que resultan de ir suprimiendo la primera fila de la matriz original y una de las columnas. La fórmula suma y resta alternativamente estos determinantes parciales multiplicándolos previamente por el elemento de la primera fila de la matriz original cuya columna se ha suprimido en la submatriz. Cada uno de los determinantes de las submatrices ha de calcularse reservando dinámicamente memoria para las submatrices correspondientes.

- 12.4.** Escribir una función que tome como entrada una cadena de caracteres y devuelva un puntero a la misma cadena, pero con los caracteres al revés, poniendo en primer lugar el último de ellos.

PROBLEMAS DE PROGRAMACIÓN DE GESTIÓN

- 12.1. Dado un array que contiene una serie de registros con los datos de clientes de un establecimiento, realizar una función en la que se dé como entrada un puntero al inicio del array y el apellido de un cliente. La función debe devolver la dirección del registro que contiene los datos del cliente buscado o NULL si no lo encuentra. Incluir la función en un programa que utilice el puntero resultado para imprimir los datos del cliente.
- 12.2. La manera más usual para representar con el lenguaje C una tabla de una base de datos es por medio de un array de estructuras, correspondiendo un campo de la estructura a cada columna de la tabla. Suponer que se tiene un catálogo de un almacén con los datos de cada uno de los artículos en *stock* y que se desea cambiar el orden en que aparecen en la tabla pero sin modificar en realidad el orden en que fueron almacenados los registros en el array. Escribir un programa que añada un campo de tipo puntero al mismo tipo de estructura a cada registro. Una vez transformado el array el programa ha de hacer que el puntero de cada estructura apunte a la estructura que estaría a continuación según un nuevo orden, por ejemplo, en orden creciente de número del artículo.
- 12.3. Utilizando el array de estructuras del ejercicio anterior escribir una función que busque un artículo por su código utilizando el método de búsqueda binaria que aprovecha la ordenación de los registros por medio de los punteros.
- 12.4. Para la gestión de un vídeo club se tienen los datos de todas las películas que se pueden alquilar y de los clientes abonados. Escribir un programa que cree dos arrays de estructuras, uno para las películas de vídeo y otro para los clientes con todos sus datos. La estructura de cada película tendrá un puntero a la estructura de la tabla de clientes al registro del cliente que la ha alquilado. También al revés, el registro de cada cliente tendrá un puntero al registro de la película que tiene alquilada. El programa pedirá continuamente el nombre de cada cinta y a quién se presta o quién la devuelve, colocando a continuación los punteros de forma que apunten a los registros que correspondan.
- 12.5. Modificar el programa anterior para sacar por pantalla un informe que indique las películas alquiladas y a quién, los clientes y las películas que tienen y que permita preguntar qué cliente tiene una determinada cinta.
- 12.6. Añadir a los dos ejercicios anteriores una función que combine la información de las dos tablas, por ejemplo, creando una tabla que contenga el título de cada película con el teléfono y la dirección que quien la tiene alquilada.
- 12.7. Al crecer el negocio se van a tener varios ejemplares de la misma película y un mismo cliente va a poder tener a la vez hasta cuatro cintas alquiladas. Aumentar el número de punteros necesarios para manejar dicha situación y modificar los programas de los ejercicios anteriores de la manera adecuada.

Asignación dinámica de memoria

Los programas pueden crear variables globales o locales. Las variables declaradas globales en sus programas se almacenan en posiciones fijas de memoria, en la zona conocida como *segmento de datos* del programa, y todas las funciones pueden utilizar estas variables. Las variables locales se almacenan en la **pila** (*stack*) y existen *sólo* mientras están activas las funciones que están declaradas. Es posible, también, crear variables *static* (similares a las globales) que se almacenan en posiciones fijas de memoria, pero sólo están disponibles en el módulo (es decir, el archivo de texto) o función en que se declaran; su espacio de almacenamiento es el segmento de datos.

Todas estas clases de variables comparten una característica común: se definen cuando se compila el programa. Esto significa que el compilador reserva (define) espacio para almacenar valores de los tipos de datos declarados. Es decir, en el caso de las variables globales y locales se ha de indicar al compilador exactamente cuántas y de qué tipo son las variables a asignar. O sea, el espacio de almacenamiento se reserva en el momento de la compilación. Sin embargo, no siempre es posible conocer con antelación a la ejecución cuánta memoria se debe reservar al programa.

En C se asigna memoria en el momento de la ejecución en el *montículo o montón (heap)*, mediante las funciones `malloc()`, `realloc()`, `calloc()` y `free()`, que asignan y liberan la memoria de una zona denominada *almacén libre*.

13.1 Gestión dinámica de la memoria

En numerosas ocasiones no se conoce la memoria necesaria hasta el momento de la ejecución. Por ejemplo, si se desea almacenar una cadena de caracteres tecleada por el usuario, no se puede prever, *a priori*, el tamaño del array necesario, a menos que se reserve un array de gran dimensión y se malgaste memoria cuando no se utilice. El método para resolver este inconveniente es recurrir a punteros y a técnicas de *asignación dinámica de memoria*.

Error típico de programación en C:

En C no se puede determinar el tamaño de un array en tiempo de ejecución.

El espacio de la variable asignada dinámicamente se crea durante la ejecución del programa, al contrario que en el caso de una variable local cuyo espacio se asigna en tiempo de compilación. El programa puede crear o destruir la asignación dinámica en cualquier momento durante la ejecución. Se puede determinar la cantidad de memoria necesaria en el momento en que se haga la asignación.

El código del programa compilado se sitúa en segmentos de memoria denominados *segmentos de código*. Los datos del programa, tales como variables globales, se sitúan en un área denominada *segmento de datos*. Las variables locales y la información de control del programa se sitúan en un área denominada *pila*. La memoria que queda se denomina *memoria del montículo* o *almacén libre*. Cuando el programa solicita memoria para una variable dinámica, se asigna el espacio de memoria deseado desde el montículo.

El mapa de memoria del modelo de un programa grande es muy similar al mostrado en la Figura 13.1. El diseño exacto dependerá del modelo de programa que se utilice. Para grandes modelos de datos, el almacén libre (*heap*) se refiere al área de memoria que existe dentro de la pila del programa. Y el almacén libre es, esencialmente, toda la memoria que queda libre después de que se carga el programa.

En C las funciones `malloc()`, `realloc()`, `calloc()` y `free()` asignan y liberan memoria de un bloque de memoria denominado el *montículo del sistema*. Las funciones `malloc()`, `calloc()` y `realloc()` asignan memoria utilizando *asignación dinámica* debido a que puede gestionar la memoria durante la ejecución de un programa.

13.2 Función `malloc()`

La forma más habitual en C para obtener bloques de memoria es mediante la llamada a la función `malloc()`. La función reserva un bloque de memoria cuyo tamaño es el número de bytes pasados como argumento. `malloc()` devuelve un puntero, que es la dirección del primer byte asignado de memoria. El puntero se utiliza para referenciar el bloque de memoria. El puntero que devuelve es del tipo `void*`. La forma de llamar a la función `malloc()` es:

```
puntero = malloc(tamaño en bytes);
```

Generalmente se hará una conversión al tipo del puntero:

```
tipo *puntero;
puntero =(tipo *)malloc(tamaño en bytes);
```

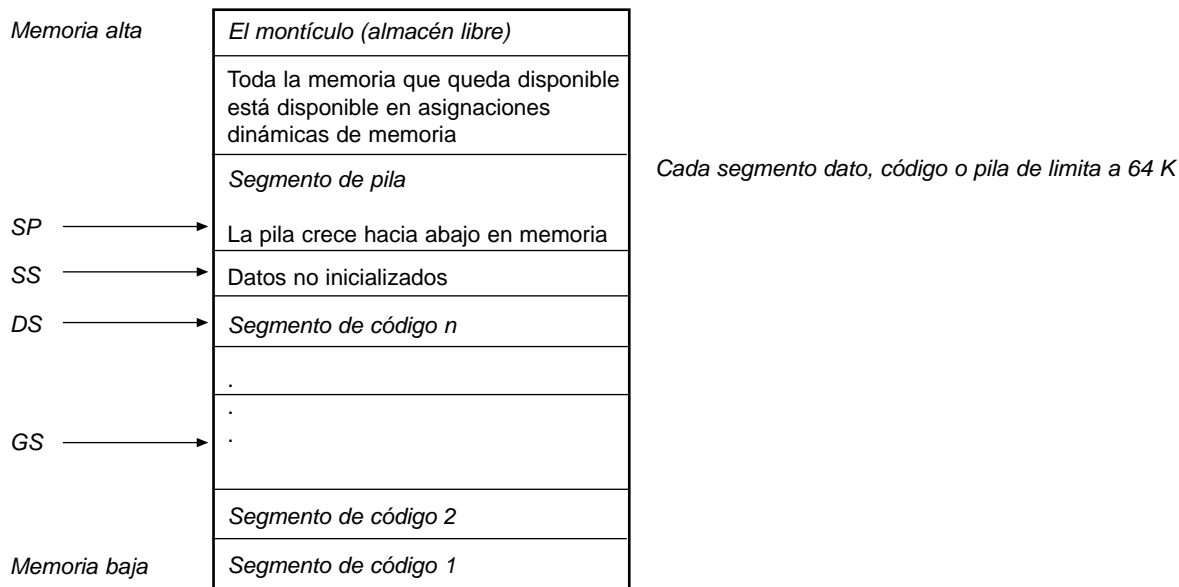


Figura 13.1 Mapa de memoria de un programa.

El operador unitario `sizeof` se utiliza con mucha frecuencia en las funciones de asignación de memoria. El operador se aplica a un tipo de dato (o una variable), el valor resultante es el número de bytes que ocupa. Al llamar a la función `malloc()` puede ocurrir que no haya memoria disponible, en ese caso `malloc()` devuelve `NULL`. Hay que comprobar *siempre* el puntero para asegurar que es válido, antes de que se asigne un valor al puntero. El prototipo es:

```
void* malloc(size_t n);
```

13.3 Liberación de memoria, función `free()`

Cuando se ha terminado de utilizar un bloque de memoria previamente asignado por `malloc()`, u otras funciones de asignación, se puede liberar el espacio de memoria y dejarlo disponible para otros usos, mediante una llamada a la función `free()`. El bloque de memoria suprimido se devuelve al espacio de almacenamiento libre, de modo que habrá más memoria disponible para asignar otros bloques de memoria. El prototipo es:

```
void free(void *);
```

13.4 Funciones `calloc()` y `realloc()`

En la sintaxis de llamada, *puntero* es el nombre de la variable puntero al que se asigna la dirección de memoria de un bloque, o `NULL` si falla la operación de asignación de memoria. El prototipo de `calloc()` es:

```
void* calloc(size_t numelementos, size_t tamaño);
```

La función `realloc()` permite ampliar un bloque de memoria reservado anteriormente. El puntero a bloque referencia a un bloque de memoria reservado previamente con `malloc()`, `calloc()` o la propia `realloc()`. El prototipo de `realloc()` es:

```
void* realloc(void* puntero_a_bloque, size_t t);
```

El segundo argumento de `realloc()`, es el tamaño total que va a tener el bloque de memoria libre. Si se pasa cero (0) como tamaño se libera el bloque de memoria al que está apuntando el puntero primer argumento, y la función devuelve `NULL`. Si el primer argumento tiene el valor de `NULL`, la función reserva tanta memoria como la indicada por el segundo argumento, como `malloc()`.

Hay que tener en cuenta que la expansión de memoria que realiza `realloc()` puede hacerla en otra dirección de memoria de la que contiene la variable puntero transmitida como primer argumento. En cualquier caso, `realloc()` copia los datos referenciados por puntero en la memoria expandida. El prototipo de las funciones está en `stdlib.h`.

Se pueden crear dinámicamente arrays multidimensionales de objetos con las funciones de asignación de memoria. Para crear un array bidimensional $n \times m$, en primer lugar, se asigna memoria para un array de punteros (de n elementos), y después se asigna memoria para cada fila (m elementos).

EJEMPLO 13.1 Definición de arrays dinámicos.

En C no se pueden definir arrays o matrices de tamaño variable. Sin embargo, se puede utilizar la asignación dinámica de memoria para el mismo propósito.

```
#define N 200

int a[N];
int *b;
int i, tam;

b= (int*) malloc (N * sizeof (int));

for (i = 0; i < N; i++)
    b[i] = a[i];

printf ("Introduzca un nuevo tamaño para el array b: ");
scanf ("%d", &tam);

realloc (b, tam);
```

PROBLEMAS RESUELTOS

13.1. Encuentre los errores en las siguientes declaraciones y sentencias.

```
int n, *p;
char** dob= "Cadena de dos punteros";
p = n*malloc(sizeof(int));
```

Análisis del problema

Una cadena de caracteres se almacena en C como un array de caracteres de tipo `char`, por lo que puede ser accedido por un puntero a `char`, pero no como un puntero doble:

```
char* dob= "Cadena de dos punteros";
```

La función `malloc` devuelve un puntero a la zona de memoria reservada en tiempo de ejecución, por lo cual no tiene sentido hacer una operación de multiplicación con la dirección que devuelve. El resultado sería válido sintácticamente, pero no daría como resultado una dirección de memoria que se pudiera utilizar. Produciría un error grave de violación de acceso a memoria. Habría que hacerlo así:

```
p = malloc (sizeof (int));
```

13.2. Dada la siguiente declaración, definir un puntero `b` a la estructura, reservar memoria dinámicamente para una estructura asignando su dirección a `b`.

```
struct boton
{
    char* rotulo;
    int codigo;
};
```

Codificación

```
struct boton *b;
b = (struct boton *) malloc (sizeof (struct boton));
```

13.3. Una vez asignada memoria al puntero `b` del ejercicio 13.2 escribir sentencias para leer los campos `rotulo` y `código`.

Análisis del problema

Se trata de utilizar el operador de indirección flecha para acceder desde el puntero `b` a cada uno de los campos de la estructura:

Codificación

```
scanf ("%s", b->rotulo);
printf ("El rotulo del botón es: %s\n", b->rotulo);
scanf ("%d", &b->codigo);
printf ("El código del boton es: %d\n", b->codigo);
```

13.4. Declara una estructura para representar un punto en el espacio tridimensional. Declara un puntero a la estructura para que tenga la dirección de un array dinámico de `n` estructuras punto. Utiliza la función `calloc()` para asignar memoria al array y comprueba que se ha podido asignar la memoria requerida.

```

struct punto3D
{
    int x;
    int y;
    int z;
};
struct punto3D *poligono;
int n;
if ((poligono = (struct punto3D*) calloc (n, sizeof (struct punto3D))) == NULL)
{
    printf ("Error de asignación de memoria dinámica");
    exit(1);
}

```

- 13.5.** *Dada la declaración de la estructura punto del ejercicio anterior escribe una función que devuelva la dirección de un array dinámico de n puntos en el espacio tridimensional. Los valores de los datos se leen del dispositivo de entrada(teclado).*

Análisis del problema

La única forma de definir un array dinámico en C es declarar un puntero al tipo de datos de los elementos del array. Durante la ejecución del programa se reservará memoria para tantos elementos como se requiera. El lenguaje C no permite declarar un array sin especificar su tamaño en tiempo de compilación; ello no es problema para utilizar el puntero al array como si se tratase del nombre de un array declarado como tal, puesto que C trata todos los arrays como punteros a sus elementos y ve los corchetes como abreviaturas de operaciones con punteros para localizar la dirección de cada uno de sus elementos según están dispuestos en la memoria.

Codificación

```

struct punto3D * leerPuntos (int numpuntos)
{
    struct punto3D * poligono, *ppol;
    int i;
    if ((poligono = (struct punto3D*) calloc (numpuntos, sizeof (struct punto3D))) == NULL)
    {
        printf ("Error de asignación de memoria dinámica");
        exit(1);
    }
    for (i=0, ppol = poligono; i<numpuntos; i++, ppol++)
    {
        printf ("\n\tIntroduzca coordenadas x,y,z siguiente punto :");
        scanf ("%d %d %d", ppol->x, ppol->y, ppol->z);
    }
    return (poligono);
}

```

- 13.6.** *Dada la declaración del array de punteros:*

```

#define N 4
char *linea[N];

```

escriba las sentencias del código fuente para leer N líneas de caracteres y asignar cada línea a un elemento del array.

Análisis del problema

La forma más eficiente de manejar un texto formado por líneas, que son cadenas de texto, es declarar un puntero doble a carácter para que apunte a un array de punteros a carácter, los cuales serán los que contengan la dirección de comienzo de cada línea del texto. Hay que recordar que se necesita un array que funcione como *buffer* temporal para leer cada línea y así poder calcular la longitud que tienen, con lo que se sabe cuánta memoria se va a utilizar para cada línea y no reservar más que para los caracteres de la cadena leída más uno para el carácter fin de cadena: `'\0'`.

Codificación

```
int i;
char temp[80];
for (i=0; i<N; i++)
{
    printf ("\n Introduzca la siguiente línea: ");
    gets (temp);
    linea[i] = (char*) malloc (strlen (temp)+1);
    strcpy (linea[i], temp);
}
```

13.7. Escriba una función que reciba el array dinámico creado en el ejercicio 13.4 y amplíe el array en otros *m* puntos del espacio.

Análisis del problema

Esta operación es muy sencilla ya que el array ha sido manejado en todo momento a partir de un solo puntero que apunta a la primera posición de los datos. Solamente hay que pasar este puntero a la función `realloc()` para que busque un espacio de memoria mayor y mueva allí los datos del array para seguir añadiendo más registros.

Codificación

```
struct punto3D * ampliarpoligono (struct punto3D * poligono, int m)
{
    struct punto3D *ppol;
    int i;
    if ((poligono = (struct punto3D*) realloc (poligono, sizeof (struct punto3D)*m)) == NULL)
    {
        printf ("Error de reasignación de memoria dinámica");
        exit(1);
    }
    for (i=0, ppol = poligono; i<m; i++, ppol++)
    {
        printf ("\n\tIntroduzca las coordenadas x, y, z del siguiente punto :");
        scanf ("%d %d %d", ppol->x, ppol->y, ppol->z);
    }
    return (poligono);
}
```

13.8. Escriba una función que reciba las *N* líneas leídas en el ejercicio 13.6 y libere las líneas de longitud menor de 20 caracteres. Las líneas restantes han de quedar en orden consecutivo, desde la posición cero.

Análisis del problema

La declaración del array de líneas era:

```
#define N 4
char *linea[N];
```

Ahora, suponiendo que el array contiene líneas válidas, se va a proceder a la eliminación de aquellas con una longitud menor a 20 caracteres. Para que el array de líneas quede como se espera, además de eliminar las líneas hay que mover las siguientes a la posición anterior, con el fin de que estén seguidas y sin huecos entre ellas que puedan revelarse a la hora de imprimirlas por pantalla.

Codificación

```
int i,j;
for (i=0; i<N; i++)
{
    if (strlen (linea[i]) < 20)
    {
        free (linea[i]);
        /* se libera el espacio ocupado por la línea corta */
        for (j=i; j<N, linea[j]; j++)
            /* se mueven sólo las líneas con contenido */
            linea [j] = linea [j+1];
        linea[j+1] = NULL;
    }
}
```

13.9. ¿Qué diferencias existen entre las siguientes declaraciones?:

```
char *c[15];
char **c;
char c[15][12];
```

Análisis del problema

```
char *c[15];
```

Es un array de 15 punteros a datos de tipo char o cadenas de caracteres, pero estos punteros están sin inicializar: no apuntan a ninguna posición de memoria válida.

```
char **c;
```

Es una variable que puede contener la dirección de un puntero a datos de tipo char o cadenas de caracteres. Tampoco está inicializado, por lo cual no apunta a ningún puntero válido a char.

```
char c[15][12];
```

Es una matriz de caracteres con 15 filas y 12 columnas. Tiene espacio de memoria ya reservado para 15 * 12 caracteres.

13.10. *Escribe un programa para leer n cadenas de caracteres. Cada cadena tiene una longitud variable y está formada por cualquier carácter. La memoria que ocupa cada cadena se ha de ajustar al tamaño que tiene. Una vez leídas las cadenas se debe realizar un proceso que consiste en eliminar todos los blancos, siempre manteniendo el espacio ocupado ajustado al número de caracteres. El programa debe mostrar las cadenas leídas y las cadenas transformadas.*

Análisis del problema

En primer lugar, para que cada línea sólo ocupe la memoria necesaria para sus caracteres y el carácter final de cadena, se realiza el procedimiento de los ejercicios anteriores, esto es, leer la cadena en un *buffer* y después reservar memoria para la línea leída nada más. Se dispone de un array de punteros a cadenas que van a ir conteniendo las direcciones de comienzo de las cadenas según se vayan almacenando en memoria dinámicamente. Por otro lado, como hay que dejar las cadenas originales tal como están, el programa define otro array de punteros a cadenas para contener las cadenas transformadas.

Para direccionar cada línea en vez de utilizar el operador corchete se usa la operación de suma sobre el puntero de inicio del array. Como los arrays siempre se comienzan a numerar desde la posición cero, la *i-ésima* línea está a una distancia *i* del origen del array, posición a la que está apuntando el puntero declarado.

Codificación

```
main (int argc, char *argv[])
{
    char **linea, **lineaT;
    int n = atoi(argv[1]);
    linea = (char **)malloc (n * sizeof char *);
    lineaT = (char **)malloc (n * sizeof char *);
    leerCadenas (linea, n);
    transformarCadenas (linea, lineaT, n);
    mostrarCadenas (linea, n);
    mostrarCadenas (lineaT, n);
}

leerCadenas ( char **linea, int n)
{
    int i;
    char temp[80];
    for (i=0; i<n; i++)
    {
        printf ("\nIntroduzca la siguiente línea:");
        gets (temp);
        linea[i] = (char*) malloc (strlen (temp)+1);
        strcpy (linea[i], temp);
    }
}

transformarCadenas (char **lineas, char **lineasT, int n)
{
    int i, j, k;
    char cad [80];
    for (i=0; i<n; i++)
    {
        lon = strlen (*(lineas+i));
        strcpy (cad, *(lineas+i));
        for (j=0; j <lon; j++)
        {
            if ( cad[j] == ' ')
                for (k = j; k<lon; k++)
                    cad [k] = cad [k+1];
            cad[k]='\0';
        }
        strcpy (*(lineasT + i), cad);
    }
}
```

```

}
mostrarCadenas (char **lineas, int n)
{
    int i, j;
    for (i=0; i<n; i++)
    {
        lon = strlen (*(lineas+i));
        for (j=0; j <lon; j++)
            printf ("%c", *((lineas +i ) + j));
        printf ("\n");
    }
}

```

13.11. *Se quiere escribir un programa para leer números grandes (de tantos dígitos que no entran en variables long) y obtener la suma de ellos. El almacenamiento de un número grande se ha de hacer en una estructura que tenga un array dinámico y otro campo con el número de dígitos. La suma de dos números grandes dará como resultado otro número grande representado en su correspondiente estructura.*

Análisis del problema

El número se leerá como cadena y se guardará como un array de enteros.

Codificación

```

struct grande
{
    int *digitos;
    int numdigitos;
};
main ( )
{
    struct grande sum1, sum2, res;
    leerNumeroGrande (&sum1);
    leerNumeroGrande (&sum2);
    res = *sumarNumerosGrandes (&sum1, &sum2);
    mostrarNumeroGrande (&res);
}
leerNumeroGrande (struct grande *numgr)
{
    char buffer [100], *pbuf;
    int i;
    printf ("\n\tIntroduzca el número : ");
    gets (buffer);
    numgr->digitos = (int*) malloc ((strlen (buffer)+1) * sizeof (int));
    numgr->numdigitos = strlen (buffer);
    /* recorrer la cadena con el número y cada dígito
       convertirlo a entero para guardarlo en el array de dígitos */
    for (pbuf = buffer, i=0; pbuf - buffer < strlen (buffer); pbuf++, i++)
        /* almacenar primero los dígitos menos significativos */
        *(numgr->digitos + numgr->numdigitos-i) = *pbuf - '0';
}
struct grande *sumarNumerosGrandes (struct grande *ngr1, struct grande *ngr2)
{

```

```

int i, maxLong;
struct grande *result;
int *pgr1, *pgr2, *pres;
result = (struct grande *) malloc (sizeof (struct grande));
maxLong = ngr1->numdigitos > ngr2->numdigitos ? ngr1->numdigitos : ngr2->numdigitos;
result->digitos = (int*) malloc (maxLong * sizeof (int));
result->numdigitos = maxLong;
i=0;
pgr1 = ngr1->digitos;
pgr2 = ngr2->digitos;
pres = result->digitos;
while ( i < ngr1->numdigitos && i++ < ngr2->numdigitos)
    *pres++ = *pgr1++ + *pgr2++;
if (i >= ngr1->numdigitos)
    for (i = ngr1->numdigitos + 1; i < ngr2->numdigitos; i++)
        *pres++ = *pgr2++;
if (i >= ngr2->numdigitos)
    for (i = ngr2->numdigitos + 1; i < ngr1->numdigitos; i++)
        *pres++ = *pgr1++;
/* calcular arrastre */
pres = result->digitos;
for (i=0; i< result->numdigitos; i++, pres++)
    if (pres[i] - '0' > 10 )
    {
        *pres = (*pres) % 10;
        *(pres+1) += 1;
    }
return (result);
}

mostrarNumeroGrande (struct grande *ngr)
{
    int i;
    for (i=0; i< ngr->numdigitos; i++)
        printf ("%c", *ngr->digitos++ + '0');
}

```

13.12. Se tiene una matriz de 20x20 elementos enteros. En la matriz hay un elemento repetido muchas veces. Se quiere generar otra matriz de 20 filas y que en cada fila estén sólo los elementos no repetidos. Escribir un programa que tenga como entrada la matriz de 20x20, genere la matriz dinámica solicitada y se muestre en pantalla.

Análisis del problema

Suponiendo que el elemento repetido es mayoritario, es decir, aparece en más de la mitad de posiciones de cada fila, lo primero es averiguar cuál es y después dejar en las filas de la nueva matriz sólo los otros elementos.

Codificación (Consultar la página web del libro)

13.13. Escribir un programa para generar una matriz simétrica con números aleatorios de 1 a 9. El usuario introduce el tamaño de cada dimensión de la matriz y el programa reserva memoria libre para el tamaño requerido.

Análisis del problema

Una matriz simétrica es una matriz en la cual los elementos que son simétricos respecto al eje son los mismos. Por tanto, solamente hay que generar la mitad de la matriz y asignar lo mismo a los dos elementos simétricos que intercambian sus

posiciones de fila por columna y al revés. Como se está haciendo en todos los ejercicios, en vez de utilizar los operadores corchetes para acceder a los elementos de la matriz bidimensional, se realiza la misma operación que hace el compilador, es decir, calcular dónde comienza cada fila a partir del puntero al comienzo del array y el tamaño de cada columna y después situar la dirección del elemento en la fila usando su número de columna.

Codificación

```
main( )
{
    int **simetrica, n, i, j;
    randomize( );
    printf ("Introduzca la dimensión de la matriz simétrica: ");
    scanf ("%d", &n);
    simetrica = (int **) malloc (n*sizeof (int*));
    for (i=0; i<n; i++)
        simetrica[i] = (int *)malloc (n*sizeof (int));
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            *(simetrica[i] + j) =
                *(simetrica[j] + i) = rand ( )%10;
    for (i=0; i<n; i++)
        *(simetrica[i] + i) = 0;
}
```

- 13.14.** *Escribir un programa para manejar enteros no negativos como cadenas de caracteres. Teniendo como entrada un entero n mayor que cero, transformar n en una cadena de dígitos, cad ; mediante una función que tenga como entrada cad , transformar el valor entero n en $2*n$ que será devuelto como otra cadena. La salida mostrará ambas cadenas.*

Análisis del problema

Esta operación ha sido realizada ya en otros ejercicios. Solamente observe que después de hacer la operación dígito a dígito, hay que tener en cuenta el arrastre, para que en cada carácter haya sólo una cifra y así corresponda su valor numérico con el carácter (su código ASCII) correspondiente.

Codificación

```
main ( )
{
    int n;
    char *cad, *cad2;
    printf ("Escriba un número entero positivo: ");
    scanf ("%d", &n);
    ntoa (n, cad);
    cad2 = doble (cad);
}

ntoa (int n, char *cad)
{
    char aux[40];
    char paux = aux;
    while (n>0)
    {
        *paux++ = n % 10 + '0';
        n = n / 10;
    }
}
```

```

    }
    *paux = '0';
    cad = (char *) malloc (sizeof (char) * (paux - aux));
    strcpy(cad,aux);
}
char* doble (char *cad)
{
    int i, longitud = strlen (cad);
    char * aux, *cadDoble = (char *) malloc (longitud * sizeof(char)+2);
    aux = cadDoble;
    for (i=0; i<longitud; i++)
        *aux++ = (*cad++ - '0') * 2 + '0';
    return (normaliz (cadDoble));
}
char * normaliz (char * cad)
{
    int i, longitud = strlen (cad);
    char * aux = cad;
    for (i=0; i<longitud; i++, aux++)
        if (*aux - '0' > 10 )
            { /* calcula el arrastre */
                *aux = ((*aux - '0') % 10) + '0';
                *(aux+1) += 1;
            }
    return cad;
}

```

13.15. Una malla de números enteros representa imágenes, cada entero expresa la intensidad luminosa de un punto. Un punto es un elemento «ruido» cuando su valor se diferencia en dos o más unidades del valor medio de los ocho puntos que le rodean. Escribir un programa que tenga como entrada las dimensiones de la malla, reserve memoria dinámicamente para una matriz en la que se lean los valores de la malla. En una función que reciba una malla, devuelva otra malla de las mismas dimensiones donde los elementos «ruido» tengan el valor 1 y los que no lo son valor 0. Todos los puntos que se encuentran en el contorno de la malla no tienen «ruido».

Análisis del problema

En este caso la malla se diseña como un **arreglo** (array) bidimensional de filas y columnas, en vez de, como en otros casos, definir primero un array de punteros que contendrán la dirección de las filas.

En la entrada de datos desde la línea de órdenes hacia el argumento `argv` de la función `main`, hay que recordar siempre que los argumentos se almacenan como cadenas de caracteres a las que apuntan los punteros del array direccionado por `argv`; por tanto, si un argumento es numérico, ha de ser convertido a entero antes de ser asignado a una variable entera o real.

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

- 13.1.** En una competición de ciclismo se presentan n ciclistas. Cada participante se representa por nombre, club, puntos obtenidos y prueba en que participará en la competición. La competición es por eliminación. Hay pruebas de dos tipos: persecución y velocidad. En *persecución* participan tres ciclistas, el primero recibe 3 puntos y el tercero se elimina. En *velocidad* participan 4 ciclistas, el más rápido obtiene 4 puntos el segundo 1 y el cuarto se elimina. Las pruebas se van alternando, empezando por velocidad. Los ciclistas participantes en una prueba se eligen al azar entre los que en menos pruebas han participado. El juego termina cuando no quedan ciclistas para alguna de las dos pruebas. Se ha de mantener arrays dinámicos con los ciclistas participantes y los eliminados. El ciclista ganador será el que más puntos tenga.
- 13.2.** Un polinomio, $P(x)$, puede representarse con un array de tantos elementos como el grado del polinomio más uno. Escribir un programa que tenga como entrada el grado n del polinomio, reserve memoria dinámicamente para un array de $n+1$ elementos. En una función se introducirán por teclado los coeficientes del polinomio, en orden decreciente. El programa tiene que permitir evaluar el polinomio para un valor dado de x .
- 13.3.** Una operación común en el tratamiento digital de imágenes consiste en aplicar un filtro para suavizar los bordes entre las figuras dibujadas. Uno de estos filtros consiste en modificar el valor de la imagen en cada punto por la media de los valores de los ocho puntos que tiene a su alrededor. Escribir un programa que parta de la imagen como una matriz de elementos enteros que representan cada uno de ellos la intensidad de cada píxel en la imagen y que aplique la operación de filtrado descrita.
- 13.4.** Escribir un programa que permita sumar, restar, multiplicar y dividir números reales con signo representados con cadenas de caracteres con un solo dígito por carácter.
- 13.5.** Escribir una función que tome como entrada un número entero y produzca una cadena con los dígitos de su expresión en base binaria.

Cadenas

El lenguaje C no tiene datos predefinidos tipo cadena (*string*). En su lugar C, manipula cadenas mediante arrays de caracteres que terminan con el carácter nulo ASCII (`'\0'`). Una cadena se considera como un array unidimensional de tipo `char` o `unsigned char`. En este capítulo se estudiarán temas tales como:

- Cadenas en C.
- Lectura y salida de cadenas.
- Uso de funciones de cadena de la biblioteca estándar.
- Asignación de cadenas.
- Operaciones diversas de cadena (longitud, concatenación, comparación y conversión).
- Localización de caracteres y subcadenas; inversión de los caracteres de una cadena.

14.1 Concepto de cadena

Una *cadena* es un tipo de dato compuesto, un array de caracteres (`char`), terminado por un carácter *nulo* (`'\0'`), NULL (Fig. 14.1).

Una cadena (también llamada *constante de cadena* o *literal de cadena*) es "ABC". En memoria esta cadena consta de cuatro elementos: 'A', 'B', 'C' y '\0', o de otra manera, se considera que la cadena "ABC" es un array de cuatro elementos de tipo `char`. El valor real de una cadena es la dirección de su primer carácter y su tipo es un puntero a `char`.

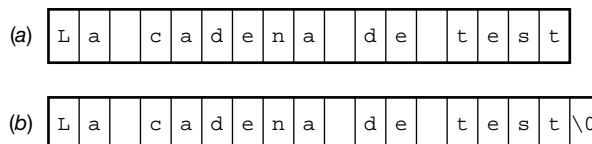


Figura 14.1 (a) array de caracteres; (b) cadena de caracteres.

El número total de caracteres de una cadena en C es siempre igual a la longitud de la cadena más 1.

14.2 Inicialización de variables de cadena

Todos los tipos de arrays requieren una inicialización que consiste en una lista de valores separados por comas y encerrados entre llaves.

```
char texto[81] = "Esto es una cadena";
char cadenatest[] = "¿Cuál es la longitud de esta cadena?";
```

La cadena `texto` puede contener 80 caracteres más el carácter nulo. La tercera cadena, `cadenatest`, se declara con una especificación de tipo incompleta y se completa sólo con el inicializador. Dado que en el literal hay 36 caracteres y el compilador añade el carácter `'\0'`, un total de 37 caracteres se asignarán a `cadenatest`.

Ahora bien, una cadena no se puede inicializar fuera de la declaración. La razón es que un identificador de cadena, como cualquier identificador de array, se trata como un valor de dirección, como un puntero constante.

Para asignar una cadena a otra hay que utilizar la función `strcpy()`. La función `strcpy()` copia los caracteres de la cadena fuente a la cadena destino. La función supone que la cadena destino tiene espacio suficiente para contener toda la cadena fuente.

EJEMPLO 14.1 Inicialización de variables de tipo cadena

Es aconsejable declarar las cadenas que se inician de tipo estático.

```
static char cadena [] = "Cadena estática";
char *cadena2;

/* No olvidar reservar espacio para cadenas gestionadas por medio de punteros */
cadena2 = (char*) malloc (strlen(cadena) + 1);
strcpy (cadena2, cadena);
```

14.3 Lectura de cadenas

La lectura usual de datos se realiza con la función `scanf()`; cuando se aplica a datos cadena el código de formato es `%s`. La función da por terminada la cadena cuando encuentra un espacio en blanco o el fin de línea. Se puede utilizar la función `gets()`, la cual permite leer la cadena completa, incluyendo cualquier espacio en blanco, hasta el carácter de fin de línea.

La función asigna la cadena al argumento transmitido a la función, que será un array de caracteres o un puntero (`char*`) a memoria libre, con un número de elementos suficiente para guardar la cadena leída. Si ha habido un error en la lectura de la cadena, devuelve `NULL`.

La función `getchar()` se utiliza para leer carácter a carácter. La llamada a `getchar()` devuelve el carácter siguiente del flujo de entrada `stdin`. En caso de error, o de encontrar el fin de archivo, devuelve `EOF` (macro definida en `stdio.h`).

La función `putchar()` se utiliza para escribir en la salida (`stdout`) carácter a carácter. El carácter que se escribe es el transmitido como argumento. Esta función (realmente es una macro definida en `stdio.h`) tiene como prototipo:

```
int putchar(int ch);
```

La función `puts()` escribe en la salida una cadena de caracteres, incluyendo el carácter fin de línea por lo que sitúa el puntero de salida en la siguiente línea. El prototipo es:

```
int puts(const char*s);
```

Las funciones `getch()` y `getche()` leen un carácter tecleado sin esperar el retorno de carro. La diferencia entre ellas está en que con `getch()` el carácter tecleado no se visualiza en pantalla (no hace eco en la pantalla), y con `getche()` sí hay eco en la pantalla. El prototipo de ambas funciones se encuentra en el archivo `conio.h`

```
int getch(void);
int getche(void);
```

EJEMPLO 14.2 *Lectura y escritura de cadenas*

```

char entrada[40];
char *ptrchar;

printf ("Introduzca una cadena de caracteres: ");
ptrchar = gets (entrada);

printf ("\n Esta es la cadena introducida: ");
for (; *ptrchar != '\0'; ptrchar++)
    putchar (*ptrchar);

puts ("\n Presione una tecla para terminar");
getch ();

```

EJEMPLO 14.3 *Manipulación de cadenas*

Se desea leer líneas de texto, máximo de 80 caracteres, y contar el número de palabras que tiene cada línea.

Cada línea se lee llamando a la función `gets()`, con un argumento que pueda almacenar el máximo de caracteres de una línea. Por consiguiente se declara la variable: `char cad[81]`, que será el argumento de `gets()`. Con el fin de simplificar, se supone que las palabras se separan con un espacio; entonces para contar las palabras se recorre el array `cad` contando el número de espacios, la longitud de la cadena se determina con una llamada a `strlen()`. El número de palabras será el número de espacios (blancos) contados, mas uno ya que la última palabra no termina con un espacio sino con el retorno de carro. La ejecución termina tecleando al inicio de una línea `^Z` (tecla “control” y Z); entonces la función `gets()` devuelve `NULL` y termina el bucle.

```

#include <stdio.h>
#include <string.h>
void main()
{
    char cad[81], *a;
    int i, n;

    puts ("Introduce líneas, separando las palabras con blancos.\n ");
    a = gets (cad);
    while (a != NULL)
    {
        n = 0;
        for (i = 0; i < strlen(cad); i++)
            if (cad[i] == ' ') n++;
            /* también se accede a los char con *(cad+i) */
        if (i > 0) ++n;
        printf ("Número de palabras: %d \n", n);
        a = gets (cad);
    }
}

```

14.4 Las funciones de `STRING.H`

La biblioteca estándar de C contiene las funciones de manipulación de cadenas utilizadas más frecuentemente. Cuando se utiliza la función, se puede usar un puntero a una cadena o se puede especificar el nombre de una variable array de `char`. La Tabla 14.1 resume algunas funciones de cadena más usuales.

Tabla 11.1. Funciones de `<string.h>`

Función	Cabecera de la función y prototipo
<code>memcpy()</code>	<code>void* memcpy(void* s1, const void* s2, size_t n);</code> Reemplaza los primeros <i>n</i> bytes de <i>*s1</i> con los primeros <i>n</i> bytes de <i>*s2</i> . Devuelve <i>s1</i> .
<code>strcat()</code>	<code>char* strcat(char* destino, const char* fuente);</code> Añade la cadena <i>fuente</i> al final de <i>destino</i> , <i>concatena</i> .
<code>strchr()</code>	<code>char* strchr(char* s1, int ch);</code> Devuelve un puntero a la primera ocurrencia de <i>ch</i> en <i>s1</i> . Devuelve NULL si <i>ch</i> no está en <i>s1</i> .
<code>strcmp()</code>	<code>int strcmp(const char* s1, const char* s2);</code> Compara alfabéticamente la cadena <i>s1</i> a <i>s2</i> y devuelve: 0 <i>si</i> <i>s1</i> = <i>s2</i> <0 <i>si</i> <i>s1</i> < <i>s2</i> >0 <i>si</i> <i>s1</i> > <i>s2</i>
<code>stricmp()</code>	<code>int stricmp(const char* s1, const char* s2);</code> Igual que <code>strcmp()</code> , pero sin distinguir entre mayúsculas y minúsculas.
<code>strcpy()</code>	<code>char* strcpy(char* dest, const char* fuente);</code> Copia la cadena <i>fuente</i> a la cadena <i>destino</i> .
<code>strncpy()</code>	<code>char* strncpy(char* dest, const char* fuente, size_t num);</code> Copia la cadena <i>fuente</i> a la cadena <i>destino</i> .
<code>strcspn()</code>	<code>size_t strcspn(const char* s1, const char* s2);</code> Devuelve la longitud de la subcadena más larga de <i>s1</i> que comienza con el carácter <i>s1[0]</i> y no contiene ninguno de los caracteres de la cadena <i>s2</i> .
<code>strlen()</code>	<code>size_t strlen (const char* s)</code> Devuelve la longitud de la cadena <i>s</i> excluyendo el carácter nulo de terminación de la cadena.
<code>strncat()</code>	<code>char* strncat(char* s1, const char* s2, size_t n);</code> Añade los primeros <i>n</i> caracteres de <i>s2</i> a <i>s1</i> . Devuelve <i>s1</i> . Si <i>n</i> >= <code>strlen(s2)</code> , entonces <code>strncat(s1, s2, n)</code> tiene el mismo efecto que <code>strcat(s1, s2)</code> .
<code>strncmp()</code>	<code>int strncmp(const char* s1, const char* s2, size_t n);</code> Compara <i>s1</i> con la subcadena formada por los primeros <i>n</i> caracteres de <i>s2</i> . Devuelve un entero negativo, cero o un entero positivo, según que <i>s1</i> lexicográficamente sea menor, igual o mayor que la subcadena <i>s2</i> . Si <i>n</i> ≥ <code>strlen(s2)</code> , entonces <code>strncmp(s1, s2, n)</code> y <code>strcmp(s1, s2)</code> tienen el mismo efecto.
<code>strnset()</code>	<code>char* strnset(char* s, int ch, size_t n);</code> Copia <i>n</i> veces el carácter <i>ch</i> en la cadena <i>s</i> a partir de la posición inicial de <i>s</i> (<i>s[0]</i>). El máximo de caracteres que copia es la longitud de <i>s</i> .
<code>strpbrk()</code>	<code>char* strpbrk(const char* s1, const char* s2);</code> Devuelve la dirección de la primera ocurrencia en <i>s1</i> de cualquiera de los caracteres de <i>s2</i> . Devuelve NULL si ninguno de los caracteres de <i>s2</i> aparece en <i>s1</i> .
<code>strrchr()</code>	<code>char* strrchr(const char* s, int c);</code> Devuelve un puntero a la última ocurrencia de <i>c</i> en <i>s</i> . Devuelve NULL si <i>c</i> no está en <i>s</i> . La búsqueda la hace en sentido inverso, desde el final de la cadena al primer carácter, hasta que encuentra el carácter <i>c</i> .
<code>strspn()</code>	<code>size_t strspn(const char* s1, const char* s2);</code> Devuelve la longitud de la subcadena izquierda(<i>s1[0]</i>)... más larga de <i>s1</i> que contiene únicamente caracteres de la cadena <i>s2</i> .

Continúa

<code>strrev()</code>	<code>char*strrev(char*s);</code> Invierte el orden de los caracteres de la cadena especificada en el argumento <code>s</code> ; devuelve un puntero a la cadena resultante.
<code>strstr()</code>	<code>char*strstr(const char*s1, const char*s2);</code> Busca la cadena <code>s2</code> en <code>s1</code> y devuelve un puntero a los caracteres donde se encuentra <code>s2</code> .
<code>strtok()</code>	<code>char* strtok(char* s1, const char* s2);</code> Analiza la cadena <code>s1</code> en <i>tokens</i> (componentes léxicos), éstos delimitados por caracteres de la cadena <code>s2</code> . La llamada inicial a <code>strtok(s1, s2)</code> devuelve la dirección del primer token y sitúa <code>NULL</code> al final del token. Después de la llamada inicial, cada llamada sucesiva a <code>strtok(NULL, s2)</code> devuelve un puntero al siguiente <i>token</i> encontrado en <code>s1</code> . Estas llamadas cambian la cadena <code>s1</code> , reemplazando cada separador con el carácter <code>NULL</code> .

14.5 Conversión de cadenas a números

La función `atoi()` convierte una cadena a un valor entero. Su prototipo es:

```
int atoi(const char*cad);
```

La cadena debe tener la representación de un valor entero y el formato siguiente:

[espacio en blanco] [signo] [dígitos]

Si la cadena no se puede convertir, `atoi()` devuelve cero.

La función `atof()` convierte una cadena a un valor de coma flotante. Su prototipo es:

```
double atof(const char*cad);
```

La conversión termina cuando se encuentre un carácter no reconocido. La cadena de caracteres debe tener una representación de caracteres de un número de coma flotante. Su formato es:

[espacio en blanco][signo][ddd][.][ddd][e/E][signo][ddd]

La función `atol()` convierte una cadena a un valor largo (`long`). Su prototipo es

```
long atol(const char*cad);
```

La utilidad las funciones `strtol()` y `strtoul()` radica en que convierten los dígitos de una cadena, en cualquier sistema de numeración (`base`), a entero (`long`) o a entero sin signo (`unsigned long`). El prototipo de las funciones se encuentra en `stdio.h`, es el siguiente:

```
long strtol (const char* c, char** pc, int base);
unsigned long strtoul (const char* c, char** pc, int base);
```

EJEMPLO 14.4 Conversión de cadenas a tipos numéricos

```
char *c = " -49 2332";
char **pc = (char**) malloc(1);
long n1;
unsigned long n2;
n1 = strtol (c,pc,0);
printf (" n1 = %ld\n", n1);
printf (" cadena actual %s\n", *pc);
c = *pc;
```

```
n2 = strtoul (c, pc, 10);
printf (" n2 = %lu", n2) ;
```

Ejecutando el fragmento de código se obtienen estos resultados:

```
n1 = -49
cadena actual 2332
n2 = 2332
```

La función `strtod()` convierte los dígitos de una cadena en un número real de tipo `double`. El primer argumento, en la llamada, es la cadena; el segundo argumento es de *salida*, al cual la función asigna un puntero al carácter de la cadena con el que terminó de formarse el número real. El prototipo de la función se encuentra en `stdio.h`, y es el siguiente:

```
double strtod (const char* c, char** pc);
```

EJEMPLO 14.5 *Conversión de cadenas a números reales*

El siguiente programa muestra cómo obtener todos los números reales (tipo `double`) de una cadena. Se puede observar el uso de la variable `errno` (archivo `errno.h`); la función asigna a `errno` la constante `ERANGE` si se produce un error por *overflow* (desbordamiento) al convertir la cadena, y entonces devuelve la constante `HUGE_VAL`.

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

void main (void)
{
    char*c = "333.55553 444444.2 3e+1221";
    char **a;
    double v=0 ;

    a = (char**) malloc(1);
    v = strtod (c, a);
    if (errno != 0)
    {
        printf ("Error \"%d\" al convertir la cadena.", errno);
        exit (-1);
    }

    printf ("c = [%s], v = %lf\n", c, v);
    while ((*a) != '\0')
    {
        c = *a;
        v = strtod (c, a);
        if (errno != 0)
        {
            printf ("Error \"%d\" al convertir la cadena.", errno);
            exit(-1);
        }
        printf("c = [%s], v = %lf\n", c, v);
    }
}
```

PROBLEMAS PROPUESTOS

- 14.1.** *Se quiere leer del dispositivo estándar de entrada los n códigos de asignaturas de la carrera de Sociología. Escribe un segmento de código para realizar este proceso.*

Análisis del problema

Suponer que el número N de asignaturas está definido en una macro de la siguiente manera:

```
#define N 10
```

Cada asignatura tiene un código de cinco caracteres alfanuméricos, por lo cual se guardan en una cadena de caracteres.

Codificación

```
int i;
char asignatura [N][20], codigo [N][6];
for (i = 0; i <= N; i++)
{
    printf ("\n\tEscriba el nombre de la asignatura: ");
    gets (asignatura [i]);
    printf ("\n\tEscriba el código de la asignatura: ");
    gets (codigo [i]);
}
```

- 14.2.** *Para entrada de cadenas de caracteres, qué diferencia existe entre `scanf("%s",cadena)` y `gets(cadena)`. ¿En qué casos será mejor utilizar una u otra?*

Análisis del problema

`scanf()` limita las variables que reconoce en la entrada por medio de los espacios en blanco que las separan; por tanto, no es capaz de reconocer una línea que contenga espacios en blanco, porque para esta función cada palabra es una cadena diferente. Por tanto, si se va a leer una cadena de caracteres que contenga espacios en blanco ha de hacerse con `gets()`. Por otro lado `gets()` tiene el peligro de que aparentemente tiene un uso más sencillo que `scanf()` pero si no se le proporciona una cadena de caracteres como argumento puede que no almacene correctamente la entrada.

- 14.3.** *Define un array de cadenas de caracteres para poder leer un texto compuesto por un máximo de 80 líneas. Escribe una función para leer el texto; la función debe tener dos argumentos, uno el texto y el segundo el número de líneas.*

Análisis del problema

Una de las ventajas de trabajar con punteros es poder reservar memoria dinámicamente, es decir, en tiempo de ejecución para las variables necesarias. En este caso si se reservase un array entero de 80 posiciones por línea, se desperdiciaría todo el espacio sobrante de las cadenas cuya longitud no llegase a 79 caracteres. Manejando punteros y reserva dinámica de memoria se puede leer cada línea en un buffer temporal, para averiguar su longitud final; luego reservar tanta memoria como sea precisa para la nueva línea, hacer que el puntero de la línea apunte a la primera posición de la memoria recién asignada y por último, copiar la línea de texto desde el buffer a la nueva zona apuntada por el puntero de la línea.

El array necesario para manejar 80 líneas, implicará, por consiguiente, 80 punteros, por lo que su declaración podrá ser la siguiente:

```
char *texto[80];
```

Codificación

```

leerTexto (char**texto, int nlineas)
{
    int i;
    char buffer [80];
    texto = (char**) malloc (nlineas * sizeof (char*));
    for (i=0; i < nlineas ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto [i], buffer);
    }
}

```

- 14.4.** *Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.*

Análisis del problema

Se solicita que la función devuelva tres valores y una función en C sólo puede devolver un valor, se opta por hacer que la función tenga tres parámetros más de «salida». Con la semántica del paso por referencia, se pasan tres punteros a las variables que la función puede modificar para dejar los tres resultados que se piden, que no son más que tres contadores.

Para averiguar el tipo del carácter, recorrer la cadena por medio de un puntero auxiliar y comparar su código ASCII con el de los números y las letras.

Codificación

```

cuentaLetras (char* cadena, int *vocales, int *consonantes, int
               *digitos)
{
    char* p = cadena;
    while (*p != '\0')
    {
        if ((*p >= 'a' && *p <= 'Z') || ((*p >= 'a' && *p <= 'z'))
            switch (*p)
            {
                case 'a': case 'A':
                case 'e': case 'E':
                case 'i': case 'I':
                case 'o': case 'O':
                case 'u': case 'U': (*vocales) ++;
                default : (*consonantes) ++;
            }
        if (*p >= '0' && *p <= '9') (*digitos)++;
        pp++;
    }
}

```

- 14.5.** *¿Qué diferencias y analogías existen entre las variables c1, c2, c3? La declaración es:*

```

char**c1;
char*c2[10];
char*c3[10][21];

```

Análisis del problema

La variable `c1` es un puntero que puede apuntar a un puntero a caracteres, pero no está inicializado con una dirección válida. La variable `c2` es un array de 10 punteros a caracteres, pero estos 10 punteros no apuntan a ningún dato válido. La variable `c3` es una matriz con espacio para 210 punteros a caracteres no inicializados, accesibles según un arreglo de 10 filas de 21 elementos cada una de ellas.

- 14.6.** *Escribe una función que obtenga una cadena del dispositivo de entrada, de igual forma que `char* gets(char*)`. Utilizar para ello `getchar()`.*

Análisis del problema

La función `gets()` de la biblioteca estándar lee caracteres de la entrada estándar, normalmente el teclado, hasta que se le introduce un salto de línea. Los caracteres que recibe son colocados en un buffer interno local y en la dirección donde indique el argumento de la función, en caso de que disponga de uno. Esta es la razón por la que es conveniente utilizar `gets()` siempre con un argumento que sea un puntero y apunte a una dirección de la memoria correcta, porque `gets()` no «mira» donde coloca los caracteres, sino que se fía del puntero que se le pasa. Tampoco se puede contar con la dirección que devuelve, porque al apuntar a un *buffer* interno, no lleva la cuenta cuando hay llamadas sucesivas, de colocar las cadenas que lee en lugares diferentes de la memoria y los resultados pueden ser desagradables.

Codificación

```
char* gets2 (char* cadena)
{
    char c, *p = cadena;
    while ((( c = getchar( )) != EOF) || (c != '\n'))
        *p++ = c;
    *p = '\0';
    return cadena;
}
```

- 14.7.** *Escribir una función que obtenga una cadena del dispositivo estándar de entrada. La cadena termina con el carácter de fin de línea, o bien cuando se han leído `n` caracteres. La función devuelve un puntero a la cadena leída, o EOF si se alcanzó el fin de fichero. El prototipo de la función debe de ser:*

```
char* lee_linea(char*c, int n);
```

Análisis del problema

Como en el ejercicio anterior la función lee carácter a carácter de la entrada estándar y lo va colocando en la posición de memoria a la que apunta el primer argumento, sin mirar si la dirección que recibe es una dirección verdaderamente libre.

Codificación

```
char* lee_linea(char*c, int n)
{
    char ch, *cc = c;
    if (( ch = getchar()) == EOF) return (EOF);
    else *cc++ = ch;
    while ((( ch = getchar( )) != '\n') || (cc - c < n))
        *cc++ = ch;
    *cc = '\0';
    return c;
}
```


- 14.8.** *Escribir un programa que lea un texto de cómo máximo 60 líneas, cada línea con un máximo de 80 caracteres. Una vez leído el texto intercambiar la línea de mayor longitud por la línea de menor longitud.*

Análisis del problema

Como el texto se maneja a partir de un array de punteros a caracteres que apuntan a cada una de las líneas, el manejo del texto se hace verdaderamente sencillo. Solamente hay que tratar cada puntero como la línea a la cual apunta, pues todas las funciones de gestión de cadenas esperan como argumento precisamente un puntero con la dirección del primer carácter de la cadena.

Codificación

```
main( )
{
    char *texto[60];
    int i, lmax, posmax, lmin, posmin;
    char buffer [80];
    for (i=0; i < 60 ; i++)
    {
        gets (buffer);
        if ( strlen (buffer) == 0) break;
        if (strlen (buffer) < lmin)
        {
            posmin = i;
            lmin = strlen (buffer);
        }
        if (strlen (buffer) > lmax)
        {
            posmax = i;
            lmax = strlen (buffer);
        }
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto [i], buffer);
    }
    strcpy (buffer, texto[posmin]);
    strcpy (texto[posmin], texto[posmax]);
    strcpy (texto[posmax], buffer);
}
```

- 14.9.** *Escribir un programa que lea una línea de texto y escriba en pantalla las palabras de que consta la línea. Utilizar las funciones de `string.h`.*

Análisis del problema

El trabajo se realiza en realidad por la función `strtok()`, que utiliza un puntero interno para recorrer la cadena que se le pasa como argumento la primera vez y se va parando en cada una de las ocurrencias de los separadores proporcionados.

Codificación

```
main( )
{
    char cad[80];
    char*separador = " ";
```

```

char*ptr = cad;
gets (cad);
printf("\n%s\n",cad);
ptr = strtok(cad, separador);
printf("\tSe rompe en las palabras");
while (ptr)
{
    printf("\n%s",ptr);
    ptr = strtok(NULL, separador);
}
}

```

14.10. *Se tiene un texto formado por un máximo de 30 líneas, del cual se quiere saber el número de apariciones de la palabra clave CLAVE. Escribir un programa que lea el texto y la palabra CLAVE y, determine el número de apariciones de CLAVE en el texto.*

Análisis del problema

Se trata de realizar una búsqueda con `strstr()` en cada una de las líneas del texto individualmente, accediendo a las mismas por medio de los punteros que apuntan hacia ellas.

Codificación

```

main( )
{
    char* texto[30], buffer [80], clave[15];
    int i, veces, *ptr;
    puts (" Introduzca la palabra clave a buscar: ");
    gets (clave);
    for (i=0; i < 30 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto [i], buffer);
        ptr = texto[i];
        while ((ptr = strstr (ptr, clave)) != NULL) veces++;
    }
    printf ("La palabra clave %s aparece %d veces en el texto.\n", clave, veces);
}

```

14.11. *Se tiene un texto de 40 líneas. Las líneas tienen un número de caracteres variable. Escribir un programa para almacenar el texto en una matriz de líneas, ajustada la longitud de cada línea al número de caracteres. El programa debe leer el texto, almacenarlo en la estructura matricial y escribir por pantalla las líneas en orden creciente de su longitud.*

Análisis del problema

Como en C las cadenas de caracteres no guardan información acerca de su propia longitud, puesto que la marca de final de línea, `'\0'`, es suficiente para determinar su extensión, se usa un array auxiliar para guardar la longitud de cada línea y la posición inicial que tienen en el texto. Después se ordena tal array por longitud de línea, tal y como se pide, y sólo resta formar otro texto con las líneas del original en las posiciones que indica el array de longitudes. Se trata en definitiva de manipular simplemente los punteros del texto, puesto que las propias líneas no tienen por qué ser trasladadas.

Codificación

```

main( )
{
    char* texto[40], buffer [80];
    int i, longlin[40][2];
    puts (" Introduzca el texto línea a línea. \n ");
    for (i=0; i < 40 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto[i], buffer);
        longlin [i][0] = strlen (buffer ) +1;
        longlin [i][1] = i;
    }
    ordenar (longlin);
    for (i=0; i < 40 ; i++)
        puts (texto[ longlin[i][1]]);
}

```

14.12. *Escribir un programa que lea líneas de texto, obtenga las palabras de cada línea y las escriba en pantalla en orden alfabético. Se puede considerar que el máximo número de palabras por línea es 28.*

Análisis del problema

Este ejercicio es sencillo puesto que la información que se solicita está dentro de cada línea por separado. Se trata de acceder a cada línea por medio del puntero donde se guardan y dividirse en palabras, guardarlas en una matriz auxiliar, ordenarlas y mostrarlas ordenadas.

Codificación

```

main( )
{
    char* texto[100], buffer [80], palabras[28][20], *ptr;
    int i, j;
    puts (" Introduzca el texto línea a línea. \n ");
    for (i=0; i < 100 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto[i], buffer);
        ptr = strtok(texto[i], " ");
        j = 0;
        while (ptr)
        {
            strcpy (palabras [j++], ptr);
            ptr = strtok(NULL, " ");
        }
        ordenar (palabras);
        for (i=0; i < 28 ; i++)
        {
            puts ( palabras[i]);
            palabras[i] [0] = '\0';
        }
    }
}

```

14.13. *Se quiere leer un texto de como máximo 30 líneas y que ese texto se muestre de tal forma que aparezcan las líneas en orden alfabético.*

Codificación

```
main( )
{
    char* texto[30], buffer [80];
    int i;
    puts (" Introduzca el texto línea a línea .\n ");
    for (i=0; i < 30 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto[i], buffer);
    }
    ordenar (texto);
    for (i=0; i < 30 ; i++)
        puts (texto[i]);
}
```

14.14. *Se sabe que en las líneas que forman un texto hay valores numéricos enteros, representan los kg de patatas recogidos en una finca. Los valores numéricos están separados de las palabras por un blanco, o el carácter fin de línea. Escribir un programa que lea el texto y obtenga la suma de los valores numéricos.*

Análisis del problema

La función estándar de conversión de cadenas en enteros, `atoi()`, devuelve 0 si no encuentra dígitos en la cadena que se le pasa como argumento. Utilizando esta característica se puede separar en palabras las líneas del texto y aplicar a cada palabra la función `atoi()`, si encuentra un número devolverá el número de kilos.

Codificación

```
main( )
{
    char* texto[100], buffer [80], *ptr;
    int i, kilos, suma;
    puts (" Introduzca el texto línea a línea. \n ");
    for (i=0; i < 100 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer ) +1) * sizeof (char));
        strcpy (texto[i], buffer);
        ptr = strtok(texto[i], " ");
        j = 0;
        while (ptr)
        {
            if ((kilos = atoi (ptr)) != 0)
                suma += kilos;
            ptr = strtok(NULL, " ");
        }
        printf ("La suma total de los kg. recogidos es de %d\n", suma );
    }
}
```

14.15. *Escribir un programa que lea una cadena clave y un texto de como máximo 50 líneas. El programa debe de eliminar las líneas que contengan la clave.*

Codificación

```
main( )
{
    char* texto[50], buffer [80], clave[15];
    int i;
    puts (" Introduzca la palabra clave a buscar: ");
    gets (clave);
    for (i=0; i < 50 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc ((strlen (buffer)+1)
        if (strstr (buffer, clave) == NULL)
            strcpy (texto [i], buffer);
    }
}
```

14.16. *Se quiere sumar números grandes, tan grandes que no pueden almacenarse en variables de tipo `long`. Por lo que se ha pensado en introducir cada número como una cadena de caracteres y realizar la suma extrayendo los dígitos de ambas cadenas. Hay que tener en cuenta que la cadena suma puede tener un carácter más que la máxima longitud de los sumandos.*

Análisis del problema

Las condiciones del enunciado indican que no se puede convertir las cadenas en enteros, sumar los enteros y convertir el resultado de vuelta en cadena. Así que con las cadenas tal cual se dan y carácter a carácter se convierten en dígitos y se suman y se convierte la suma en un dígito del resultado, teniendo cuidado, claro está, en que si el resultado es mayor que 9 tiene que haber arrastre sobre el dígito siguiente (en realidad el anterior en la cadena), puesto que en cada carácter solamente se puede representar un dígito del 0 al 9.

Para facilitar el diseño del algoritmo, como los números se introducen como se escriben, desde las cifras más significativas a las menos, se da la vuelta en la cadena, porque se suma al revés y así se hace que la suma se realice en el mismo sentido del avance corriente de los índices de los arrays, esto es, en sentido creciente.

Codificación

```
char* leerGrandes (char* num1, char* num2)
{
    /* para alinear los números dar la vuelta a las cadenas
    487954558      855459784
    +   235869      + 968532
    -----
    488190427      724091884
    así se pueden sumar los dígitos en el sentido del array */
    char* rnum1, * rnum2, *result;
    int i, mayor;
    rnum1 = strrev (num1);
    rnum2 = strrev (num2);
    mayor = strlen (num1) > strlen (num2) ? strlen (num1) : strlen (num2);
    result = (char*) malloc ((mayor +2) * sizeof (char));
    for (i=0; i<=mayor; i++)
        result[i] = *rnum1++ + *rnum2++;
```

```

for (i=0; i<=mayor; i++) /* cálculo del arrastre */
    if (result[i] > 10)
    {
        result[i+1] += (int) result[i] / 10;
        /* división entera */
        result[i] %= 10;
    }
return (strrev (result));
}

```

14.19. *Un texto está formado por líneas de longitud variable. La máxima longitud es de 80 caracteres. Se quiere que todas las líneas tengan la misma longitud, la de la cadena más larga. Para ello se debe cargar con blancos por la derecha las líneas hasta completar la longitud requerida. Escribir un programa para leer un texto de líneas de longitud variable y formatear el texto para que todas las líneas tengan la longitud de la máxima línea.*

Análisis del problema

Como en los ejercicios anteriores se trata línea por línea a partir de sus punteros. En cada línea se obtiene su longitud y se escriben blancos a continuación hasta completar la longitud de la línea más larga más el 0 final.

Codificación

```

main( )
{
    char* texto[100], buffer [80], *ptr;
    int i, j, mayor;
    puts (" Introduzca el texto linea a linea.\n ");
    for (i=0; i < 100 ; i++)
    {
        gets (buffer);
        texto [i] = (char*) malloc (80 * sizeof (char));
        strcpy (texto[i], buffer);
        if (mayor < strlen (buffer))
            mayor = strlen (buffer);
    }
    /* rellenado con blancos */
    for (i=0; i < 100 ; i++)
    {
        texto[i]=realloc(texto[i], mayor+1);
        for (j= strlen (texto[i]); j < mayor; j++)
            *(texto[i] + j) = ' ';
        texto[i][mayor+1] = '\0';
    }
}

```

14.20. *Escribir un programa que encuentre dos cadenas introducidas por teclado que sean anagramas. Se considera que dos cadenas son anagramas si contienen exactamente los mismos caracteres en el mismo o en diferente orden. Hay que ignorar los blancos y considerar que las mayúsculas y las minúsculas son iguales.*

Análisis del problema

Para averiguar si dos cadenas son anagramas, se necesita entonces saber qué letras tienen. Para eso se utiliza un array con una posición para cada letra del abecedario. Este array contendrá el número de veces que se encuentra cada carácter cuyo

código ASCII coincide con el valor de su posición en el array. Al final sólo se compara la información guardada en los arrays de cada cadena.

Codificación

```
main( )
{
    char cad1[40], cad2[40];
    char* ptr1 = cad1, *ptr2 = cad2;
    int letras1[28], letras2[28], i;
    gets (cad1);
    cad1 = tolower (cad1);
    gets (cad2);
    cad2 = tolower (cad2);
    for (i=0; i < 40; i++)
    {
        if (alpha (cad1[i]))
            letras1[cad1[i] - 'a']++;
        if (alpha (cad2[i]))
            letras1[cad2[i] - 'a']++;
    }
    for (i=0; i < (28); i++)
        if (letras1[i] != letras2[i])
            return; puts ("Las cadenas introducidas no son anagramas. \n");
    puts ("Las cadenas introducidas son anagramas. \n");
}
```

PROBLEMAS PROPUESTOS

- 14.1.** La función `atoi()` transforma una cadena formada por dígitos decimales en el equivalente número entero. Escribir una función que transforme una cadena formada por dígitos hexadecimales en un entero largo.
- 14.2.** Escribir una función para transformar un número entero en una cadena de caracteres formada por los dígitos del número entero.
- 14.3.** Escribir una función para transformar un número real en una cadena de caracteres que sea la representación decimal del número real.
- 14.4.** Escribir un programa que lea una línea de texto y escriba en pantalla las palabras de que consta la línea sin utilizar las funciones de `string.h` y particularmente sin usar `strtok()`.
- 14.5.** Escribir un programa que lea líneas de texto, obtenga las palabras que aparecen en él y las escriba en pantalla en orden alfabético, añadiendo el número de veces que aparecen.
- 14.6.** Un texto está formado por líneas de longitud variable. La máxima longitud es de 80 caracteres. Se quiere que todas las líneas tengan la misma longitud, la de la cadena más larga. Para ello se debe rellenar con blancos los espacios que ya existen entre las palabras. Tal relleno debe ser lo más uniforme posible para que no se note la transformación. Escribir un programa para leer un texto de líneas de longitud variable y formatear el texto para que todas las líneas tengan la longitud de la máxima línea.
- 14.7.** Un sistema de cifrado simple consiste en sustituir cada carácter de un mensaje por el carácter que está situado a tres posiciones alfabéticas por delante suyo. Escribir una función que tome como parámetro una cadena y devuelva otra cifrada como se ha explicado.

- 14.8.** Otro sistema de encriptación consiste en sustituir cada carácter del alfabeto por otro decidido de antemano, pero siempre el mismo. Utilizar este método en una función que tome como parámetros el mensaje a cifrar y una cadena con las correspondencias ordenadas de los caracteres alfabéticos. La función devolverá un puntero a la cadena cifrada del mensaje.
- 14.9.** Escribir una función que cada vez que se le llame genere un código alfanumérico diferente, devolviéndolo en forma de cadena. El argumento de dicha función es el número de caracteres que va a tener el código generado.
- 14.10.** Escribir un programa que tome como entrada un programa escrito en lenguaje C de un fichero de texto y compruebe si los comentarios están bien escritos. Es decir, se trata de comprobar si después de cada secuencia `'/*'` existe otra del tipo `'*/'`, recordando que no se pueden anidar comentarios.
- 14.11.** Escriba una función que reciba una palabra y genere todas las palabras que se pueden construir con sus letras.

Entrada y salida por archivos

Hasta este momento se han realizado las operaciones básicas de entrada y salida. La operación de introducir (*leer*) datos en el sistema se denomina **lectura** y la generación de datos del sistema se denomina **escritura**. La lectura de datos se realiza desde su teclado e incluso desde su unidad de disco, y la escritura de datos se realiza en el monitor y en la impresora de su sistema.

Las funciones de entrada/salida no están definidas en el propio lenguaje C, sino que están incorporadas en cada compilador de C bajo la forma de *biblioteca de ejecución*. En C existe la biblioteca `stdio.h` estandarizada por ANSI; esta biblioteca proporciona tipos de datos, macros y funciones para acceder a los archivos. El manejo de archivos en C se hace mediante el concepto de **flujo** (*streams*) o canal, o también denominado secuencia. Los flujos pueden estar abiertos o cerrados, conducen los datos entre el programa y los dispositivos externos. Con las funciones proporcionadas por la biblioteca se pueden tratar archivos secuenciales, de acceso directo, archivos indexados...

En este capítulo aprenderá a utilizar las características típicas de E/S para archivos en C, así como las funciones de acceso más utilizadas.

15.1 Flujos

Un **flujo** (*stream*) es una abstracción que se refiere a un *flujo* o *corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o canal por la cual circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene al archivo. Hay tres flujos o canales abiertos automáticamente:

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

Estas tres variables se inicializan al comenzar la ejecución del programa y todas ellas admiten secuencias de caracteres en modo texto. Tienen el siguiente cometido:

<code>stdin</code>	asocia la entrada estándar (teclado) con el programa.
<code>stdout</code>	asocia la salida estándar (pantalla) con el programa.
<code>stderr</code>	asocia la salida de mensajes de error (pantalla) con el programa.

El acceso a los archivos se hace con un *buffer* intermedio. Se puede pensar en el *buffer* como un array donde se van almacenando los datos dirigidos al archivo, o desde el archivo; el *buffer* se vuelca cuando de una forma u otra se da la orden de vaciarlo.

Por ejemplo, cuando se llama a una función para leer del archivo una cadena, la función lee tantos caracteres como quepan en el *buffer*. Luego, la primera cadena del *buffer* es la que se obtiene; una siguiente llamada a la función obtendrá la siguiente cadena del *buffer*, así hasta que se quede vacío y sea llenado con una posterior llamada a la función de lectura.

15.2 Apertura de un archivo

Para comenzar a procesar un archivo en C la primera operación a realizar es abrir el archivo. La apertura del archivo supone conectar el archivo externo con el programa, e indicar cómo va a ser tratado el archivo: binario, texto. El programa accede a los archivos a través de un puntero a la estructura `FILE`, la función de apertura devuelve dicho puntero.

La función para abrir un archivo es `fopen()`; el formato de llamada:

```
FILE *fopen(char *nombre_archivo, char *modo);
nombre      ≡ cadena      Contiene el identificador externo del archivo.
modo        ≡ cadena      Contiene el modo en que se va a tratar el archivo.
```

La función puede detectar un error al abrir el archivo, por ejemplo que el archivo no exista y se quiera leer, entonces devuelve `NULL`.

`fopen()` espera como segundo argumento el modo de tratar el archivo. Fundamentalmente se establece si el archivo es para leer, para escribir o para añadir; y si es de texto o binario. Los modos básicos se expresan como Modo en la Tabla 15.1. A éstos se añade la *t* para modo texto, la *b* para modo binario.

Tabla 15.1 Modos de apertura de un archivo

Modo	Significado
"r"	Abre para lectura.
"w"	Abre para crear nuevo archivo (si ya existe se pierden sus datos).
"a"	Abre para añadir al final.
"r+"	Abre archivo ya existente para modificar (leer/escribir).
"w+"	Crea un archivo para escribir/leer (si ya existe se pierden los datos).
"a+"	Abre el archivo para modificar (escribir/leer) al final. Si no existe es como <code>w+</code> .

Al terminar la ejecución del programa podrá ocurrir que haya datos en el buffer de entrada/salida, si no se volcasen en el archivo quedaría éste sin las últimas actualizaciones. Siempre que se termina de procesar un archivo y siempre que se termine la ejecución del programa los archivos abiertos hay que cerrarlos para que entre otras acciones se vuelque el buffer.

La función `fclose()` cierra el archivo asociado al *puntero_file*, devuelve `E0F` si ha habido un error al cerrar. El prototipo es:

```
int fclose(FILE* puntero_file);
```

15.3 Funciones de lectura y escritura

Las funciones `putc()` y `fputc()` son idénticas, `putc()` está definida como macro. Escriben un carácter *c* en el archivo asociado con el puntero a `FILE`. Devuelven el carácter escrito, o bien `E0F` si no puede ser escrito. El formato de llamada es:

```
putc(c, puntero_archivo);
fputc(c, puntero_archivo);
```

Las funciones `getc()` y `fgetc()`, leen un carácter (el siguiente carácter) del archivo asociado al puntero a `FILE`. Devuelven el carácter leído o `E0F` si es fin de archivo (o si ha habido un error). El prototipo de ambas funciones es el siguiente:

```
int getc(FILE* pf);
int fgetc(FILE* pf);
```

La función `fputs()` escribe una cadena de caracteres. La función devuelve `EOF` si no ha podido escribir la cadena, un valor no negativo si la escritura es correcta; el formato de llamada es:

```
fputs(cadena, puntero_archivo);
```

La función `fgets()` lee una cadena de caracteres del archivo. Termina la captación de la cadena cuando lee el carácter de fin de línea, o bien cuando ha leído `n-1` caracteres, siendo `n` un argumento entero de la función. La función devuelve un puntero a la cadena devuelta, o `NULL` si ha habido un error. El formato de llamada es:

```
fgets(cadena, n, puntero_archivo);
```

Las funciones `printf()` y `scanf()` permiten escribir o leer variables de cualquier tipo de dato estándar; los códigos de formato (`%d`, `%f` ...) indican a C la transformación que debe de realizar con la secuencia de caracteres (conversión a entero ...). La misma funcionalidad tienen `fprintf()` y `fscanf()` con los flujos a que se aplican. Estas dos funciones tienen como primer argumento el puntero asociado al archivo de texto. El prototipo de ambas funciones es el siguiente:

```
int fprintf(FILE* pf, const char* formato, . . .);
int fscanf(FILE* pf, const char* formato, . . .);
```

La función `feof()` devuelve un valor distinto de 0 (`true`) cuando se lee el carácter de fin de archivo, en caso contrario devuelve 0 (`false`). El prototipo de la función es el siguiente:

```
int feof(FILE* pf);
```

Con la función `rewind()` se sitúa el puntero del archivo al inicio de éste. El prototipo:

```
void rewind(FILE* pf);
```

EJEMPLO 15.1 *Lectura y escritura en un archivo (fichero) de texto*

En un archivo (fichero) de texto la información se guarda en formas de cadenas de caracteres separadas por saltos de línea. El programa siguiente solicita una serie de líneas al usuario, las escribe en un fichero y a continuación imprime.

```
FILE *ftexto;
char linea[80];

if ((ftexto = fopen ("ejemplo.txt", "w+t")) == NULL)
    fprintf (stderr, "Error al abrir el archivo");

while (!strcmp (linea, "fin"))
{
    gets (linea);
    fputs (linea, strlen(linea) + 1, ftexto);
}
printf ("Estas han sido las líneas recibidas:\n");

while (!feof (ftexto))
{
    fgets (linea, strlen(linea) + 1, ftexto);
    puts (linea);
}
fclose (ftexto);
```

Además, C dispone de la función `fflush()` para volcar y vaciar el buffer del archivo pasado como argumento. La función devuelve 0 si no ha habido error, en caso de error devuelve la constante `EOF`. El prototipo es el siguiente:

```
int fflush (FILE* pf);
```

EJEMPLO 15.2 *Uso de fflush()*

En el siguiente fragmento se realiza una entrada de un número entero, llamando a `scanf()`, y de una cadena de caracteres, llamando a `gets()`. La llamada `fflush(stdin)` hace que se vacíe íntegramente el *buffer* de entrada, en caso contrario quedaría el carácter fin de línea y `gets()` leería una cadena vacía.

```
int cuenta;
char b[81];
...
printf ("Cantidad: ");
scanf ("%d", &cuenta);
fflush (stdin);
printf ("Dirección: ");
gets (b);
```

15.4 Archivos binarios de C

Para abrir un archivo en modo binario hay que especificar la opción `b` en el modo. Los archivos binarios son secuencias de bytes. Los archivos binarios optimizan el espacio, sobre todo con campos numéricos. Así, almacenar en modo binario un entero supone una ocupación de 2 bytes o 4 bytes (depende del sistema), y un número real 4 bytes o 8 bytes; en modo texto primero se convierte el valor numérico en una cadena de dígitos (`%d`, `%8.2f` ...) y después se escribe en el archivo.

La función `fwrite()` escribe un *buffer* de cualquier tipo de dato en un archivo binario. El prototipo de la función es:

```
size_t fwrite(const void * direccion_buffer, size_t tamaño,
              size_t num_elementos, FILE * puntero_archivo);
```

La función `fread()` lee de un archivo `n` bloques de bytes y los almacena en un *buffer*. El número de bytes de cada bloque (*tamaño*) se pasa como parámetro, al igual que la dirección del *buffer* (o variable) donde se almacena. El prototipo de la función es:

```
size_t fread(const void * direccion_buffer, size_t tamaño,
             size_t num_elementos, FILE * puntero_archivo);
```

Con la función `fseek()` se puede tratar un archivo en C como un array que es una estructura de datos de acceso aleatorio. `fseek()` sitúa el puntero del archivo en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos.

El segundo argumento de `fseek()` es el desplazamiento, el tercero es el origen del desplazamiento. El prototipo es :

```
long fseek(FILE * puntero_archivo, long desplazamiento, int origen);
```

<i>origen</i>	Posición desde la que se cuenta el número de bytes a mover. Puede tener tres valores, que son:
0	⇒ <code>SEEK_SET</code> : Cuenta desde el inicio del archivo.
1	⇒ <code>SEEK_CUR</code> : Cuenta desde la posición actual del puntero al archivo.
2	⇒ <code>SEEK_END</code> : Cuenta desde el final del archivo.

La posición actual del archivo se puede obtener llamando a la función `ftell()` y pasando un puntero al archivo como argumento. La función devuelve la posición como número de bytes (en entero largo: *long int*) desde el inicio del archivo (byte 0). El prototipo es:

```
long int ftell(FILE *pf);
```

Otra forma de conocer la *posición actual* del archivo, o bien mover dicha posición es mediante las funciones `fgetpos()` y `fsetpos()`. La función `fgetpos()` tiene dos argumentos, el primero representa al archivo (flujo) mediante el puntero `FILE` asociado. El segundo argumento de tipo puntero a `fpos_t` (tipo entero declarado en `stdio.h`) es de *salida*; la función le asigna la posición actual del archivo. La función `fsetpos()` se utiliza para cambiar la posición actual del archivo. La nueva posición se pasa como segundo argumento (de tipo `const fpos_t*`) en la llamada a la función. El primer argumento es el puntero `FILE` asociado al archivo.

La dos funciones devuelven cero si no ha habido error en la ejecución, en caso contrario devuelven un valor distinto de cero (el número del error). Sus prototipos están en `stdio.h`, son los siguientes:

```
int fgetpos (FILE* pf, fpos_t* p);
int fsetpos (FILE* pf, const fpos_t* p);
```

EJEMPLO 15.3 Lectura y escritura en un archivo binario

En un archivo se desea grabar la notas que tienen los alumnos de una asignatura junto al nombre del profesor y el resumen de aprobados y suspensos. La estructura va ser la siguiente: Primer registro con el nombre de la asignatura y curso. Segundo registro con el nombre del profesor, número de alumnos, de aprobados y suspensos. Cada uno de los alumnos, con su nombre y nota.

Se crea un archivo binario (modo `wb+`) con la estructura que se indica en el enunciado. Antes de escribir el segundo registro (profesor) se obtiene la posición actual, llamando a `fgetpos()`. Una vez que se han grabado todos los registros de alumnos, se sitúa como posición actual, llamando a `fgetpos()`, el registro del profesor con el fin de grabar el número de aprobados y suspensos. Naturalmente, según se solicitan las notas de los alumnos se contabiliza si la calificación es aprobado o suspenso. La entrada de datos se realiza desde el teclado.

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

typedef struct
{
    char asg[41];
    int curso;
} ASGTA;

typedef struct
{
    char nom[41];
    int nal, aprob, susp;
} PROFS;

typedef struct
{
    char nom[41];
    float nota;
} ALMNO;

void entrada (ALMNO* a);

void main (void)
{
    ASGTA a;
    PROFS h = {" ", 0, 0, 0}; /* valores iniciales: alumnos,   aprobados, suspensos */
    ALMNO t;
    FILE* pf;
    int i;
    fpos_t* p = (fpos_t*) malloc (sizeof(fpos_t));
```

```

pf = fopen ("CURSO.DAT", "wb+");
if (pf == NULL)
{
    printf ("Error al abrir el archivo, modo wb+");
    exit (-1);
}

printf ("Asignatura: ");
gets (a.asg);
printf ("Curso: ");
scanf ("%d%c", &a.curso);
fwrite (&a,sizeof(ASGTA), 1, pf);

printf ("Nombre del profesor: ");
gets (h.nom);
printf ("Número de alumnos: ");
scanf ("%d%c", &h.nal);
fgetpos (pf, p);                               /* guarda en p la posición actual */
fwrite (&h,sizeof(PROFS),1,pf);

for (i = 1; i <= h.nal; i++)
{
    entrada (&t);
    if (t.nota <= 4.5)
        h.susp++;
    else
        h.aprob++;
    fwrite (&t, sizeof(ALMNO), 1, pf);
}

fflush (pf);
fsetpos (pf, p);                               /*se sitúa en registro del profesor */
fwrite (&h, sizeof(PROFS), 1, pf);
fclose(pf);
}

void entrada(ALMNO* a)
{
    printf ("Nombre: ");
    gets (a -> nom);
    printf ("Nota: ");
    scanf ("%f%c", &(a -> nota));
}

```

15.5 Datos externos al programa con argumentos de `main()`

La función `main()` tiene dos argumentos opcionales: el primero es un argumento entero que contiene el número de parámetros transmitidos al programa (incluyendo el mismo nombre del programa). El segundo argumento contiene los parámetros transmitidos, en forma de cadenas de caracteres; por lo que el tipo de este argumento es un array de punteros a `char`. Puede haber un tercer argumento que contiene las variables de entorno, definido también como array. El prototipo de `main()` será:

```
int main(int argc, char*argv[]);
```

Los nombres de los argumentos pueden cambiarse.

PROBLEMAS RESUELTOS

- 15.1.** *Escribir las sentencias necesarias para abrir un archivo de caracteres cuyo nombre y acceso se introduce por teclado en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.*

Análisis del problema

Éstas son las operaciones básicas para realizar la apertura de un fichero. Una observación importante es que siempre se ha de comprobar si la apertura del archivo ha sido realizada con éxito, puesto que es una operación que realiza el sistema operativo para el programa y queda fuera de control. En caso de que la apertura no fuera correcta, es recomendable abortar la ejecución del programa para averiguar qué es lo ha podido ir mal.

ENTRADA: Nombre del archivo a abrir.

Codificación

```
FILE *fp;
char nombre[14]; /* tiene que haber espacio para el nombre completo */
printf ("Escriba el nombre del fichero: ");
gets (nombre);
if ((fp = fopen (nombre, "rt")) == NULL)
{
    puts ("Error de apertura para lectura ");
    fp = fopen (nombre, "wt");
}
```

- 15.2.** *Señale los errores del siguiente programa:*

```
#include <stdio.h>
int main( )
{
    FILE* pf;
    pf = fopen("almacen.dat");
    fputs("Datos de los alma-cenes TIES0", pf);
    fclose(pf);
    return 0;
}
```

Análisis del problema

La función `fopen()` carece de segundo argumento para indicar el modo de apertura del archivo. Tampoco se comprueba que el fichero se haya podido abrir sin errores, según el valor devuelto por la función `fopen()`.

- 15.3.** *Se tiene un archivo de caracteres de nombre «SALAS.DAT». Escribir un programa para crear el archivo «SALAS.BIN» con el contenido del primer archivo pero en modo binario.*

Análisis del problema

Observar que la diferencia externa al usar un archivo binario y otro de texto está solamente en el argumento que indica el modo de apertura, porque las operaciones de lectura y escritura se ocupan de leer o escribir la misma variable según el diferente formato: en el archivo de texto byte a byte convirtiéndolo a y de su código ASCII y en el archivo binario se vuelva de y a la memoria sin realizar ninguna transformación.

Codificación

```

main( )
{
    FILE * pft, * pfb;
    char línea[80];
    if ((pft = fopen ("SALAS.DAT", "rt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    if ((pfb = fopen ("SALAS.BIN", "wb")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof(pft))
    {
        fgets ( línea, 80, pft);
        fwrite (línea, strlen (línea) +1, 1, pfb);
    }
    fclose (pft);
    fclose (pfb);
}

```

- 15.4.** La función `rewind()` sitúa el puntero del archivo en el inicio del archivo. Escribir una sentencia, con la función `fseek()` que realice el mismo cometido.

Análisis del problema

Como `fseek()` posiciona el puntero del fichero en la posición que indican sus dos últimos argumentos, hacer una llamada que posicione en el byte 0 desde el origen del fichero.

```
fseek( puntero_archivo, 0, SEEK_SET);
```

- 15.5.** Utiliza los argumentos de la función `main()` para dar entrada a dos cadenas; la primera representa una máscara, la segunda el nombre de un archivo de caracteres. El programa tiene que localizar las veces que ocurre la máscara en el archivo.

Análisis del problema

La organización de un archivo de tipo texto es una serie de cadenas de caracteres almacenadas secuencialmente y separadas por caracteres final de línea y salto de carro, que cuando se leen en memoria se convierten en cadenas terminadas en `'\0'`, como todas las cadenas del lenguaje C. Los caracteres están almacenados en un byte cuyo contenido es el código ASCII correspondiente. Las funciones de entrada y salida, es decir, de lectura y escritura con archivos de modo texto son de dos tipos, o leen y escriben byte a byte, carácter a carácter, o leen y escriben línea a línea.

En este programa se trata de leer cada línea de un archivo de texto y buscar en ella la ocurrencia de una subcadena denominada máscara, para lo cual se utilizan las funciones estándar de C de tratamiento de cadenas.

Codificación

```

main (int argc, char **argv)
{
    FILE * pf;
    char mascara[20], nombre[14], línea[80], *ptr;

```

```

int veces, i;
if (argc != 3)
{
    printf ("Uso: programa máscara archivo.\n");
    exit (1);
}
strcpy (mascara, argv[1]);
strcpy (nombre, argv[2]);
if ((pf = fopen (nombre, "rt")) == NULL)
{
    puts ("Error de apertura ");
    exit(1);
}
while (!feof (pf))
{
    fgets ( línea, 80, pf);
    i++;
    ptr = línea;
    while (*ptr )
    {
        ptr++ = strstr (ptr, mascara);
        printf ("La mascara aparece en la línea %d \n", i);
        veces++;
    }
    printf ("La máscara aparece %d veces en el fichero. \n", veces);
    fclose (pf);
}

```

- 15.6.** *Un archivo contiene enteros positivos y negativos. Utiliza la función `fscanf()` para leer el archivo y determinar el número de enteros negativos.*

Análisis del problema

Una vez dominadas las funciones de entrada y salida por teclado y consola de la biblioteca estándar de C, es muy fácil programar con las funciones de entrada y salida para ficheros de texto, puesto que son las mismas. Así `fscanf()` es totalmente similar a `scanf()`, solamente variando en que la segunda lee siempre de la entrada estándar —teclado—, mientras que la primera puede leer de cualquier archivo abierto en modo texto.

Aunque esta operación es muy simple, es bueno entender el mecanismo de conversión que se está utilizando. El fichero abierto contiene números enteros, pero al ser un archivo de texto esos números están almacenados no de forma binaria sino como una cadena de caracteres que representan los dígitos y el signo del número en forma de secuencia de sus códigos ASCII binarios.

Esto no quiere decir que haya que leer línea a línea y en cada una de ellas convertir las secuencias de códigos de caracteres a los números enteros en binario correspondiente, para almacenarlos así en la memoria. Este trabajo es el que realiza la función `fscanf()` cuando el formato indica que lo que se va a encontrar es un entero. Es la misma operación de conversión que realiza la función `scanf()` cuando lee secuencias de códigos de teclas desde la entrada estándar.

Codificación

```

main (int argc, char **argv)
{
    FILE * pf;
    char mascara[20], nombre[14], línea[80];

```

```

int num, neg;
if ((pf = fopen ("NUMEROS.TXT", "rt")) == NULL)
{
    puts ("Error de apertura ");
    exit(1);
}
while (!feof (pf))
{
    fscanf (pf, "%d", &num);
    if (num < 0) neg++;
}
printf ("El número de enteros no negativos en el fichero es %d \n", neg);
fclose (pf);
}

```

- 15.7.** *Un archivo de caracteres quiere escribirse en la pantalla. Escribir un programa para escribir el archivo, cuyo nombre viene dado en la línea de órdenes, en pantalla.*

Análisis del problema

Los archivos de texto están creados para ser leídos en memoria línea a línea, transformándose los caracteres final de línea y salto de carro en el carácter terminado de cadena que entiende C en memoria. Por eso, para escribir el contenido de un archivo en memoria basta con leer cada línea y escribirla en pantalla con cualquiera de las funciones que lo hacen.

Codificación

```

main (int argc, char **argv)
{
    FILE * pf;
    char nombre[14], línea[80];
    if (argc != 2)
    {
        printf ("Uso: programa archivo.\n");
        exit (1);
    }
    strcpy (nombre, argv[1]);
    if ((pf = fopen (nombre, "rt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof (pf))
    {
        fgets ( línea, 80, pft);
        puts (línea);
    }
    fclose (pf);
}

```

- 15.8.** *Escribir una función que devuelva una cadena de caracteres de longitud n, del archivo cuyo puntero se pasa como argumento. La función termina cuando se han leído los n caracteres o es fin de archivo. Utilizar la función `fgetc()`. El prototipo de la función pedida es:*

```
char* leer_cadena(FILE* pf, int n);
```

Análisis del problema

La función `fgetc()` es usada para ir leyendo el fichero abierto en modo texto carácter a carácter, o lo que es lo mismo, byte a byte.

Codificación

```
char* leer_cadena(FILE* pf, int n){
{
    char cadena[n+1];
    int i;
    while (!feof(pf) || i < n)
        cadena[i++] = (char) fgetc (pf);
    cadena[i] = '\0';
    return cadena;
}
```

- 15.9.** *Se quiere concatenar archivos de texto en un nuevo archivo. La separación entre archivo y archivo ha de ser una línea con el nombre del archivo que se acaba de procesar. Escribir el programa correspondiente de tal forma que los nombres de los archivos se encuentren en la línea de órdenes.*

Análisis del problema

El segundo argumento de `main()`, llamado comúnmente `argv`, es un puntero a un array que contiene punteros que apuntan al inicio de las cadenas de caracteres donde el sistema guarda los elementos de la línea de órdenes. Como en este caso esos elementos son nombres de archivos, se van a ir tomando uno a uno desde el segundo —porque el primero siempre es el nombre del ejecutable del programa que permite su ejecución— para pasárselos a las funciones `fopen()` y así poder abrir cada archivo necesario.

Codificación

```
main (int argc, char **argv)
{
    FILE * pf, pfcat;
    char nombre[14], línea[80];
    int i = 2;
    if (argc < 2)
    {
        printf ("Uso: programa lista de archivos. \n");
        exit (1);
    }
    if ((pfcat = fopen (argv[1], "wt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    argc--;
    while (--argc > 0)
    {
        strcpy (nombre, argv[i++]);
        if ((pf = fopen (nombre, "rt")) == NULL)
        {
            puts ("Error de apertura ");
        }
    }
}
```

```

        exit(1);
    }
    fputs (nombre, pfcats);
    while (!feof (pf))
    {
        fgets ( línea, 80, pf);
        fputs (línea, pfcats);
    }
    fclose (pf);
}
fclose(pfcats);
}

```

- 15.10.** *Escribir una función que tenga como argumentos un puntero de un archivo de texto, un número de línea inicial y otro número de línea final. La función debe mostrar las líneas del archivo comprendidas entre los límites indicados.*

Análisis del problema

Por supuesto, los archivos de texto no contienen líneas numeradas, a no ser que se hayan escrito en ellos incluyendo su numeración. Pero como la lectura es secuencial, es necesario utilizar una variable entera a modo de índice o contador, para llevar en todo momento el número de líneas leídas y esto puede servir para saber cuál es la posición de cada una de ellas.

Codificación

```

mostrarLineas (FILE *pf, int inicio, int fin)
{
    char línea[80];
    int i=1;
    while (!feof (pf))
    {
        fgets (línea, 80, pf);
        if (i >= inicio && i <= fin)
            printf ("%d: %s/n", i++, línea);
    }
}

```

- 15.11.** *Escribir un programa que escriba por pantalla las líneas de texto de un archivo, numerando cada línea del mismo.*

Codificación

```

main (int argc, char **argv)
{
    FILE * pf;
    char nombre[14], línea[80];
    int i;
    if (argc != 2)
    {
        printf ("Uso: programa archivo.\n");
        exit (1);
    }
    strcpy (nombre, argv[1]);
    if ((pf = fopen (nombre, "rt")) == NULL)

```

```

    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof (pf))
    {
        fgets ( línea, 80, pf);
        printf ( "%+4d : %s\n" i++, línea);
    }
    fclose (pf);
}

```

15.12. *Escribir un programa que compare dos archivos de texto. El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.*

Análisis del problema

Si solamente se pidiera saber qué líneas son diferentes, se podría comparar cada línea con una función como `strcmp()`, pero como se indica que es necesario saber en qué caracteres difieren, hay que comparar en cada par de líneas que se leen de cada fichero cada carácter con el que está en la misma posición del otro archivo (fichero).

Codificación

```

main( )
{
    char linea1[80], linea2[80];
    int numlinea, i;
    FILE * pft, * pfb;
    if ((pf1 = fopen ("TEXTOS.DAT", "rt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    if ((pf2 = fopen ("COPIA.DAT", "rt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof(pf1) && !feof(pf2))
    {
        fgets (línea1, 80, pf1);
        fgets (línea2, 80, pf2);
        numlinea ++;
        for (i=0; i<80 || línea1[i] != línea2[i]; i++)
            if (línea1[i] != línea2[i])
            {
                printf ("\n Línea %d y columna %d:", numlinea, i);
                printf (" primer fichero -> %s segundo fichero -> %s",
                    &línea1[i], &línea2[i]);
            }
    }
    fclose(pf1);
    fclose(pf2);
}

```

15.13. *Un atleta utiliza un pulsómetro para sus entrenamientos. El pulsómetro almacena las pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos del pulsómetro por parejas: tiempo, pulsaciones.*

Análisis del problema

Los datos que hay que almacenar son de tipo `struct`, es decir definidos por usuario. Se podría convertir estos datos a cadenas y almacenarlos en un archivo de texto. Parece más directo utilizar un archivo binario, en el que se guardan los datos tal y como están en memoria. Así se ahorraría una conversión al leer y escribir. Lo único que hay que tener en cuenta es que, como los archivos binarios no tienen ninguna clase de marca sobre la estructura de lo que están almacenado en ellos, son los programas los que tendrán que leer con las mismas estructuras con que se escribieron.

Codificación

```
struct entrenamiento
{
    char fecha[11]; /* dd/mm/aaaaa */
    char hora[9] /* hh:mm:ss */
    int minutos;
} rege;
struct pulsom
{
    char hora[9] /* hh:mm:ss */
    int pulsaciones;
} regp;
main( )
{
    FILE * pf;
    if ((pf = fopen ("ENTRENAM.DAT", "wb")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    printf ("\nFecha entrenamiento : ");
    gets (rege.fecha);
    printf ("\nHora entrenamiento: ");
    gets (rege.hora);
    printf ("\nDuración en minutos del entrenamiento: ");
    scanf ("%d",&rege.minutos);
    fwrite ( &rege, sizeof (rege), 1, pf);
    printf ("\n\n- Datos pulsómetro -");
    for (m = 0; m < rege.minutos * 60 /15; m++)
    {
        printf ("\nHora : ");
        gets (regp.hora);
        printf ("\nPulsaciones : ");
        scanf ("%d", &regp.pulsaciones);
        fwrite ( &regp, sizeof (regp), 1, pf);
    }
    fclose(pf);
}
```

15.14. *Se quiere obtener una estadística de un archivo de caracteres. Escribir un programa para contar el número de palabras de que consta un archivo, así como una estadística de cada longitud de palabra.*

Análisis del problema

La tabla `longitudes` tiene en cada uno de sus elementos el número de palabras con una longitud igual a su lugar en la tabla. La variable `numpalabras` contiene el número total de palabras en el archivo.

Codificación

```
main( )
{
    int longitudes[20], numpalabras;
    char línea[80];
    FILE * pf;
    if ((pf = fopen ("PRUEBA.DAT", "rt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof(pf))
    {
        fgets (línea, 80, pf);
        for (i=0; i<80 || línea[i]; i++)
        {
            if (línea[i] == ' ')
            {
                numpalabras ++;
                longitudes [longpal] ++;
                longpal =0;
            }
            else longpal++;
        }
    }
    fclose(pf);
    printf("Número de palabras y %d", numpalabras);
    for (i=0; i<20; i++)
        printf("Aparecen y %d palabras de longitud y %d", longitudes[i], i);
}
```

15.15. *En un archivo binario se encuentran pares de valores que representan la intensidad en miliamperios y el correspondiente voltaje en voltios para un diodo. Por ejemplo:*

```
0.5  0.35
1.0  0.45
2.0  0.55
2.5  0.58
. . .
```

El problema es que dado un valor del voltaje v , comprendido entre el mínimo valor y el máximo encontrar el correspondiente valor de la intensidad. Para ello el programa debe leer el archivo, formar una tabla y aplicar un método de interpolación. Una vez calculada la intensidad, el programa debe de escribir el par de valores en el archivo.

Análisis del problema

El programa va a leer los datos desde el fichero y con ellos va a formar una tabla en memoria. A partir de esta tabla se calcularán los nuevos valores y se añadirán al final del array. Una vez terminada la operación se escribe de nuevo la tabla entera en el fichero.

Codificación

```
struct pares
{
    float intensidad;
    float voltaje;
};
main( )
{
    FILE * pf;
    struct pares tabla [100], valor;
    int i, l;
    float volt;
    if ((pf = fopen ("SALAS.DAT", "r+b")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while(!feof(pf))
    {
        fread (&valor, sizeof (valor), 1, pf);
        tabla[i].intensidad = valor.intensidad;
        tabla[i++].voltaje = valor.voltaje;
    }
    printf ("\nIntroduzca un valor de voltaje: ");
    scanf ("%f", &volt);
    mAmp = interpolacion (volt, &l);
                                /* calcula el valor de la intensidad interpolado y el lugar
                                de la tabla tras el que debería colocarse */
    rewind(pf);
    for (i=0; i<=l; i++)
        fwrite (&tabla[i], sizeof (valor), 1, pf);
    valor.intensidad = mAmp;
    valor.voltaje = volt;
    fwrite (&valor, sizeof (valor), 1, pf);
    for (i=l+1; i<100; i++)
        fwrite (&tabla[i], sizeof (valor), 1, pf);
    fclose(pf);
}
```

- 15.16.** *Un profesor tiene 30 estudiantes y cada estudiante tiene tres calificaciones en el primer parcial. Almacenar los datos en un archivo, dejando espacio para dos notas más y la nota final. Incluir un menú de opciones, para añadir más estudiantes, visualizar datos de un estudiante, introducir nuevas notas y calcular nota final.*

Análisis del problema

El menú indica una manera de especificar cuál de las operaciones que permite el programa se puede activar. La primera función va a ir escribiendo las estructuras en el fichero según se va introduciendo la primera parte de las notas. Hay que recor-

dar que los archivos binarios no son más que secuencias de bits sin estructura alguna. La estructura la da el programa que define qué estructura, o estructuras, necesita definir para los datos que va a escribir y leer.

De esta forma, una vez escritos los registros con las primeras notas, no hace falta más que leerlos de nuevo para añadir más notas o calcular la nota final. Para volver a escribir la información en el disco sin tener que cargar todo en memoria y volverlo a escribir entero, como el disco permite un acceso secuencial, se puede utilizar la función `fseek()` para, una vez leído un registro y modificado en memoria, volver atrás al inicio de ese registro en el archivo y reescribirlo con la información actualizada.

Codificación

```
struct nota
{
    char nombre[40];
    int notas[5];
    int notafinal;
};

main( )
{
    int op;
    puts ("Menú del programa de gestión de notas");
    puts ("=====");
    puts ("(1) Introducir notas primer parcial.");
    puts ("(2) Introducir notas segundo parcial.");
    puts ("(3) Calcular notas finales.");
    puts ("(4) Añadir nuevos alumnos.");
    puts ("(5) Consultar notas.");
    puts ("(0) Salir.");
    puts ("Introduzca su opción.");
    puts("");
    scanf ("%d", &op);
    switch (op)
    {
        case 0: exit (1);
        case 1: primerParcial( ); break;
        case 2: segundoParcial( ); break;
        case 3: notasFinales( ); break;
        case 4: anhadir( ); break;
        case 5: consultar( ); break;
    }
}

primerParcial( )
{
    struct nota reg;
    FILE * pf;
    if ((pf = fopen ("NOTAS.DAT", "r+b")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof (pf))
    {
        fread (&reg, sizeof (reg), 1, pf);
        visualizar (reg);
    }
}
```

```

        puts ("Escriba nota primera");
        scanf ("%d", &reg.nota[0]);
        puts ("Escriba nota segunda");
        scanf ("%d", &reg.nota[1]);
        puts ("Escriba nota tercera");
        scanf ("%d", &reg.nota[2]);
        fwrite (&reg, sizeof (reg), 1, pf);
    }
    fclose(pf);
}

segundoParcial( )
{
    struct nota reg;
    FILE * pf;
    if ((pf = fopen ("NOTAS.DAT", "r+b")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof (pf))
    {
        fread (&reg, sizeof (reg), 1, pf);
        visualizar (reg);
        puts ("Escriba nota cuarta");
        scanf ("%d", &reg.nota[3]);
        puts ("Escriba nota quinta");
        scanf ("%d", &reg.nota[4]);
        fwrite (&reg, sizeof (reg), 1, pf);
    }
    fclose(pf);
}

notasFinales( )
{
    struct nota reg;
    int media;
    FILE * pf;
    if ((pf = fopen ("NOTAS.DAT", "r+b")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof (pf))
    {
        fread (&reg, sizeof (reg), 1, pf);
        visualizar (reg);
        media = (reg.notas[0] + reg.notas[1] + reg.notas[2]
                + reg.notas[3] + reg.notas[4]) / 5;
        reg.notafinal = media;
        fseek (pf, -sizeof (reg), SEEK_CUR);
        fwrite (&reg, sizeof (reg), 1, pf);
    }
}

```

```
        fclose(pf);
    }
    visualizar (struct nota r)
    {
        printf ("\t\n Nombre del Alumno : %s.", r.nombre);
        printf ("\t\n Notas : %d %d %d %d %d.", r.notas[0], r.notas[1], r.notas[2], r.notas[3],
            r.notas[4]);
        printf ("\t\n Nota final: %d\n.", r.notafinal);
    }

    anadir( )
    {
        struct nota reg;
        FILE * pf;
        if ((pf = fopen ("NOTAS.DAT", "r+b")) == NULL)
        {
            puts ("Error de apertura ");
            exit(1);
        }
        fseek (pf, 0, SEEK_END);
        printf ("\t\n Nombre del Alumno :");
        gets (reg.nombre);
        puts ("Escriba nota primera");
        scanf ("%d", &reg.nota[0]);
        puts ("Escriba nota segunda");
        scanf ("%d", &reg.nota[1]);
        puts ("Escriba nota tercera");
        scanf ("%d", &reg.nota[2]);
        fwrite (&reg, sizeof (reg), 1, pf);
        puts ("Escriba nota cuarta");
        scanf ("%d", &reg.nota[3]);
        puts ("Escriba nota quinta");
        scanf ("%d", &reg.nota[4]);
        fwrite (&reg, sizeof (reg), 1, pf);
        fclose(pf);
    }

    consultar( )
    {
        struct nota reg;
        char alumno[40];
        FILE * pf;
        if ((pf = fopen ("NOTAS.DAT", "r+b")) == NULL)
        {
            puts ("Error de apertura ");
            exit(1);
        }
        printf ("\t\n Nombre del Alumno :");
        gets (alumno);
        while (!feof (pf))
        {
            fread (&reg, sizeof (reg), 1, pf);
            if (!strcmp (reg.nombre, alumno))
```

```

        {
            visualizar (reg);
            return;
        }
    }
    puts ("Alumno no existente. \n");
    fclose(pf);
}

```

15.17. *Se quiere escribir una carta de felicitación navideña a los empleados de un centro sanitario. El texto de la carta se encuentra en el archivo CARTA.TXT. El nombre y dirección de los empleados se encuentra en el archivo binario EMPLA.DAT, como una secuencia de registros con los campos nombre y dirección. Escribir un programa que genere un archivo de texto por cada empleado, la primera línea contiene el nombre, la segunda está en blanco, la tercera la dirección y en la quinta empieza el texto CARTA.TXT.*

Análisis del problema

En este problema lo único que hay que observar con cuidado es la secuencia de operaciones, puesto que para cada lectura de datos de archivo binario hay que copiar el archivo de texto en el archivo resultado. También hay que fijarse en que hay que operar de forma diferente los dos archivos pues son de tipos diferentes. El archivo binario posee datos que son cadenas de caracteres, pero al estar almacenados de forma binaria hay que leerlos como tales, a partir de su estructura.

Codificación

```

struct empleado
{
    char nombre[40];
    char direccion [40];
};

main( )
{
    struct empleado reg;
    FILE * pf, *pfc, *pff;
    if ((pfc = fopen ("CARTA.TXT", "rt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    if ((pf = fopen ("EMPLA.DAT", "rb")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    if ((pff = fopen ("CARTAS.TXT", "wt")) == NULL)
    {
        puts ("Error de apertura ");
        exit(1);
    }
    while (!feof (pf))
    {
        fread (&reg, sizeof (reg), 1, pf);
        fputs (reg.nombre, pff);
    }
}

```

```

    fputs (" ", pff);
    fputs (reg.direccion, pff);
    fputs (" ", pff);
    fputs (" ", pff);
    while (!feof(pfc))
    {
        fgets (línea, 80, pfc);
        fputs (línea, pff);
    }
}
fclose(pf);
fclose(pff);
fclose(pfc);
}

```

15.18. *Se quiere crear un archivo binario formado por registros que representan productos de perfumería. Los campos de cada registro son código de producto, descripción, precio y número de unidades. La dirección de cada registro viene dada por una función hash que toma como campo clave el código del producto (tres dígitos):*

hash(clave) = (clave modulo 97) + 1

El número máximo de productos distintos es 100. Las colisiones, de producirse, se situarán secuencialmente a partir del registro número 120.

Análisis del problema

En un archivo binario en el que se escriben estructuras o registros de un único tipo, se puede considerar que esas estructuras están numeradas, como lo estarían si estuviesen en un array en memoria. El lugar de cada estructura viene dado por su posición desde el comienzo del archivo. El primer registro sería el cero siempre y así sucesivamente. Buscar el registro a partir de su número de índice es tan sencillo como calcular el número del byte donde comienza dicho registro. Este byte se calcula multiplicando el número del índice del registro por los bytes que ocupa cada registro. Una vez localizado el byte, sólo resta pasarlo a la función `fseek()` para que posicione el puntero de lectura y escritura del archivo en él. Las operaciones de lectura o escritura siempre se realizan desde la posición del puntero, avanzándole en un número de bytes igual al tercer parámetro de las llamadas `fread()` o `fwrite()` o igual al número de bytes leídos, en caso de que éste fuera menor.

En este problema se obtiene el número del registro, o lo que es lo mismo, la posición del archivo en la que va a ser escrito a partir de una función que parte del contenido del propio registro. Ésta es la función llamada *hash* que permite también al contrario encontrar la posición de un registro a partir de su contenido.

Codificación

```

struct producto
{
    int codigo;
    char descripcion[80];
    float precio;
    int unidades;
};
struct producto reg;
int hash;

main( )
{
    int colisiones = 120;
    FILE * pf;
    if ((pf = fopen ("PERFUM.DAT", "r+b")) == NULL)

```

```

    {
        puts ("Error de apertura ");
        exit(1);
    }
    printf ("\n\tCódigo de producto: ");
    scanf ("%d", &reg.codigo);
    printf ("\n\t Descripción del producto: ");
    gets (reg.descripcion);
    printf ("\n\t Precio: ");
    scanf ("%f", &reg.precio);
    printf ("\n\t Número de unidades: ");
    scanf ("%d", &reg.unidades);
    hash = reg.codigo % 97 +1;
    fseek (pf, hash * sizeof (reg), SEEK_SET);
    fread ( &reg, sizeof (reg), 1, pf);
    if (reg.codigo == 0)
        fwrite ( &reg, sizeof (reg), 1, pf);
    else
    {
        fseek (pf, colisiones++ * sizeof (reg), SEEK_SET);
        fwrite ( &reg, sizeof (reg), 1, pf);
    }
    fclose(pf);
}

```

15.19. *Modificar el problema 15.13 para añadir un menú con opciones de añadir al archivo nuevos entrenamientos, obtener el tiempo que se está por encima del umbral aeróbico (dato pedido por teclado) para un día determinado y media de las pulsaciones.*

Análisis del problema

En este programa cada función abre el fichero, lee todo su contenido, realiza las operaciones correspondientes y lo cierra. Otra manera de hacerlo podría haber sido abrir el fichero una vez en la función principal y pasar el puntero al fichero a cada una de las funciones para que operen sobre él, sea como parámetro o como una variable global al programa.

Codificación (Consultar la página web del libro)

15.20. *Un archivo de texto consta en cada línea de dos cadenas de enteros separadas por el operador +, o −. Se quiere formar un archivo binario con los resultados de la operación que se encuentra en el archivo de texto.*

Análisis del problema

Como de antemano se conoce el formato de las líneas que componen el fichero de texto y además es necesario hacer una conversión de caracteres a enteros, se puede utilizar la función `fscanf()` para realizar ambas tareas: la lectura y la conversión, tras proporcionarles el formato concreto que va a encontrar en cada línea que lea.

Codificación

```

main( )
{
    char linea[80];
    FILE * pf, *pfr;
    int res, val1, val2;

```

```

if ((pf = fopen ("VALORES.TXT", "rt")) == NULL)
{
    puts ("Error de apertura ");
    exit(1);
}
if ((pfr = fopen ("RESULT.DAT", "wb")) == NULL)
{
    puts ("Error de apertura ");
    exit(1);
}
while(!feof(pf))
{
    fgets ( línea, 80, pf);
    if (strchr (línea, '+'))
    {
        fscanf (pf, "%d+%d\n", &val1, &val2);
        res = val1 + val2;
        fwrite (&res, sizeof (int), 1, pfr);
    }
    if (strchr (línea, '-'))
    {
        fscanf (pf, "%d-%d\n", &val1, &val2);
        res = val1 - val2;
        fwrite (&res, sizeof (int), 1, pfr);
    }
}
fclose(pf);
fclose(pfr);
}

```

PROBLEMAS PROPUESTOS

- 15.1.** Las funciones `fgetpos()` y `fsetpos()` devuelven la posición actual del puntero del archivo, y establecen el puntero en una posición dada. Escribir las funciones

`pos_actual()` y `mover_pos()`, con los prototipos:

```

int pos_actual(FILE* pf, long* p);
int mover_pos(FILE* pf, const long* p);

```

La primera función devuelve en `p` la posición actual del archivo. La segunda función establece el puntero del archivo en la posición `p`.

- 15.2.** Un atleta utiliza un pulsómetro para sus entrenamientos. El pulsómetro almacena las pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro

contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos del pulsómetro por parejas: tiempo, pulsaciones. El archivo donde se almacene la información ha de ser de tipo texto.

- 15.3.** Escribir un programa para listar el contenido de un determinado subdirectorio, pasado como parámetro a la función `main()`.

- 15.4.** Escribir un programa que gestione una base de datos con los registros de una agenda de direcciones y teléfonos. Cada registro debe tener datos sobre el nombre, la dirección, el teléfono fijo, el teléfono móvil, la dirección de correo electrónico de una persona. El programa debe mostrar un menú para poder añadir nuevas entradas, modificar, borrar y buscar registros de personas a partir del nombre.

PROBLEMAS PROPUESTOS DE PROGRAMACIÓN DE GESTIÓN

- 15.1.** Mezclar dos archivos ordenados para producir otro archivo ordenado consiste en ir leyendo un registro de cada uno de ellos y escribir en otro archivo de salida el que sea menor de los dos, repitiendo la operación con el registro no escrito y otro leído del otro archivo, hasta que todos los registros de los dos archivos hayan sido leídos y escritos en el archivo de salida. Éste tendrá al final los registros de los dos archivos de entrada pero ordenados. Suponer que la estructura de los registros es:

```
struct articulo
{
    long clave;
    char nombre [20];
    int cantidad;
    char origen[10];
};
```

El campo clave es por el que tendrán que estar ordenados los registros.

- 15.2.** Se tiene que ordenar un archivo compuesto de registros con las existencias de los artículos de un almacén. El archivo es demasiado grande como para leerlo en memoria en un array de estructuras, ordenar el array y escribirlo al final, una vez ordenado, en el disco. La única manera es leer cada doscientos registros, ordenarlos en memoria y escribirlos en archivos diferentes. Una vez que se tienen todos los archivos parciales ordenados hay que irlos mezclando, como se indica en el ejercicio anterior, sucesivamente hasta tener un archivo con todos los registros del original, pero ordenados. Utilizar la estructura del ejercicio anterior.
- 15.3.** En un archivo de texto se conserva la salida de las transacciones con los proveedores de una cadena de tienda. Se ha perdido el listado original de proveedores y se desea reconstruirlo a partir del archivo de transacciones. Se sabe que cada línea del archivo comienza con el nombre del proveedor. Se pide escribir un programa que lea el archivo de transacciones, obtenga el nombre del proveedor con los caracteres hasta el primer espacio en blanco y muestre una lista con todos los proveedores diferentes con los que se trabaja en las tiendas.
- 15.4.** Una vez obtenida del ejercicio anterior la lista de proveedores desde el archivo de transacciones, se desea saber con qué proveedores se ha trabajado más. Para ello se sabe que en archivo de texto se apuntaba en cada línea la cantidad pagada en cada operación. Evidentemente como hay varias transacciones con un mismo proveedor es necesario acumularlas todas para saber el monto total de las transacciones por proveedor. Una vez que se tiene esta cantidad hay que escribirla en un archivo de proveedores ordenados descendentemente por la cantidad total pagada.
- 15.5.** El sistema informático de una gran superficie comercial va guardando en un archivo binario la información de las operaciones de cada una de las cajas. La información de cada operación es de tamaño variable y consiste en el número de caja, la cantidad total pagada, el modo de pago (contado o tarjeta) una lista con cada uno de los artículos comprados y su cantidad. Como cada operación tiene una lista de artículos comprados de tamaño diferente, el primer campo de cada registro guardado indica el número de artículos de la lista del registro que aparece a continuación. Escribir un programa que lea cada registro de forma adecuada y muestre en pantalla toda la información de cada operación. El tipo del registro de tamaño variable utilizado es el siguiente:

```
struct oper
{
    int numarticulos;
    int caja;
    float total;
    struct articulo
    {
        char nombre[15];
        float precio;
    } *lista;
};
```

- 15.6.** Modificar el programa del ejercicio anterior para que produzca listados por agrupados caja o por nombre de artículo.

Organización de datos en un archivo

Grandes cantidades de datos se almacenan normalmente en dispositivos de memoria externa. Este capítulo se dedica a la organización y gestión de datos estructurados sobre dispositivos de almacenamiento secundario, tales como discos magnéticos, CDs... Las técnicas requeridas para gestionar datos en archivos son diferentes de las técnicas que estructuran los datos en memoria principal, aunque se construyen con ayuda de estructuras utilizadas en memoria principal.

Los algoritmos de ordenación presentados en el capítulo 10 no se pueden aplicar si la cantidad de datos a ordenar no cabe en la memoria principal de la computadora y están en un dispositivo de almacenamiento externo. Es necesario aplicar nuevas técnicas de ordenación que se complementen con las ya estudiadas. Entre las técnicas más importantes destaca la *fusión o mezcla*. Mezclar, significa combinar dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento.

16.1 Registros

Un *archivo o fichero* es un conjunto de datos estructurados en una colección de registros, que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas *campos*. Un registro es una colección de campos lógicamente relacionados, que pueden ser tratados como una unidad por algún programa.

EJEMPLO 16.1 Definición de registro

El concepto de registro es similar al concepto de estructura (`struct`) de C. Una posible representación en C del registro libro es la siguiente:

```
struct libro
{
    char titulo[46];
    char autor [80];
    char editorial[35];
    struct fecha fechaEdicion;
    int numPags;
    long isbn;
};
```

Una clave es un campo de datos que identifica el registro y lo diferencia de otros registros. Normalmente los registros de un archivo se organizan según un campo clave. Claves típicas son números de identificación, nombres; en general puede ser una clave cualquier campo que admita *relaciones de comparación*. Por ejemplo, un archivo de libros puede estar organizado por autor, o bien por editorial, etc.

16.2 Organización de archivos

La organización de un archivo define la forma en que los registros se disponen sobre el dispositivo de almacenamiento. La organización determina cómo estructurar los registros en un archivo. Se consideran tres organizaciones fundamentales:

- Organización secuencial.
- Organización directa.
- Organización secuencial indexada.

16.2.1 ORGANIZACIÓN SECUENCIAL

Un archivo con organización secuencial es una sucesión de registros almacenados consecutivamente, uno detrás de otro, de tal modo que para acceder a un registro dado es necesario pasar por todos los registros que le preceden.

Un archivo con organización secuencial se puede procesar tanto en modo texto como en modo binario. En C para crear estos archivos se abren (`fopen()`) especificando en el argumento *modo*: “w”, “a”, “wb” o “ab”; a continuación se escriben los registros utilizando, normalmente, las funciones `fwrite()`, `fprintf()` y `fputs()`.

EJEMPLO 16.2 Operaciones con archivos secuenciales

En el primer programa se leen una serie de registros de tipo `Distrito` para guardarlos en un archivo y en el segundo programa se leen del fichero (archivo) para contabilizar los datos escritos.

```
typedef struct
{
    char candidato1[41];
    long vot1;
    char candidato2[41];
    long vot2;
    char candidato3[41];
    long vot3;
} Distrito;

char* archivo = "Petanca.dat";
FILE *pf = NULL;

void main()
{
    Distrito d;
    int termina;

    pf = fopen (archivo, "ab");
    if (pf == NULL)
    {
        puts ("No se puede crear el archivo.");
        exit (-1);
    }

    strcpy (d.candidato1,"Lis Alebuche");
    strcpy (d.candidato2,"Pasionis Cabitorihé");
```

```

strcpy (d.candidato3,"Gulius Martaria");

termina = 0;
puts ("Introducir los votos de cada candidato, termina con 0 0 0");
do {
    leeRegistro (&d);
    if ((d.vot1 == 0) && (d.vot2 == 0) && (d.vot3 == 0))
    {
        termina = 1;
        puts ("Fin del proceso. Se cierra el archivo");
    }
    else
        fwrite (&d, sizeof(Distrito), 1, pf);
} while (!termina);
fclose (pf);
}

void leeRegistro (Distrito* d)
{
    printf ("Votos para %s : ", d -> candidato1);
    scanf ("%ld", &(d -> vot1));
    printf ("Votos para %s : ", d -> candidato2);
    scanf ("%ld", &(d -> vot2));
    printf ("Votos para %s : ", d -> candidato3);
    scanf ("%ld", &(d -> vot3));
}

/*
    Código fuente del programa, cntavoto.c, que lee secuencialmente
    los registros del archivo Petanca.dat y cuenta los votos.
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "petanca.h"

void main()
{
    Distrito d;
    int votos[3] = {0,0,0};

    pf = fopen (archivo, "rb");
    if (pf == NULL)
    {
        puts ("No se puede leer el archivo.");
        exit (-1);
    }

    fread (&d, sizeof(Distrito),1, pf);
    while (!feof (pf))
    {
        votos[0] += d.vot1;
        votos[1] += d.vot2;
        votos[2] += d.vot3;
    }
}

```

```

        fread(&d, sizeof(Distrito),1, pf);
    }
    fclose (pf);

    puts("\n\tVOTOS DE CADA CANDIDATO\n");
    printf (" %s %ld: \n", d.candidato1, votos[0]);
    printf (" %s %ld: \n", d.candidato2, votos[1]);
    printf (" %s %ld: \n", d.candidato3, votos[2]);
}

```

16.2.2 ORGANIZACIÓN DIRECTA

Un archivo con organización directa (aleatoria), o sencillamente archivo directo, se caracteriza por que el acceso a cualquier registro es directo mediante la especificación de un índice, que da la posición ocupada por el registro respecto al origen del archivo.

EJEMPLO 16.3 *Tratamiento de un archivo directo*

En la sección siguiente de un programa de gestión de habitaciones en un hotel, se leen los registros utilizando una función que devuelve la situación del comienzo de cada registro en el archivo directo a partir del número de habitación:

```

#define desplazamiento(n) ((n - 1) * sizeof(Habitacion))

void entrada(void)
{
    Habitacion h;
    int encontrado, nh;
                                /* Búsqueda secuencial de primera habitación libre */
    encontrado = 0;
    nh = 0;

    if (fb == NULL) fb = fopen (fich, "rb+");
    fseek (fb, 0L, SEEK_SET);

    while ((nh < numhab) && !encontrado)
    {
        fread (&h, sizeof(h), 1, fb);
        nh++;
        if (strcmp (h.nif , "") == 0                                /* Habitación libre */
        {
            encontrado = 1;
            leerRes (&h);

            fseek (fb, desplazamiento(h.num), SEEK_SET);
            fwrite (&h, sizeof(h), 1, fb);
            puts ("Datos grabados");
        }
    }
    if (!encontrado) puts ("Hotel completo ");
    fflush (fb);
}

```

16.3 Archivos con direccionamiento *hash*

La organización directa tiene el inconveniente de que no haya un campo del registro que permita obtener posiciones consecutivas, y, como consecuencia, surgen muchos huecos libres entre registros. Entonces la organización directa necesita programar una relación entre un campo clave del registro y la posición que ocupa.

Una *función hash*, o de *dispersión*, convierte un campo clave (un entero o una cadena), en un valor entero dentro del rango de posiciones que puede ocupar un registro de un archivo.

EJEMPLO 16.4 Funciones *hash*

La clave puede ser el número de serie de un artículo (hasta 6 dígitos) y si están previstos un máximo de `tamIndex` registros, la función de direccionamiento tiene que ser capaz de transformar valores pertenecientes al rango 0 .. 999999, en un conjunto de rango 0 .. `tamIndex-1`. La clave también puede ser una cadena de caracteres, en ese caso se hace una transformación previa a valor entero.

La función *hash* mas utilizada por su sencillez se denomina *aritmética modular*. Genera valores dispersos calculando el resto de la división entera entre la clave *x* y el número máximo de registros previstos.

$x \% \text{tamIndex} =$ genera un número entero de 0 a `tamIndex-1`

La función *hash* o función de transformación debe reducir al máximo las colisiones. Se produce una colisión cuando dos registros de claves distintas, *c1* y *c2*, producen la misma dirección, $h(c1) = h(c2)$. Nunca existirá una garantía plena de que no haya colisiones, y más sin conocer de antemano las claves y las direcciones. La experiencia enseña que siempre habrá que preparar la *resolución de colisiones* para cuando se produzca alguna.

EJEMPLO 16.5 Obtención de la función "hash" más adecuada

Considerar una aplicación en la que se debe almacenar $n = 900$ registros. El campo clave elegido para dispersar los registros en el archivo es el número de identificación. Elegir el tamaño de la tabla de dispersión y calcular la posición que ocupa los registros cuyo número de identificación es: 245643, 245981 y 257135

Una buena elección, en este supuesto, de `tamIndex` es 997 al ser un número primo mayor que el número de registros que se van a grabar, 900. Se aplica la función *hash* de *aritmética modular* y se obtienen estas direcciones:

```
h(245643)= 245643 % 997 = 381
h(245981)= 245981 % 997 = 719
h(257135)= 257135 % 997 = 906
```

Para el diseño del archivo se deben considerar dos áreas de memoria externa. El área principal y el área de sinónimos o colisiones. Aproximadamente el archivo se crea con un 25 por 100 más que el número de registros necesarios. Un archivo *hash* se caracteriza por:

- Se accede a las posiciones del archivo a través del valor que devuelve una función *hash*.
- La función *hash* aplica un algoritmo para transformar uno de los campos llamado campo clave en una posición del archivo.
- El campo elegido para la función debe ser único (no repetirse) y conocido fácilmente por el usuario, porque a través de ese campo el usuario va a acceder al programa.
- Todas las funciones *hash* provocan colisiones o sinónimos. Para solucionar estas repeticiones se definen dos zonas:
 - Zona de datos o principal en la que el acceso es directo al aplicarle la función *hash*.
 - Zona de colisiones o sinónimos en la que el acceso es secuencial. En esta zona se guardan las estructuras o registros en los que su campo clave ha producido una posición repetida. Y se van colocando secuencialmente, es decir, en la siguiente posición que esté libre.

EJEMPLO 16.6 Manejo de un archivo directo con acceso por función "hash"

Los libros de una pequeña librería van a guardarse en un archivo para poder realizar accesos tan rápido como sea posible. Cada registro (libro) tiene los campos código (cadena de 6 caracteres), autor y título. El archivo debe estar

organizado como de acceso directo con transformación de claves (archivo hash), la posición de un registro se obtendrá aplicando aritmética modular al campo clave: código. La librería tiene capacidad para 190 libros.

Para el diseño del archivo se crearán 240 registros que se distribuirán de la siguiente forma:

1. Posiciones 0 - 198 constituyen el área principal del archivo.
2. Posiciones 199 - 239 constituyen el área de desbordamiento o de colisiones.

El campo clave es una cadena de 6 caracteres, que se transforma considerando que es una secuencia de valores numéricos (ordinal ASCII de cada carácter) en base 27. Por ejemplo, el código 2R545 se transforma en:

$$'2' * 27^4 + 'R' * 27^3 + '5' * 27^2 + '4' * 27^1 + '5' * 27^0$$

- En C, un carácter se representa como un valor entero que es, precisamente, su ordinal. La transformación da lugar a valores que sobrepasan el máximo entero (incluso con enteros largos), generando números negativos. No es problema, simplemente se cambia de signo.
- La creación del archivo escribe 240 registros, con el campo código == "*", para indicar que están disponibles (de baja).
- Para dar de alta un registro, primero se obtiene la posición (función *hash*); si se encuentra dicha posición ocupada, el nuevo registro deberá ir al área de colisiones (sinónimos).
- El proceso de consulta de un registro debe comenzar con la entrada del código, la transformación de la clave permite obtener la posición del registro. A continuación se lee el registro, la comparación del código de entrada con el código del registro determina si se ha encontrado. Si son distintos, se explora secuencialmente el área de colisiones.
- La baja de un registro también comienza con la entrada del código, se realiza la búsqueda, de igual forma que en la consulta, y se escribe la marca "*" en el campo código (se puede elegir otro campo) para indicar que ese hueco (registro) está libre.
- La función *hash* devuelve un número entero n de 0 a (199-1); por esa razón el desplazamiento desde el origen del archivo se obtiene multiplicando n por el tamaño de un registro.

16.4 Archivos secuenciales indexados

La guía de teléfonos es un ejemplo típico de archivo secuencial indexado con dos niveles de índices, el nivel superior para las letras iniciales y el nivel menor para las cabeceras de página. Por consiguiente, cada archivo secuencial indexado consta de un archivo de índices y un archivo de datos.

Para que un archivo pueda organizarse en forma secuencial indexada el tipo de los registros contendrá un campo clave identificador. La clave se asocia con la dirección (posición) del registro de datos en el archivo principal.

Un archivo con organización secuencial indexada consta de las siguientes partes:

- *Área de datos.* Contiene los registros de datos en forma secuencial, sin dejar huecos intercalados.
- *Área de índices.* Es una tabla que contiene la clave identificativa y la dirección de almacenamiento. Puede haber índices enlazados.

El área de índices normalmente está en memoria principal para aumentar la eficiencia en los tiempos de acceso. Ahora bien, debe haber un archivo donde guardar los índices para posteriores explotaciones del archivo de datos. Entonces al diseñar un archivo indexado hay que pensar que se manejarán dos tipos de archivos, el de datos y el de índices, con sus respectivos registros.

EJEMPLO 16.7 Proceso de un archivo secuencial indexado

Por ejemplo, si se quiere grabar los atletas federados en un archivo secuencial indexado, el campo índice que se puede elegir es el nombre del atleta (también se puede elegir el número de *carpet de federado*). Habría que declarar dos tipos de registros:

```
typedef struct
{
    int edad;
```

```

char carnet[15];
char club[29];
char nombre[41];
char sexo;
char categoria[21];
char direccion[71];
} Atleta;

typedef struct
{
    char nombre[41];
    long posicion;
} Indice;

```

Al diseñar un archivo secuencial indexado, lo primero a decidir es cuál va a ser el campo clave. Los registros han de ser grabados en orden secuencial, y simultáneamente a la grabación de los registros, el sistema crea los índices en orden secuencial ascendente del contenido del campo clave.

A continuación se desarrollan las operaciones (*altas, bajas, consultas* ...) para un archivo con esta organización. También es necesario considerar el inicio y la salida de la aplicación que procesa un archivo indexado, para cargar y descargar, respectivamente, la tabla de índices.

Los registros tratados se corresponden con artículos de un supermercado. Los campos de cada registro: *nombre del artículo, identificador, precio, unidades*. Un campo clave adecuado para este tipo de registro es el *nombre del artículo*. Para añadir registros al archivo de datos este se abre (puede que previamente se haya abierto) en *modo lectura/escritura*, se realizarán operaciones de lectura para comprobar datos. El proceso sigue estos pasos:

1. Leer el campo clave y el resto de campos del artículo.
2. Comprobar si existe, o no, en la tabla de índices. Se hace una búsqueda binaria de la clave en la tabla.
3. Si existe en la tabla: se lee el registro del archivo de datos según la dirección que se obtiene de la tabla. Puede ocurrir que el artículo, previamente, se hubiera dado de baja, o bien que se quiera re-escribir; en cualquier caso se deja elegir al usuario la acción que desee.
4. Si no existe: se graba en el siguiente registro vacío del archivo de datos. A continuación, se inserta ordenadamente en la tabla de índices el nuevo campo clave junto a su dirección en el archivo de datos.

Para dar de baja un registro (en el ejemplo, un artículo) del archivo de datos, simplemente, se marca el campo *estado* a cero que indica borrado, y se elimina la entrada de la tabla de índices. El archivo de datos estará abierto en *modo lectura/escritura*. El proceso sigue estos pasos:

1. Leer el campo clave del registro a dar de baja.
2. Comprobar si existe, o no, en la tabla de índices (*búsqueda binaria*).
3. Si existe en la tabla: se lee el registro del archivo de datos según la dirección que se obtiene de la tabla para confirmar la acción.
4. Si el usuario confirma la acción, se escribe el registro en el archivo con la marca *estado* a cero. Además, en la tabla de índices se elimina la entrada del campo clave.

La consulta de un registro (un artículo) sigue los pasos:

1. Leer el campo clave (en el desarrollo, el nombre del artículo) del registro que se desea consultar.
2. Buscar en la tabla de índices si existe, o no (*búsqueda binaria*).
3. Si existe: se lee el registro del archivo de datos según la dirección que se obtiene de la tabla para mostrar el registro en pantalla.

La operación *modificar*, típica de archivo, sigue los mismos pasos que los expuestos anteriormente. Se debe añadir el paso de escribir el registro, que se ha leído, con el campo modificado.

La primera vez que se ejecuta la aplicación se crea el archivo de datos y el de índices, cada vez que se produce un alta se graba un registro y a la vez se inserta una entrada en la tabla. Cuando se de por terminada la ejecución se gra-

bará la tabla en el archivo de índices llamando a `grabaIndice()`. Nuevas ejecuciones han de leer el archivo de índices y escribir estos en la tabla (memoria principal). El primer registro del archivo contiene el número de entradas, el resto del archivo son los índices. Como se grabaron en orden del campo clave, también se leen en orden y entonces la tabla de índices queda ordenada.

16.5 Ordenación de archivos: ordenación externa

Los algoritmos de ordenación estudiados hasta ahora utilizan arrays para contener los elementos a ordenar, por lo que es necesario que la memoria interna tenga capacidad suficiente. Para ordenar secuencias grandes de elementos que se encuentran en soporte externo (posiblemente no pueden almacenarse en memoria interna), se aplican los *algoritmos de ordenación externa*.

El tratamiento de archivos secuenciales exige que estos se encuentren ordenados respecto a un campo del registro, denominado *campo clave K*. Los distintos algoritmos de ordenación externa utilizan el esquema general de *separación y fusión* o *mezcla*. Por separación se entiende la distribución de secuencias de registros ordenados en varios archivos; por fusión la mezcla de dos o más secuencias ordenadas en una única secuencia ordenada. Variaciones de este esquema general dan lugar a diferentes algoritmos de ordenación externa.

FUSIÓN DE ARCHIVOS

La fusión o mezcla de archivos consiste en reunir en un archivo los registros de dos o mas archivos ordenados por un campo clave *T*. El archivo resultante también está ordenado por la clave *T*.

EJEMPLO 16.8 Fusión de archivos

Suponer que se dispone de dos archivos ordenados *F1* y *F2*, se desea mezclar o fundir en un sólo archivo ordenado, *F3*. Las claves son

```
F1      12  24  36  37  40  52
F2      3   8   9  20
```

Para realizar la fusión es preciso acceder a los archivos *F1* y *F2*, en cada operación sólo se lee un elemento del archivo dado. Es necesario una variable de trabajo por cada archivo (*actual1*, *actual2*) para representar el elemento actual de éste el archivo.

Se comparan las claves *actual1* y *actual2* y se sitúa la mas pequeña 3 (*actual2*) en el archivo de salida (*F3*). A continuación, se avanza un elemento el archivo *F2* y se realiza una nueva comparación de los elementos situados en las variables *actual1*.

```

      actual1
F1    [ 12 | 24 | 36 | 40 | 52 ]
F2    [ 3  | 8  | 20 ]
      actual2
F3    [ 3  ]
```

La nueva comparación sitúa la clave mas pequeña 8 (*actual2*) en *F3*. Se avanza un elemento (20) el archivo *F2* y se realiza una nueva comparación. Ahora la clave mas pequeña es 12 (*actual1*) que se sitúa en *F3*. A continuación, se avanza un elemento el archivo *F1* y se vuelve a comparar las claves *actual1*.

```

      actual1
F1    [ 12 | 24 | 36 | 40 | 52 ]
F2    [ 3  | 8  | 20 ]
      actual2
F3    [ 3  | 8  | 12 ]
```

Cuando uno u otro archivo de entrada se ha terminado, se copia el resto del archivo sobre el archivo de salida. El resultado final será:

F3	3	8	12	24	36	40	52
----	---	---	----	----	----	----	----

La codificación correspondiente de fusión de archivos (se supone que el campo clave es de tipo int) es:

```
void fusion (FILE* f1, FILE* f2, FILE* f3)
{
    Registro actual1, actual2, d;

    f3 = fopen ("fileDestino", "wt");
    f1 = fopen ("fileOrigen1", "rt");
    f2 = fopen ("fileOrigen2", "rt");

    if (f1 == NULL || f2 == NULL || f3 == NULL)
    {
        puts ("Error en los archivos.");
        exit (-1);
    }

    fread (&actual1, sizeof(Registro), 1, f1);
    fread (&actual2, sizeof(Registro), 1, f2);

    while (!feof (f1) && !feof (f2))
    {
        if (actual1.clave < actual2.clave)
        {
            d = actual1;
            fread (&actual1, sizeof(Registro), 1, f1);
        }
        else
        {
            d = actual2;
            fread (&actual2, sizeof(Registro), 1, f2);
        }
        fwrite (&d, sizeof(Registro), 1, f3);
    }

    /* Lectura terminada de f1 o f2. Se escriben los registros no procesados */
    while (!feof (f1))
    {
        fwrite (&actual1, sizeof(Registro), 1, f3);
        fread (&actual1, sizeof(Registro), 1, f1);
    }
    while (!feof (f2))
    {
        fwrite (&actual2, sizeof(Registro), 1, f3);
        fread (&actual2, sizeof(Registro), 1, f2);
    }

    fclose (f);
    fclose (f1);
    fclose (f2);
}
```

CLASIFICACIÓN POR MEZCLA DIRECTA

El método más fácil de comprender es el denominado *mezcla directa*. Utiliza el esquema iterativo de *separación y mezcla*. Se manejan tres archivos, el archivo original y dos archivos auxiliares.

El proceso consiste en:

1. Separar registros individuales del archivo original O en dos archivos $F1$ y $F2$.
2. Mezclar los archivos $F1$ y $F2$ combinando registros individuales (según sus claves) y formando pares ordenados que son escritos en el archivo O .
3. Separar pares de registros del archivo original O en dos archivos $F1$ y $F2$.
4. Mezclar $F1$ y $F2$ combinando pares de registros y formando cuádruplos ordenados que son escritos en el archivo O .

Cada separación (partición) y mezcla duplica la longitud de las secuencias ordenadas. La primera *pasada* (*separación + mezcla*) se hace con secuencias de longitud 1 y la mezcla produce secuencias de longitud 2; la segunda pasada produce secuencias de longitud 4. Cada pasada duplica la longitud de las secuencias; en la pasa n la longitud será 2^n . El algoritmo termina cuando la longitud de la secuencia supere el número de registros del archivo a ordenar.

EJEMPLO 16.9 Mezcla directa

Un archivo está formado por registros que tienen un campo clave de tipo entero. Suponiendo que las claves del archivo son:

34 23 12 59 73 44 8 19 28 51

Se van a realizar los pasos que sigue el algoritmo de mezcla directa para ordenar el archivo. Se considera el archivo O como el original, $F1$ y $F2$ archivos auxiliares.

Pasada 1

Separación :

F1: 34 12 73 8 28
F2: 23 59 44 19 51

Mezcla formando duplos ordenados:

0: 23 34 12 59 44 73 8 19 28 51

Pasada 2

Separación:

F1: 23 34 44 73 28 51
F2: 12 59 8 19

Mezcla formando cuádruplos ordenados:

0: 12 23 34 59 8 19 44 73 28 51

Pasada 3

Separación:

F1: 12 23 34 59 28 51
F2: 8 19 44 73

Mezcla formando octuplos ordenados:

0: 8 12 19 23 34 44 59 73 28 51

Pasada 4

Separación:

F1: 8 12 19 23 34 44 59 73

F2: 28 51

Mezcla con la que ya se obtiene el archivo ordenado:

0: 8 12 19 23 28 34 44 51 59 73

PROBLEMAS RESUELTOS

16.1 Codifique del algoritmo mezcla directa

Análisis del problema

La implementación del método se basa, fundamentalmente, en dos rutinas: `distribuir()` y `mezclar()`. La primera *separa* secuencias de registros del archivo original en los dos archivos auxiliares. La segunda *mezcla* secuencias de los dos archivos auxiliares y la escribe en el archivo original. Las *pasadas* que da el algoritmo son iteraciones de un bucle *mientras* longitud_secuencia *menor* numero_registros; cada iteración consiste en llamar a `distribuir()` y `mezclar()`. El número de registros del archivo se determina dividiendo `posición_fin_archivo` por `tamaño_registro`:

Codificación

```
int numeroReg (FILE* pf)
{
    if (pf != NULL)
    {
        fpos_t fin;
        fseek (pf, 0L, SEEK_END);
        fgetpos (pf, &fin);
        return fin / sizeof (Registro);
    }
    else
        return 0;
}
```

La implementación que se escribe a continuación supone que los registros se ordenan respecto de un campo clave de tipo `int`:

```
typedef int TipoClave;

typedef struct
{
    TipoClave clave;

} Registro;

void mezclaDirecta(FILE *f)
{
    int longSec;
```

```

int numReg;
FILE *f1 = NULL, *f2 = NULL;

f = fopen("fileorg","rb");

numReg = numeroReg(f);
longSec = 1;

while (longSec < numReg)
{
    distribuir(f, f1, f2, longSec, numReg);
    mezclar(f1, f2, f, &longSec, numReg);
}

void distribuir(FILE* f, FILE* f1, FILE* f2, int lonSec, int numReg)
{
    int numSec, resto, i;

    numSec = numReg/(2*lonSec);
    resto = numReg%(2*lonSec);

    f = fopen("fileorg","rb");
    f1 = fopen("fileAux1","wb");
    f2 = fopen("fileAux2","wb");

    for (i = 1; i <= numSec; i++)
    {
        subSecuencia(f, f1, lonSec);
        subSecuencia(f, f2, lonSec);
    }
    /*
    Se procesa el resto de registros del archivo
    */
    if (resto > lonSec)
        resto -= lonSec;
    else
    {
        lonSec = resto;
        resto = 0;
    }
    subSecuencia(f, f1, lonSec);
    subSecuencia(f, f2, resto);

    fclose(f1); fclose(f2);fclose(f);

}

void subSecuencia(FILE* f, FILE* t, int longSec)
{
    Registro r;
    int j;
    for (j = 1; j <= longSec; j++)

```

```
{
    fread(&r, sizeof(Registro), 1, f);
    fwrite(&r, sizeof(Registro), 1, t);
}
}

void mezclar(FILE* f1, FILE* f2, FILE* f, int* lonSec, int numReg)
{
    int numSec, resto, s, i, j, k, n1, n2;
    Registro r1, r2;

    numSec = numReg/(2*(*lonSec)); /* número de subsecuencias */
    resto = numReg%(2*(*lonSec));

    f = fopen("fileorg","wb");
    f1 = fopen("fileAux1","rb");
    f2 = fopen("fileAux2","rb");

    fread(&r1, sizeof(Registro), 1, f1);
    fread(&r2, sizeof(Registro), 1, f2);

    for (s = 1; s <= numSec+1; s++)
    {
        n1 = n2 = (*lonSec);
        if (s == numSec+1)
        { /* proceso de los registros de la subsecuencia incompleta */
            if (resto > (*lonSec))
                n2 = resto - (*lonSec);
            else
            {
                n1 = resto;
                n2 = 0;
            }
        }

        i = j = 1;
        while (i <= n1 && j <= n2)
        {
            Registro d;
            if (r1.clave < r2.clave)
            {
                d = r1;
                fread(&r1, sizeof(Registro), 1, f1);
                i++;
            }
            else
            {
                d = r2;
                fread(&r2, sizeof(Registro), 1, f2);
                j++;
            }
            fwrite(&d, sizeof(Registro), 1, f);
        }
    }
}
```

```

/*
  Los registros no procesados se escriben directamente
*/
for (k = i; k <= n1; k++)
{
    fwrite(&r1, sizeof(Registro), 1, f);
    fread(&r1, sizeof(Registro), 1, f1);
}

for (k = j; k <= n2; k++)
{
    fwrite(&r2, sizeof(Registro), 1, f);
    fread(&r2, sizeof(Registro), 1, f2);
}

(*lonSec) *= 2;
fclose (f);fclose(f1);fclose(f2);
}

```

- 16.2** Las reservas de un hotel de n habitaciones se van a gestionar con un archivo directo. Cada reserva tiene los campos nombre del cliente, NIF y número de habitación asignada. Los números de habitación son consecutivos, desde 1 hasta el número de habitaciones. Entonces, se utiliza como índice de registro el número de habitación. Las operaciones que se podrán realizar son: inauguración, entrada de una reserva, finalización de estancia, consulta de habitaciones.

Análisis del problema

Cada registro del archivo se va a corresponder con una reserva y a la vez con el estado de una habitación. Si la habitación n está ocupada el registro de índice n contendrá el nombre del cliente y su nif . Si está vacía, libre, el campo nif va a tener un asterisco (*). Por consiguiente, se utiliza como indicador de habitación libre que $nif == *$.

La operación inauguración inicializa el archivo, escribe tantos registros como habitaciones; cada registro con el campo nif igual a la clave, *, para indicar habitación libre y su número de habitación.

La operación entrada busca en el archivo la primera habitación libre y en su registro escribe uno nuevo con los datos de la reserva.

La finalización de una reserva consiste en asignar al campo nif la clave (*) que indica habitación libre.

También se añade la operación ocupadas para listar todas las habitaciones ocupadas en ese momento.

Codificación

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define numhab 55
FILE *fb = NULL;
const char fich[] = "fichero.dat";

typedef struct
{
    int num;
    char nif[13];
    char nombre[45];
} Habitacion;

```

```
#define desplazamiento(n) ((n - 1) * sizeof(Habitacion))

void inaguracion (void);
void entrada (void);
void salida (void);
void ocupadas (void);
void leerRes (Habitacion * ph);
void escribirRes (Habitacion h);

void main()
{
    char opcion;

    do
    {
        puts ("1. Inaguracion");
        puts ("2. Entrada   ");
        puts ("3. Salida    ");
        puts ("4. Ocupadas   ");
        puts ("5. Salir      ");
        do {
            printf("Elige una opción ");
            scanf("%c%c", &opcion);
        } while (opcion < '1' || opcion > '5');

        switch (opcion)
        {
            case '1':
                inaguracion();
                break;
            case '2':
                entrada();
                break;
            case '3':
                salida();
                break;
            case '4':
                ocupadas();
                system("pause");
                break;
        }
    }
    while (opcion != '5');
    if (fb != NULL) fclose (fb);
}

void inaguracion (void)
{
    Habitacion h;
    int i;
    char opcion;

    if (fb != NULL)
```



```

{
    printf ("Archivo ya creado, ¿ desea continuar(S/N) ?: ");
    scanf ("%c%c", &opcion);
    if (toupper (opcion) != 'N') return;
}

fb = fopen (fich, "wb+");
for (i = 1; i <= numhab; i++)
{
    h.num = i;
    strcpy (h.nif, "");
    fwrite (&h, sizeof(h), 1, fb);
}
flux (fb);
}

void entrada(void)
{
    Habitacion h;
    int encontrado, nh;

    /* Búsqueda secuencial de primera habitación libre */

    encontrado = 0;
    nh = 0;

    if (fb == NULL)
        fb = fopen (fich, "rb+");
    fseek (fb, 0L, SEEK_SET);

    while ((nh < numhab) && !encontrado)
    {
        fread (&h, sizeof(h), 1, fb);
        nh++;
        if (strcmp (h.nif, "") == 0) /* Habitación libre */
        {
            encontrado = 1;
            leerRes (&h);
            fseek (fb, desplazamiento(h.num), SEEK_SET);
            fwrite (&h, sizeof(h), 1, fb);
            puts ("Datos grabados");
        }
    }
    if (!encontrado) puts ("Hotel completo ");
    fflush (fb);
}

void salida (void)
{
    Habitacion h;
    int n;
    char r;

    if (fb == NULL)
        fb = fopen (fich, "rb+");

```

```

printf ("Numero de habitacion: ");
scanf ("%d%c", &n);

fseek (fb, desplazamiento(n), SEEK_SET);
fread (&h, sizeof(h), 1, fb);

if (strcmp (h.nif,"*") != 0)
{
    escribirRes (h);
    printf ("¿Son correctos los datos?(S/N) ");
    scanf ("%c%c",&r);
    if (toupper (r) == 'S')
    {
        strcpy (h.nif, "*");
        fseek (fb, - sizeof(h), SEEK_CUR);

        fwrite (&h, sizeof(h), 1, fb);
    }
}
else
    puts ("La habitacion figura como libre");

fflush (fb);
}

void ocupadas (void)
{
    Habitacion h;
    int n;

    if (fb == NULL)
        fb = fopen(fich, "rb+");
    fseek (fb, 0L, SEEK_SET);
    puts ("      Numero \t NIF \t\t Nombre");
    puts ("      habitacion          ");
    for (n = 1 ; n <= numhab; n++)
    {
        fread (&h, sizeof(h), 1, fb);
        if (strcmp (h.nif ,"*") != 0)
            escribirRes (h);
    }
}

void leerRes (Habitacion *ph)
{
    printf ("Nif: ");
    gets (ph -> nif);
    printf ("Nombre ");
    gets (ph -> nombre);
}

void escribirRes (Habitacion h)
{

```

```

printf ("\t %d", h.num);
printf ("\t%s\t", h.nif);
printf ("\t%s\n", h.nombre);
}

```

- 16.3.** Un archivo secuencial contiene registros con un campo clave de tipo entero en el rango de 0 a 777. Escribir la función `volcado()`, que genere un archivo directo de tal forma que el número de registro coincida con el campo clave.

Análisis del problema

Se va a suponer que en los registros no se repite ninguna clave. La entrada de la función es el nombre del nuevo fichero directo y el puntero al fichero secuencial. La función lee cada registro del fichero secuencial, obtiene la clave y utiliza ese valor para colocar el registro en una posición de valor igual a la clave.

Codificación

```

Volcado (char *nuevonombre, FILE *fps)
{
FILE *pfd;
struct reg registro;
int nreg;

if ((pfd = fopen (nuevonombre, "wb"))==NULL)
{
puts ("NO se puede crear el fichero");
return;
}

while (feof (fps))
{
fread (&registro, sizeof(registro), 1, fps);
nreg = registro.clave;
fseek (pfd, nreg*sizeof(registro), SEEK_SET);
fwrite (&registro, sizeof(registro), 1, pfd);
}
fclose(pfd);
}

```

- 16.4.** Escribir la función principal para gestionar el archivo secuencial indexado de artículos de un supermercado. Los campos que describen cada artículo son: nombre del artículo, identificador, precio, unidades.

Análisis del problema

Únicamente se escribe la codificación de la función `main()` con un sencillo menú para que el usuario elija la operación que quiere realizar. El archivo `articulo.h` contiene la declaración de las estructuras `Articulo` e `Indice`, la macro `desplazamiento` y la declaración de los punteros a `FILE`, `fix` y `findices`. Los prototipos de las funciones desarrolladas en el anterior apartado se encuentran en el archivo `indexado.h`; la implementación de las funciones está en el archivo `indexado.c`.

Codificación (Consultar la página web del libro)

- 16.5** Los registros de un archivo secuencial indexado tienen un campo, `estado`, para conocer si el registro está dado de baja. Escribir una función para compactar el archivo de datos, de tal forma que se eliminen físicamente los registros dados de baja.

Análisis del problema

La mejor manera de realizar la operación consiste en ir leyendo la información del fichero original copiando en un nuevo fichero solo aquellos registros con información válida. Como el fichero original está indexado habrá que asegurarse que el fichero destino también lo esté. La forma que se realiza esto es construyendo un nuevo archivo de índices tras la generación del nuevo fichero compactado.

Codificación

```
typedef struct
{
    char clave[40];
    long posicion;
} Indice;

#define MXAR 200
compactar()
{
    Indice tabla[MXAR];
    FILE *origen, *temp;
    FILE *indices;
    long pos = 0;

    origen = fopen ("datos.dat", "rb+");
    temp = fopen ("temp.dat", "wb");
    indices = fopen ("indices.idx", "wb");

    if ((!origen || !temp || !indices)
        {
            puts ("NO se puede crear el fichero");
            return;
        }

    while (!feof (origen))
    {
        fread (&registro, sizeof(registro), 1, origen);
        if (estado == ACTIVO)
        {
            fwrite (&registro, sizeof(registro), 1, temp);
            strcpy (tabla[i].clave, registro.clave);
            tabla[i++].posicion = pos;
            pos += sizeof(registro);
        }
    }
    for (j=0; j<i; j++)
        fwrite(&tabla[j], sizeof (Indice), findices);
    fclose(origen);
    fclose(temp);
    fclose(findices);
    remove("datos.dat");
    rename("temp.dat", "datos.dat");
}
```

- 16.6** Realizar los cambios necesarios en la función *fusión()* que mezcla dos archivos secuenciales ordenados, ascendentemente respecto al campo fecha de nacimiento. La fecha de nacimiento está formada por los campos de tipo *int*: mes, día y año.

Análisis del problema

La única diferencia respecto a una función de fusión que compare claves enteras es que la comparación entre claves de registro debe hacerse por medio de una función.

Codificación (Consultar la página web del libro)

- 16.7** Escribir una función que distribuya los registros de un archivo no ordenado, *F*, en otros dos *F1* y *F2*, con la siguiente estrategia: leer *M* (por ejemplo, 16) registros a la vez del archivo, ordenarlos utilizando un método de ordenación interna y a continuación escribirlos, alternativamente, en *F1* y *F2*.

Análisis del problema

Se trata de sustituir la función de distribución por otra que cumpla la especificación del enunciado.

Codificación

```
typedef int TipoClave;

typedef struct
{
    TipoClave clave;
} Registro;

void distribuirM (FILE* f, FILE* f1, FILE* f2, int M, int numReg)
{
    int numSec, resto, i;

    numSec = numReg / (2*M);
    resto = numReg % (2*M);

    f = fopen ("fileorg","rb");
    f1 = fopen ("fileAux1","wb");
    f2 = fopen ("fileAux2","wb");

    for (i = 1; i <= numSec; i++)
    {
        subSecuenciaM (f, f1, M);
        subSecuenciaM (f, f2, M);
    }
    /*
     Se procesa el resto de registros del archivo
    */
    if (resto > M)
    {
        resto -= M;
        subSecuenciaM (f, f1, M);
        subSecuenciaM (f, f2, resto);
    }
}
```

```

    else
        subSecuenciaM (f, f1, resto);

    fclose(f1);
    fclose(f2);
    fclose(f);
}

void subSecuenciaM (FILE* f, FILE* t, int m)
{
    Registro r, tabla[m];
    int j;

    for (j = 1; j <= m; j++)
    {
        fread (&tabla[i], sizeof(Registro), 1, f);
    }
    ordenar (tabla, m);
    for (j = 1; j <= m; j++)
    {
        fwrite (&tabla[i], sizeof(Registro), 1, t);
    }
}

```

16.8 *Modificar la implementación de la mezcla directa de tal forma que inicialmente se distribuya el fichero origen en secuencias de M registros ordenados, según se explica en ejercicio 16.5. Y a partir de esa distribución se repitan los pasos del algoritmo mezcla directa: fusión de M -uplas para dar lugar a $2M$ registros ordenados, separación ...*

Codificación

```

void mezclaDirecta(FILE *f)
{
    int longSec;
    int numReg;
    FILE *f1 = NULL, *f2 = NULL;

    f = fopen ("fileorg","rb");

    numReg = numeroReg (f);
    longSec = 16;

    distribuirM (f, f1, f2, longsec, numReg);
    mezclar (f1, f2, f, &longsec, numReg);

    while (longSec < numReg)
    {
        distribuir (f, f1, f2, longSec, numReg);
        mezclar (f1, f2, f, &longSec, numReg);
    }
}

```

- 16.9.** *Un archivo está ordenado alfabéticamente respecto un campo clave que es una cadena de caracteres. Diseñar un algoritmo e implementarlo para que la ordenación sea en sentido inverso.*

Análisis del problema

Solamente es necesario copiarlo al revés en un fichero auxiliar y renombrarlo después.

Codificación

```
OrdenacionInversa (char *fichero)
{
    struct registro reg;
    int numregs;
    FILE *pf, *pftemp, *aux;
    fpos_t fin;

    if ((pf = fopen (fichero, "rb")) == NULL)
    {
        puts ("NO se puede crear el fichero");
        return;
    }
    if ((pftemp = fopen ("temp", "rb")) == NULL)
    {
        puts ("NO se puede crear el fichero");
        return;
    }

    fseek (pf, 0L, SEEK_END);
    fgetpos (pf, &fin);
    nregs = fin / sizeof(Registro);
    while (nregs--)
    {
        fseek (pf, -sizeof (registro), SEEK_SET);
        fread (&reg, sizeof (registro), 1, pf);
        fwrite (&reg, sizeof (registro), 1, pftemp);
        fseek( pf, -sizeof (registro), SEEK_SET);
    }
    fclose (pftemp);
    fclose (pf);
    remove (fichero);
    rename ("temp", fichero);
}
```

- 16.10.** *Un archivo secuencial no ordenado se quiere distribuir en dos ficheros F1 y F2 siguiendo estos pasos:*

Leer N registros del archivo origen y ponerlos en una lista secuencial. Marcar cada registro de la lista con un estatus, por ejemplo activo = 1.

Obtener el registro t con clave más pequeña de los que tienen el estatus activo y escribirlo en el archivo destino F1.

Sustituir el registro t por el siguiente registro del archivo origen. Si el nuevo registro es menor que t, se marca como inactivo, es decir activo = 0; en caso contrario se marca activo. Si hay registros en la lista activos volver al paso 2.

Cambiar el archivo destino, si el anterior es F1 ahora será F2 y viceversa. Activar todos los registros de la lista y volver al paso 2.

Codificación

```

distribucion()
{
FILE fp, fp1, fp2;
Registro reg, tabla[N];
int activo [N], min, hayactivos, menor;

pf = fopen ("origen", "rb");
pf1 = fopen ("destino1", "wb");
pf2 = fopen ("destino2", "wb");
if (pf || !pf1 || !pf2)
{
printf ("Error al abrir los ficheros\n");
exit (1);
}

for (i = 0; i < N; i++)
{
fread (&tabla[i], sizeof (Registro), 1, fp);
activo[i] = 1;
}

while (!feof (fp))
{
do {
/* encontrar el menor de los activos y comprobar que
todavía hay registros activos en la lista */

min = 0;
menor = -1;
hayactivos = 0;
for (i = 0; i < N; i++)
if (tabla[i].clave < min) &&
hayactivos = activo[i])
{
min = tabla[i].clave;
menor = i;
}
if (menor != -1)
fwrite (&tabla[menor], sizeof (Registro), 1, fp1);
if (tabla[menor].clave < min)
activo[menor] = 0;
} while (hayactivos);
aux = fp1;
fp1 = fp2;
fp2 = aux;
for (i=0; i<N; i++) activo [i] = 1;
}

```

16.11. *Los registros que representan los objetos de una perfumería se van a guardar en un archivo hash, se prevé como máximo 1024 registros. El campo clave es una cadena de caracteres, cuya máxima longitud es 10. Con este supuesto codificar la función de dispersión y mostrar 10 direcciones dispersas.*

Análisis del problema

Según lo comentado en el comienzo del capítulo una función sencilla que utilice aritmética modular podría ser la siguiente.

Codificación

```
int hash (Registro reg)
{
    for (i = 0; i <= strlen(reg.clave); i++)
        sumacar += reg.clave[i];
    return sumacar% 1024;
}
```

16.12. Diseñar un algoritmo e implementar un programa que permita crear un archivo secuencial *PERFUMES* cuyos registros constan de los siguientes campos:

Nombre

Descripción

Precio

Código

Creador

```
typedef struct
{
    char nombre[40];
    char descripción[100];
    float precio;
    char codigo[10];
    char creador[40];
} Registro;

main()
{
    FILE *pf;
    Registro reg;

    if ((pf = fopen("perfumes", "wb+"))==NULL)
    {
        printf("ERROR en la creación del fichero");
        exit(1);
    }

    while (1)
    {
        printf ("Introduzca los siguientes datos:\n");
        printf ("Nombre:");
        scanf("%s", reg.nombre);
        printf ("Descripcion:");
        scanf("%s", reg.descripcion);
        printf ("Precio:");
        scanf("%f", &reg.precio);
        printf ("Codigo:");
        scanf("%s", reg.codigo);
        printf ("Creador:");
```

```

scanf("%s", reg.creador);
fseek (pf, 0L, SEEK_END);
fwrite (&reg, sizeof (reg), 1, pf);

printf ("Desea continuar (s/n):");
scanf ("%c", &r);
if (r == 's') break;
}

fclose (pf);
}

```

16.13. Realizar un programa que copie el archivo secuencial *PERFUMES* del ejercicio anterior en un archivo hash *PERME_DIR*; el campo clave es el código del perfume que tiene como máximo 10 caracteres alfanuméricos.

Codificación (Consultar la página web del libro)

16.14. Diseñar un algoritmo e implementar un programa para crear un archivo secuencial indexado denominado *DIRECTORIO*, que contiene los datos de los habitantes de una población que actualmente está formada por 5590 personas. El campo clave es el número de DNI.

Codificación (Consultar la página web del libro)

16.15. Escribir un programa que liste todas las personas del archivo indexado *DIRECTORIO* que pueden votar.

Codificación

```

void ListadoVotantes(Indice* tabla, int nreg)
{
    hab reg;
    long DNI;
    int p;
    FILE * fp;

    fp = fopen ("directorio.dat", "wb+");
    if (fp == NULL)
    {
        printf ("Error al abrir el archivo");
        exit(2);
    }

    while (ifeof (fp))
    {
        fread (&reg, sizeof(reg), 1, fp);
        if (reg.edad >= 18)
        {
            printf (" Nombre: %s\n", reg.nombre);
            printf (" DNI: %ld\n", reg.DNI);
            printf (" Edad: %d\n", reg.edad);
            printf (" Direccion: %s\n", reg.direccion);
            printf (" Sexo: %c", reg.sexo);
        }
    }
}

```

```
fclose (fp);  
}
```

16.16. Realizar un programa que copie los registros de personas con edad ente 18 y 30 años, del archivo *DIRECTORIO* del ejercicio 16.5, en un archivo secuencial *JOVENES*.

Codificación

```
void ListadoJovenes(Indice* tabla, int nreg)  
{  
    hab reg;  
    long DNI;  
    int p;  
    FILE * fp;  
  
    fp = fopen("directorio.dat", "wb+");  
    fp2 = fopen("jovenes.dat", "wb+");  
  
    if (fp == NULL || fp2 == NULL)  
    {  
        printf ("Error al abrir el archivo");  
        exit(2);  
    }  
  
    while (feof(fp))  
    {  
        fread (&reg, sizeof(reg), 1, fp);  
        if (reg.edad >= 18 && reg.edad <= 30)  
            fwrite (&reg, sizeof(reg), 1, fp2);  
    }  
  
    fclose (fp);  
    fclose (fp2);  
}
```

PROBLEMAS PROPUESTOS

- 16.1.** El archivo secuencial F, almacena registros con un campo clave de tipo entero. Supóngase que la secuencia de claves que se encuentra en el archivo es la siguiente:

14 27 33 5 8 11 23 44 22 31 46 7 8 11 1 99 23
40 6 11 14 17

Aplicando el algoritmo de mezcla directa realizar la ordenación del archivo y determinar el número de pasadas necesarias.

- 16.2.** Un archivo secuencial F contiene registros y requiere ser ordenado utilizando 4 archivos auxiliares. Suponiendo que la ordenación se desea hacer respecto a un campo de tipo entero, con estos valores:

22 11 3 4 11 55 2 98 11 21 4 3 8 12 41 21 42
58 26 19 11 59 37 28 61 72 47

Aplicar el esquema seguido en el algoritmo de mezcla directa (tener en cuenta que se utilizan 4 archivos en vez de 2) y obtener el número de pasadas necesarias para su ordenación.

- 16.3.** El archivo JOVENES (ejercicio 16.15) se desea ordenarlo alfabéticamente. Aplicar el método *mezcla directa*.
- 16.4.** Dado un archivo *hash*, diseñar un algoritmo e implementar el código para compactar el archivo después de dar de baja un registro. Es decir, un registro del área de sinónimos se mueve al área principal si el registro del área principal, con el que colisionó el registro del área de sinónimos, fue dado de baja.
- 16.5.** Se necesita construir un catálogo para organizar los datos de una colección de CDs de música. El sistema de gestión de datos guardará la información de cada CD (autor, título, año) y de cada tema (autor, título del tema, título del CD). La información se almacenará en dos ficheros secuenciales indexados. Escribir un programa que además de permitir la entrada y eliminación de los

datos en ambos ficheros, proporcione las opciones de consultar por temas y por CDs.

- 16.6.** Añadir al programa anterior una función que liste el contenido del fichero de CDs y para cada uno de ellos liste los temas que contiene según la información del fichero de temas.
- 16.7.** Utilizar el algoritmo de mezcla directa para ordenar los ficheros de los dos ejercicios anteriores. Escribir una función que utilice el algoritmo de búsqueda binaria para encontrar un registro determinado en los ficheros indexados descritos.
- 16.8.** Un sistema de gestión de información de un almacén debe mantener tres ficheros relacionados. El primer fichero tiene información de cada proveedor indexado por el nombre del mismo. Un segundo fichero mantiene las características de cada producto, incluyendo el nombre del proveedor, y está indexado por un código alfanumérico. El tercer fichero lleva la información de las existencias de cada producto y su localización en el almacén.
- Escribir un programa que liste los productos de los que existan menos de 10 unidades en el almacén. Al lado de cada producto en el listado deben aparecer los datos del proveedor.
- 16.9.** En el escenario del ejercicio anterior se puede suponer que cada vez que se adquieren nuevos productos en el almacén se añaden registros con la cantidad incorporada en cada producto. También cuando se venden productos se añaden registros por cada producto en los que se especifica una cantidad negativa. Implementar un programa que al final del día reagrupe la información de entradas y salidas dejando un registro por producto con las existencias reales.
- 16.10.** Completar el programa del ejercicio 16.8. con una función que ordene los registros de producto del segundo fichero agrupando todos los registros del mismo proveedor.

Tipos abstractos de datos

TAD/objetos

En este capítulo se examinan los conceptos de *modularidad*, *abstracción de datos* y *objetos*. La modularidad es la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. *Abstracción de datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles. Los *objetos* combinan en una sola unidad *datos* y *funciones* que operan sobre esos datos.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C. Casi todos los lenguajes de programación tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato*, **TAD**, (*abstract data type*, **ADT**). El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de *tipo abstracto de dato (TAD)* siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

17.1 Tipos de datos

Todos los lenguajes de programación soportan algún tipo de datos. Por ejemplo, el lenguaje de programación convencional C soporta tipos base tales como enteros, reales y caracteres; así como tipos compuestos tales como *arrays* (vectores y matrices) y *estructuras*(registros).

A tener en cuenta

Un tipo de dato es un conjunto de valores, y un conjunto de operaciones definidas sobre esos valores.

Un valor depende de su representación y de la interpretación de la representación, por lo que una definición informal de un tipo de dato es: *Representación + Operaciones*.

Un *tipo de dato* describe un conjunto de objetos con la misma representación. Existen un número de operaciones asociadas con cada tipo. Es posible realizar aritmética sobre tipos de datos enteros y reales, concatenar cadenas o recuperar o modificar el valor de un elemento.

La mayoría de los lenguajes tratan las variables y constantes de un programa como *instancias* de un *tipo de dato*. Un tipo de dato proporciona una descripción de sus instancias que indica al compilador cosas como cuánta memoria se debe asignar para una instancia, cómo interpretar los datos en memoria y qué operaciones son permisibles sobre esos datos. Por ejemplo, cuando se escribe una declaración tal como `float z` en C ó C++, se está declarando una instancia denominada *z* del tipo de dato `float`. El tipo de datos `float` indica al compilador que reserve, por ejemplo, 32 bits de memoria, y qué operaciones tales como “sumar” y “multiplicar” están permitidas, mientras que operaciones tales como el “el resto” (*módulo*) y “desplazamiento de bits” no lo están. Sin embargo, no se necesita escribir la declaración del tipo `float` -el autor de compilador lo hizo ya y se construyen en el compilador-. Los tipos de datos que se construyen en un compilador de este modo, se conocen como *tipos de datos fundamentales (predefinidos)*, y por ejemplo en C y C++ son entre otros: `int`, `char`, `float` y `double`.

Cada lenguaje de programación incorpora una colección de tipos de datos fundamentales, que incluyen normalmente enteros, reales, carácter, etc. Los lenguajes de programación soportan también un número de constructores de tipos incorporados que permiten generar tipos más complejos. Por ejemplo, C soporta registros (estructuras) y arrays.

A tener en cuenta

El programador no tiene que preocuparse de saber cómo el compilador del lenguaje implementa los tipos de datos predefinidos, simplemente usa los tipos de datos en el programa.

17.2 Tipos abstractos de datos

Algunos lenguajes de programación tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina *tipo abstracto de datos (TAD)*, se implementa considerando los valores que se almacenan en las variables y las operaciones disponibles para manipular esas variables. Por ejemplo, en C el tipo *Punto*, que representa a las coordenadas *x* e *y* de un sistema de coordenadas rectangulares, no existe; el programador puede definir el *tipo abstracto de datos Punto* que represente las coordenadas rectangulares, y las operaciones que se pueden realizar (*distancia, módulo* ...). En esencia un tipo abstracto de datos es un tipo de datos que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre esos datos. Un **TAD** se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

Para recordar

Un tipo abstracto de datos puede definirse mediante la ecuación:

TAD = Representación (datos) + Operaciones (funciones y procedimientos)

Desde un punto de vista global, un *tipo abstracto de datos* se compone de la interfaz y de la implementación (Figura 17.1). Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los TAD están encapsuladas dentro de los propios TAD. La característica de ocultamiento de la información del TAD significa que disponen de *interfaces públicas*, sin embargo, las representaciones e implementaciones de esas interfaces son *privadas*.

VENTAJAS DE LOS TIPOS ABSTRACTOS DE DATOS

Un *tipo abstracto de datos* es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor conceptualización y modelización del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la robustez del sistema. Los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.

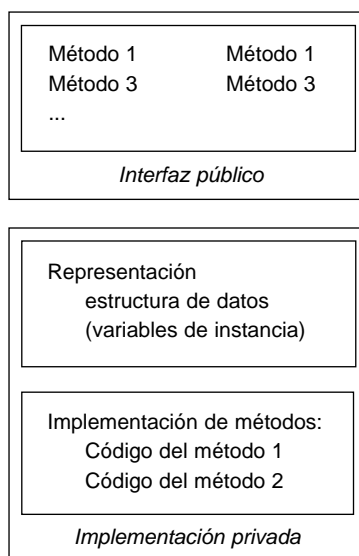


Figura 17.1 Estructura de un tipo abstracto de datos (TAD)

3. Mejora el rendimiento (prestaciones). Para sistemas tipificados, el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar a la interfaz pública del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un TAD lo hace teniendo en cuenta las operaciones o funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los *usuarios* de un TAD se comunican con este a partir de la interfaz que ofrece el TAD mediante funciones de acceso. Podría cambiarse la implementación de tipo de datos sin afectar al programa que usa el TAD ya que para el programa está *oculta* la implementación.

IMPLEMENTACIÓN DE LOS TAD

Los lenguajes convencionales, tales como C, permiten la definición de nuevos tipos y la declaración de funciones para realizar operaciones sobre objetos de los tipos. Sin embargo, tales lenguajes no permiten que los datos y las operaciones asociadas sean declaradas juntos como una unidad y con un solo nombre. En los lenguajes en el que los módulos (TAD) se pueden implementar como una unidad, éstos reciben nombres distintos:

Turbo Pascal	<i>unidad, objeto</i>
Modula-2	<i>módulo</i>
Ada	<i>paquete</i>
C++	<i>clase</i>
Java	<i>clase</i>

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos ocultos al exterior, y la *implementación*, que muestra el código fuente de las operaciones y que permanece oculto al exterior del módulo.

En C no existe como tal una construcción del lenguaje para especificar un TAD. Sin embargo se puede agrupar la interfaz y la representación de los datos en un archivo de inclusión: *archivo.h*. La implementación de la interfaz, de las funciones se realiza en el correspondiente *archivo.c*. Los detalles de la codificación de las funciones quedan *ocultos* en el *archivo.c*.

Las ventajas de los TAD se pueden manifestar en toda su potencia, debido a que las dos partes de los módulos (*especificación* e *implementación*) se pueden compilar por separado mediante la técnica de compilación separada ("*separate compilation*").

17.3 Especificación de los TAD

Un tipo abstracto de datos es un tipo de datos definido por el usuario que tiene un conjunto de datos y unas operaciones. La especificación de un TAD consta de dos partes, la descripción matemática del conjunto de datos, y las operaciones definidas en ciertos elementos de ese conjunto de datos. El objetivo de la especificación es describir el comportamiento del TAD.

La especificación del TAD puede tener un enfoque *informal*, en el que se describen los datos y las operaciones relacionadas en *lenguaje natural*. Otro enfoque mas riguroso, especificación formal, supone suministrar un conjunto de axiomas que describen las operaciones en su aspecto sintáctico y semántico.

PROBLEMAS RESUELTOS

- 17.1.** Realizar una especificación informal del TAD Conjunto con las operaciones: *ConjuntoVacio*, *Esvacio*, *Añadir un elemento al conjunto*, *Pertenece un elemento al conjunto*, *Retirar un elemento del conjunto*, *Union de dos conjuntos*, *Intersección de dos conjuntos* e *Inclusión de conjuntos*.

Análisis del problema

La especificación informal consiste en dos partes:

- detallar en los datos del tipo, los valores que pueden tomar.
- describir las operaciones, relacionándolas con los datos.

El formato que especificación emplea, primero especifica el nombre del TAD y los datos:
TAD *nombre del tipo* (valores y su descripción)

A continuación cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural.
Operación(argumentos). *Descripción funcional*

Como ejemplo se va a especificar el tipo abstracto de datos *Conjunto*:

TAD Conjunto (Especificación de elementos sin duplicidades pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).

Operaciones, existen numerosas operaciones matemáticas sobre conjuntos, algunas de ellas:

Conjuntovacio.

Crea un conjunto sin elementos

Añadir(Conjunto, elemento).

Comprueba si el elemento forma parte del conjunto, en caso negativo es añadido. La función modifica al conjunto.

Retirar(Conjunto, elemento).

En el caso de que el elemento pertenezca al conjunto es eliminado de este. La función modifica al conjunto.

Pertenece(Conjunto, elemento).

Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*.

Esvacio(Conjunto)

Verifica si el conjunto no tiene elementos, en cuyo caso devuelve *cierto*.

Cardinal(Conjunto)

Devuelve el número de elementos del conjunto

Union (Conjunto, Conjunto).

Realiza la operación matemática de la unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes a los dos argumentos.

Intersección (Conjunto, Conjunto).

Realiza la inclusión matemática de la intersección de dos conjuntos. La operación devuelve un conjunto con los elementos comunes a los dos argumentos.

Inclusión (Conjunto, Conjunto).

Verifica si el primer conjunto está incluido en el conjunto especificado en el segundo argumento, en cuyo caso devuelve *cierto*.

- 17.2.** Realizar la especificación formal del TAD *Conjunto* con las operaciones indicadas en el ejercicio 17.1. Considerar a las operaciones *ConjuntoVacio* y *Añadir* como constructores.

Análisis del problema

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir una parte de sintaxis, en cuanto a los tipos de los argumentos y el tipo del resultado, y una parte de semántica, donde se detalla para unos valores particulares de los argumentos la expresión del resultado que se obtiene. La especificación formal ha de ser lo bastante *potente* para que cumpla el objetivo de verificar la corrección de la implementación del TAD.

El esquema que se sigue para especificar formalmente un TAD consta de una cabecera con el nombre del TAD y los datos: TAD *nombre del tipo* (valores que toma los datos del tipo)

Le sigue la sintaxis de las operaciones (se listan las operaciones indicando los tipos de los argumentos y el tipo del resultado):

Sintaxis

Operación(Tipo argumento, ...) → Tipo resultado

y a continuación, la semántica de las operaciones. Esta se construye dando unos valores particulares a los argumentos de las operaciones, a partir de los cuales se obtiene una expresión resultado que puede tener referencias a tipos ya definidos, valores de tipo lógico o referencias a otras operaciones del propio TAD.

Semántica

Operación(valores particulares argumentos) ⇒ expresión resultado

Al hacer una especificación formal siempre hay operaciones definidas por sí mismas, se consideran constructores del TAD. Se puede decir que mediante estos constructores se generan todos los posibles valores del TAD. Normalmente, se elige como constructor la operación que inicializa (por ejemplo, *Conjuntovacio* en el TAD *Conjunto*), y la operación que añade un dato o elemento (esta operación es común a la mayoría de los tipos abstractos de datos). Se acostumbra a marcar con un asterisco a las operaciones que son constructores.

A continuación se hace la especificación formal del TAD *Conjunto*, para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa *si-entonces-sino*.

TAD *Conjunto*(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).

Sintaxis

*Conjuntovacio	→	Conjunto
*Añadir(Conjunto, Elemento)	→	Conjunto
Retirar(Conjunto, Elemento)	→	Conjunto
Pertenece(Conjunto, Elemento)	→	Conjunto
Esvacio(Conjunto)	→	boolean
Cardinal(Conjunto)	→	entero
Union(Conjunto, Conjunto)	→	Conjunto
Interseccion(Conjunto, Conjunto)	→	Conjunto
Incluido(Conjunto, Conjunto)	→	boolean

Semántica

$\forall e_1, e_2 \in \text{Elemento y } \forall C, D \in \text{Conjunto}$

Añadir(Añadir(C, e1), e1)	⇒ Añadir(C, e1)
Añadir(Añadir(C, e1), e2)	⇒ Añadir(Añadir(C, e2), e1)
Retirar(Conjuntovacio, e1)	⇒ Conjuntovacio
Retirar(Añadir(C, e1), e2)	⇒ si e1 = e2 entonces Retirar(C,e2) sino Añadir(Retirar(C,e2),e1)
Pertenece(Conjuntovacio, e1)	⇒ falso
Pertenece(Añadir(C, e2), e1)	⇒ si e1 = e2 entonces cierto sino Pertenece(C, e1)
Esvacio(Conjuntovacio)	⇒ cierto
Esvacio(Añadir(C, e1))	⇒ falso
Cardinal(Conjuntovacio)	⇒ Cero
Cardinal(Añadir(C, e1))	⇒ si Pertenece(C,e1) entonces Cardinal(C) sino 1 + Cardinal(C)
Union(Conjuntovacio, Conjuntovacio)	⇒ Conjuntovacio
Union(Conjuntovacio, Añadir(C, e1))	⇒ Añadir(C, e1)
Union(Añadir(C, e1), D)	⇒ Añadir(Union(C, D), e1)
Interseccion(Conjuntovacio, Conjuntovacio)	⇒ Conjuntovacio
Interseccion(Añadir(C, e1),Añadir(D, e1))	⇒ Añadir(Interseccion (C,D), e1)
Incluido(Conjuntovacio, Conjuntovacio)	⇒ cierto
Incluido(Añadir(C, e1), Añadir(D, e1))	⇒ cierto si Incluido (C, D)

- 17.3.** Crear un TAD que represente un dato tipo cadena (*string*) y sus diversas operaciones: *CadenaVacía*, *Asignar*, *Longitud*, *Buscar posición de un carácter dado*, *Concatenar cadenas*, *Extraer una subcadena*. Realizar la especificación informal y formal considerando como constructores las operaciones *CadenaVacía* y *Asignar*.

ESPECIFICACIÓN INFORMAL

TAD Cadena (Secuencia de caracteres ASCII terminada por un byte nulo).

Operaciones

Cadenavacía.

Crea una cadena vacía

Asignar (Cadena, Cadena1).

Elimina el contenido de la primera cadena si lo hubiere y lo sustituye por la segunda.

Longitud (Cadena).

Devuelve el número de caracteres de la cadena sin contar el byte final.

Buscar (Cadena, Carácter)

Devuelve la posición de la primera ocurrencia del carácter por la izquierda.

Concatenar (Cadena1, Cadena2).

Añade el contenido de Cadena2 a la cadena del primer argumento.

Extraer (Cadena, Posición, NumCaracteres).

Devuelve la subcadena del primer argumento que comienza en la posición del segundo argumento y tiene tantos caracteres como indica el tercero.

ESPECIFICACIÓN FORMAL

TAD Cadena (Secuencia de caracteres ASCII terminada por un byte nulo).

Sintaxis

*Cadenavacia	-> Cadena
*Asignar (Cadena, Cadena)	-> Cadena
Longitud (Cadena)	-> entero
Buscar (Cadena, Carácter)	-> entero
Concatenar (Cadena1, Cadena2)	-> Cadena
Extraer (Cadena, Posición, NumCaracteres)	-> Cadena

- 17.4.** *Diseñar el TAD Bolsa como una colección de elementos no ordenados y que pueden estar repetidos. Las operaciones del tipo abstracto: CrearBolsa, Añadir un elemento, BolsaVacía (verifica si tiene elementos), Dentro (verifica si un elemento pertenece a la bolsa), Cuantos (determina el número de veces que se encuentra un elemento), Union y Total. Realizar la especificación informal y formal considerando como constructores las operaciones CrearBolsa y Añadir.*

ESPECIFICACIÓN INFORMAL

TAD Bolsa (Colección de elementos no ordenados que pueden estar repetidos).

Operaciones*CrearBolsa*

Crea una bolsa vacía.

Añadir (Bolsa, elemento)

Añade un elemento a la bolsa.

BolsaVacía (Bolsa).

Verifica que la bolsa no tiene elementos.

Dentro (elemento, Bolsa).

Verifica si un elemento pertenece a la bolsa

Cuantos (elemento, Bolsa).

Determina el número de veces que se encuentra un elemento en una bolsa

Union (Bolsa1, Bolsa2).

Devuelve una bolsa con los elementos de los dos argumentos.

Total (Bolsa).

Devuelve el número de elementos de una bolsa.

ESPECIFICACIÓN FORMAL

TAD Cadena (Secuencia de caracteres ASCII terminada por un byte nulo).

Sintaxis

*CrearBolsa	-> Bolsa
*Añadir (Bolsa, elemento)	-> Bolsa
BolsaVacía (Bolsa)	-> boolean
Dentro (elemento, Bolsa)	-> boolean
Cuantos (elemento, Bolsa)	-> entero
Union (Bolsa1, Bolsa2)	-> Bolsa
Total (Bolsa)	-> Bolsa

- 17.5.** *Diseñar el TAD Complejo para representar a los números complejos. Las operaciones que se deben definir: AsignaReal (asigna un valor a la parte real), AsignaImaginaria (asigna un valor a la parte imaginaria), ParteReal (devuelve la parte real de un complejo), ParteImaginaria (devuelve la parte imaginaria de un complejo), Modulo de un complejo y Suma de dos números complejos. Realizar la especificación informal y formal considerando como constructores las operaciones que desee.*

ESPECIFICACIÓN INFORMAL

TAD Complejo (Par de números reales que representan la parte real e imaginaria de un número complejo según el concepto matemático).

Operaciones

AsignaReal (Complejo, real).

Asigna un valor a la parte real de un número complejo.

AsignaImaginaria (Complejo, real).

Asigna un valor a la parte imaginaria de un número complejo.

ParteReal (Complejo).

Devuelve la parte real de un número complejo.

ParteImaginaria (Complejo).

Devuelve la parte imaginaria de un número complejo.

Modulo (Complejo).

Devuelve el módulo de un número complejo.

Suma (Complejo1, Complejo2).

Devuelve la suma de dos números complejos

ESPECIFICACIÓN FORMAL

TAD Complejo (Par de números reales que representan la parte real e imaginaria de un número complejo según el concepto matemático).

Sintaxis

*AsignaReal (Complejo, real)	-> Complejo
*AsignaImaginaria (Complejo, real)	-> Complejo
ParteReal (Complejo)	-> real
ParteImaginaria (Complejo)	-> real
Modulo (Complejo)	-> real
Suma (Complejo1, Complejo2)	-> Complejo

- 17.6.** Diseñar el tipo abstracto de datos Vector con la finalidad de representar una secuencia de n elementos del mismo tipo. Las operaciones a definir: *CrearVector* (crea un vector n posiciones vacías), *Asignar* (asigna un elemento en la posición j), *ObtenerElemento* (devuelve el elemento que se encuentra en la posición j), *SubVector* (devuelve el vector comprendido entre las posiciones i, j). Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

ESPECIFICACIÓN INFORMAL

TAD Vector (secuencia de n elementos del mismo tipo).

Operaciones

CrearVector (entero).

Crea un vector n posiciones vacías.

Asignar (Vector, posición, elemento).

Asigna un elemento en la posición indicada en el segundo parámetro.

ObtenerElemento (Vector, posición).

Devuelve el elemento que se encuentra en la posición indicada en el segundo parámetro.

SubVector (Vector, inicial, final).

Devuelve el vector comprendido entre las posiciones indicadas en los parámetros finales.

ESPECIFICACIÓN FORMAL

TAD Vector (secuencia de n elementos del mismo tipo).

Sintaxis

*CrearVector (entero)	-> Vector.
*Asignar (Vector, entero, elemento)	-> Vector.
ObtenerElemento (Vector, entero)	-> elemento.
SubVector (Vector, entero, entero)	-> Vector.

- 17.7.** Diseñar el tipo abstracto de datos Matriz con la finalidad de representar matrices matemáticas. Las operaciones a definir: *CrearMatriz* (crea una matriz, sin elementos, de m filas por n columnas), *Asignar* (asigna un elemento en la fila i columna j), *ObtenerElemento* (obtiene el elemento de la fila i y columna j), *Sumar* (realiza la suma de dos matrices cuando tienen las mismas dimensiones), *ProductoEscalar* (obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor). Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

ESPECIFICACIÓN INFORMAL

TAD Matriz (Secuencia de elementos organizados en filas y columnas).

Operaciones

CrearMatriz (filas, columnas).

Crea una matriz, sin elementos, de las dimensiones que indican los argumentos.

Asignar (Matriz, fila, columna, elemento).

Asigna un elemento en la posición que indican los argumentos finales.

ObtenerElemento (Matriz, fila, columna).

Obtiene el elemento de la posición que indican los argumentos finales.

Sumar (Matriz1, Matriz2).

Realiza la suma de dos matrices cuando tienen las mismas dimensiones

ProductoEscalar (Matriz, valor).

Obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor.

ESPECIFICACIÓN FORMAL

TAD Matriz (Secuencia de elementos organizados en filas y columnas).

Sintaxis

*CrearMatriz (entero, entero)	-> Matriz.
Asignar (Matriz, entero, entero, elemento)	-> Matriz.
ObtenerElemento (Matriz, entero, entero)	-> Matriz.
Sumar (Matriz, Matriz)	-> Matriz.
ProductoEscalar (Matriz, valor)	-> Matriz.

- 17.8.** Implementar el TAD Conjunto con las operaciones especificadas en los ejercicios 17.1 y 17.2.

Análisis del problema

La implementación del tipo abstracto de datos debe incluir dos partes diferenciadas:

- representación de los datos.
- implementación de las operaciones descritas en la especificación.

Los archivos de *inclusión* o de **cabecera** se utilizan para agrupar en ellos variables externas, declaraciones de datos comunes y prototipos de funciones. Estos archivos de cabecera se incluyen en los archivos que contienen la codificación de las funciones, archivos fuente, y también en los archivos de código que hagan referencia a algún elemento del *archivo de inclusión*, con la directiva del preprocesador `#include`. Al implementar un TAD en C, se agrupa, en cierto modo se *encierra*, en estos archivos la representación de los datos y el interfaz del TAD, a su vez, representado por los prototipos de las funciones. De esta forma en los archivos de código fuente que utilicen el TAD hay que escribir la directiva

```
#include "tipodedato.h"
```

Para hacer la implementación lo más flexible posible, no se establece que el conjunto pueda tener un máximo de elementos. Esta característica exige el uso de asignación dinámica de memoria.

En el archivo de cabecera, `conjunto.h`, se realiza la declaración de la estructura que va a representar a los datos. El tipo de los datos puede ser cualquiera, entonces es necesario que `TipoDato` esté especificado antes de incluir `conjunto.h`. La constante `M`, que arbitrariamente toma el valor de 10, es el número de “huecos” o posiciones de memoria, que se reservan cada vez que hay que ampliar el tamaño de la estructura.

Archivo conjunto.h

```
#define M 10
typedef struct
{
    TipoDato* cto;
    int cardinal;
    int capacidad;
} Conjunto;

void conjuntoVacio (Conjunto* c);
int esVacio (Conjunto c);
void añadir (Conjunto* c, TipoDato elemento);
void retirar (Conjunto* c, TipoDato elemento);
int pertenece (Conjunto c, TipoDato elemento);
int cardinal (Conjunto c);
Conjunto unionC (Conjunto c1, Conjunto c2);
Conjunto interseccionC (Conjunto c1, Conjunto c2);
int incluido (Conjunto c1, Conjunto c2);
```

Este archivo de cabecera, `conjunto.h`, hay que incluirlo en todos los archivos con código C que vaya a utilizar el tipo *Conjunto*. Es importante recordar que antes de escribir la sentencia `include` hay que asociar un tipo predefinido a `TipoDato`. Por ejemplo, si los elementos del conjunto son las coordenadas de un punto en el plano:

```
typedef struct
{
    float x;
    float y;
} Punto;

typedef Punto TipoDato;
#include "conjunto.h"
```

Las funciones cuyos prototipos han sido ya escritos, se codifican y se guardan en el archivo `conjunto.c`. La compilación de `conjunto.c` da lugar al archivo con el código objeto que se ensamblará con el código objeto de otros archivos fuente que hacen uso del TAD *Conjunto*.

Codificación

Archivo conjunto.c

```
typedef struct    {var(s)} Tipo;

typedef Tipo TipoDato;
#include "conjunto.h"

/* iguales() devuelve 1(cierto) si todos los campos lo son.
   La implementación depende del tipo concreto de los dato
   del conjunto.
*/
int iguales (TipoDato e1, TipoDato e2)
{
    return (e1.v1 == e2.v1) && (e1.v2 == e2.v2) ... ;
}

void conjuntoVacio(Conjunto* c)
{
    c -> cardinal = 0;
    c -> capacidad = M;
    c -> cto = (TipoDato*)malloc (M*sizeof(TipoDato));
}

int esVacio(Conjunto c)
{
    return (c.cardinal == 0);
}

void añadir (Conjunto* c, TipoDato elemento)
{
    if (!pertenece(*c, elemento))
    {
        /* verifica si hay posiciones libres,
           en caso contrario amplia el conjunto */
        if (c -> cardinal == c -> capacidad )
        {
            Conjunto nuevo;
            int k, capacidad;
            capacidad = (c -> capacidad + M)*sizeof(TipoDato)
            nuevo.cto = (TipoDato*) malloc(capacidad);

            for (k = 0; k < c -> capacidad; k++)
                nuevo.cto[k] = c -> cto[k];

            free(c -> cto);
            c -> cto = nuevo.cto;
        }
        c -> cto[c -> cardinal++] = elemento;
    }
}
```



```

void retirar (Conjunto* c, TipoDato elemento)
{
    int k;

    if (pertenece (*c, elemento))
    {
        k = 0;
        while (!iguales (c -> cto[k], elemento)) k++;

        /* desde el elemento k hasta la última posición
           mueve los elementos una posición a la izquierda */

        for (; k < c -> cardinal ; k++)
            c -> cto[k] = c -> cto[k+1];

        c -> cardinal--;
    }
}

int pertenece (Conjunto c, TipoDato elemento)
{
    int k, encontrado;
    k = encontrado = 0;

    while (k < c.cardinal && !encontrado)
    {
        encontrado = iguales (c.cto[k], elemento);
        k++;
    }
    return encontrado;
}

int cardinal(Conjunto c)
{
    return c.cardinal;
}

Conjunto unionC(Conjunto c1, Conjunto c2)
{
    Conjunto u;
    int k;
    u.cardinal = 0;
    u.capacidad = c1.capacidad;
    u.cto = (TipoDato*)malloc (u.capacidad*sizeof(TipoDato));

    for (k = 0; k < c1.capacidad; k++)
        u.cto[k] = c1.cto[k];
    u.cardinal = c1.cardinal;

    for (k = 0; k < c2.capacidad; k++)
        añadir (&u, c2.cto[k]);
    return u;
}

```

```

Conjunto interseccionC (Conjunto c1, Conjunto c2)
{
    Conjunto ic;
    int k, l;

    ic.cardinal = 0;
    ic.capacidad = c1.capacidad;
    ic.cto = (TipoDato*)malloc (u.capacidad*sizeof(TipoDato));

    for (k = 0; k < c1.capacidad; k++)
        for (l = 0; l < c1.capacidad; l++)
            if (iguales (c1.cto[k], c2.cto[l]))
            {
                annadir (&ic, c1.cto[k]);
                ic.cardinal++;
            }
    return ic;
}

int incluido (Conjunto c1, Conjunto c2)
{
    int k;

    if (c1.cardinal==0) return 1;
    for (k = 0; k < c1.capacidad; k++)
        if (!pertenece (c2, c1.cto[k]))
            return 0;

    return 1;
}

```

- 17.9.** Implementar el TAD Bolsa descrito en el ejercicio 17.4. Probar la implementación con un programa que invoque a las operaciones del tipo abstracto Bolsa.

Codificación (Consultar la página web del libro)

- 17.10.** Implementar el TAD Cadena descrito en el ejercicio 17.3. Probar la implementación con un programa que realice diversas operaciones con cadenas.

Análisis del problema

En primer lugar es conveniente definir el contenido de un fichero de cabecera para definir los prototipos de las funciones que se implementarán y el tipo de datos “cadena”.

Codificación

```

cadena.h

#define N 100
typedef char cadena [N];

cadena cadnavacia ();

```

```

cadena asignar (cadena cad1, cadena cad2);
int longitud (cadena cad);
int buscar (cadena cad, char c);
cadena concatenar (cadena cad1, cadena cad2);
cadena extraer (cadena cad, int pos, int NumCar);

```

Las funciones cuyos prototipos han sido ya escritos, se codifican en el archivo `cadena.c`.

```

#include cadena.h

cadena cadnavacia ()
{
    char *vacía = (char*) malloc(80);
    return vacía;
}

cadena asignar (cadena cad1, cadena cad2)
{
    return strcat (cad1, cad2);
}

int longitud (cadena cad)
{
    return (strlen (cad));
}

int buscar (cadena cad, char c)
{
    return (strchr (cad, c));
}

cadena concatenar (cadena cad1, cadena cad2)
{
    return (strcat(cad1,cad2));
}

cadena extraer (cadena cad, int pos, int NumCar);
{
    char *aux = (char*) malloc (numcar + 1);
    int i,j = 0;

    for (i = pos; i < pos+numcar; i++)
        aux[j] = cad[i];
    return aux;
}

```

Un programa que pruebe las funciones anteriores podría ser el siguiente:

```

main()
{
    cadena cad1;
    cadena cad2, cad3;

```

```

cad1 = cadnavacia ();
cad2 = cadnavacia ();
cad2 = cadnavacia ();

printf ("Introduzca una cadena de caracteres: ");
scanf ("%s", cad1);
printf ("\nSu cadena tiene %d caracteres.\n", longitud(cad1));

printf ("Esta es la cadena con las dos mitades intercambiadas\n");

mitad = (int)(longitud(cad1)/2);
cad2 = extraer (cad1, 0, mitad);
cad3 = extraer (cad1, mitad+1, mitad);
cad3 = concatenar (cad3, cad2);

puts (cad3);
}

```

Deberá ser compilado junto con el fichero `cadena.c` que contiene las implementaciones de las funciones.

- 17.11.** Implementar el TAD Matriz especificado en el ejercicio 17.7 con una estructura dinámica. Escribir un programa que haciendo uso del TAD Matriz se realicen operaciones diversas y escriba las matrices generadas.

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

- 17.1.** Implementar el TAD Complejo especificado en el ejercicio 17.5. Escribir un programa en el que se realicen diversas operaciones con números complejos.
- 17.2.** Implementar el TAD Vector especificado en el ejercicio 17.6 con una estructura dinámica
- 17.3.** Diseñar el TAD Grafo como un conjunto de nodos y de aristas. Las operaciones del tipo abstracto serán: *CrearGrafo*, *AñadirNodo*, *AñadirArista*, *GrafoVacio* (verifica si tiene nodos o aristas), *Recorrido en Profundidad*, *Recorrido en Anchura*, *Cuantos* (determina el número de nodos y el número de aristas), *Union* de dos grafos y *Conectados* (Verifica si existe un camino entre dos nodos). Realizar la especificación informal y formal considerando como constructores las operaciones *CrearGrafo*, *AñadirNodo* y *AñadirArista*.
- 17.4.** Implementar el TAD Grafo especificado en el ejercicio anterior con una estructura dinámica. Escribir un programa que haciendo uso del TAD. Grafo se realicen operaciones diversas.
- 17.5.** Diseñar el TAD Árbol Binario ordenado. Las operaciones del tipo abstracto serán: *CrearArbol*, *AñadirNodo*, *ArbolVacio* (verifica si tiene nodos), *Recorrido en Profundidad*, *Recorrido en Anchura*, *Cuantos* (determina el número de nodos), *Union* de dos árboles y *Equilibrar* el árbol. Realizar la especificación informal y formal considerando como constructores las operaciones *CrearArbol* y *AñadirNodo*.
- 17.6.** Implementar el TAD Árbol Binario ordenado especificado en el ejercicio anterior con una estructura dinámica. Escribir un programa que haciendo uso del TAD Árbol Binario ordenado se realicen operaciones diversas.
- 17.7.** Diseñar el TAD Buzón de Mensajes. Las operaciones del tipo abstracto serán: *CrearBuzon*, *AbrirBuzon*, *BuzonVacio*, *RecibirMensaje* (recibir el último mensaje),

EnviarMensaje, *VaciarBuzon*, *DestruirBuzon*. Realizar la especificación informal y formal considerando como constructor la operación *CrearBuzon*.

17.8. Implementar el TAD Buzón de Mensajes especificado en el ejercicio anterior con una estructura dinámica. Escribir un programa que haciendo uso del TAD Buzón de Mensajes se realicen operaciones diversas.

17.9. Diseñar el TAD Semáforo. Las operaciones del tipo abstracto serán: *CrearSemaforo* (Crear un semáforo y

ponerlo en un estado determinado), *DestruirSemaforo*, *AbrirSemaforo*, *CerrarSemaforo*, *EsperarSemaforo* (Esperar a que se abra el semáforo y cerrarlo). Realizar la especificación informal y formal.

17.10. Implementar el TAD Semáforo especificado en el ejercicio anterior. Escribir un programa que haciendo uso del TAD Semáforo se realicen operaciones diversas.

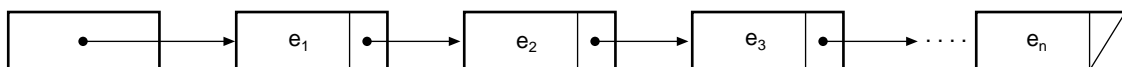
Listas enlazadas

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* - listas, vectores y tablas - y *estructuras*) en las que su tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la **lista enlazada** (*ligada o encadenada*, “*linked list*”) que es una colección de elementos (denominados *nodos*) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un “*enlace*” o “*puntero*”. Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación

18.1 Fundamentos teóricos

Gracias a la asignación dinámica de variables, se pueden implementar listas de modo que la memoria física utilizada se corresponda con el número de elementos de la tabla. Para ello se recurre a los **punteros** (*apuntadores*). Una **lista enlazada** es una secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un “*enlace*” o “*puntero*”. La idea consiste en construir una lista cuyos elementos llamados **nodos** se componen de dos partes o *campos*: la primera parte o campo contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato*, *TipoElemento*, *Info*, etc.) y la segunda parte o *campo* es un puntero (denominado *enlace* o *sgt*, *sig*, etc.) que apunta al siguiente elemento de la lista. La representación gráfica es la siguiente.



e_1, e_2, \dots, e_n son valores del tipo `TipoElemento`.

Figura 18.1 Lista enlazada.

18.2 Clasificación de las listas enlazadas

Las listas se pueden dividir en cuatro categorías:

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor.

- *Listas doblemente enlazadas.* Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor.
- *Lista circular simplemente enlazada.* Una lista enlazada simplemente en la que el último elemento se enlaza al primer elemento.
- *Lista circular doblemente enlazada.* Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa.

Por cada uno de estos cuatro tipos de estructuras de listas, se puede elegir una implementación basada en arrays o una implementación basada en punteros.

18.3 Operaciones en listas enlazadas

Las operaciones sobre listas enlazadas mas usuales son: *Declaración de los tipos nodo y puntero a nodo; inicialización o creación; insertar elementos en una lista; eliminar elementos de una lista; buscar elementos de una lista; recorrer una lista enlazada; comprobar si la lista está vacía.*

EJEMPLO 18.1 Declaración de un Nodo

En C, se puede declarar un nuevo tipo de dato por un nodo mediante las palabras reservadas Struct de la siguiente forma

<pre>struct Nodo { int info; struct Nodo* sig; };</pre>	<pre>typedef struct Nodo { int info; struct Nodo *sig; }NODO;</pre>	<pre>typedef double Elemento; Struct nodo { Elemento info; struct nodo *sig; };</pre>
---	---	---

Puntero de cabecera y cola. Un puntero al primer nodo se llama **puntero cabeza**. En ocasiones, se mantiene también un puntero al último nodo de una lista enlazada. El último nodo es la **cola** de la lista, y un puntero al último nodo es el **puntero cola**. Cada puntero a un nodo debe ser declarado como una variable puntero.

Puntero nulo. La palabra NULL representa el **puntero nulo** NULL, que es una constante de la biblioteca estándar `stdlib.h` de C. Este puntero se usa: en el campo `sig` del nodo final de una lista enlazada; en una lista vacía.

Operador -> de selección de un miembro. Si `p` es un puntero a una estructura y `m` es un miembro de la estructura, entonces `p -> m` accede al miembro `m` de la estructura apuntada por `P`. El símbolo “->” es un operador simple. Se denomina *operador de selección de componente*.

`P -> m` significa lo mismo que `(*p).m`

EJEMPLO 18.2 Construcción de una lista.

Un algoritmo para la creación de una lista enlazada añadiendo dato por el principio de la lista es el siguiente:
Declarar el tipo de dato y el puntero `ant` y `nl`

```
inicio
    ant ← NULL
    nl ← NULL
    mientras queden datos por añadir a la lista hacer
        Leer dato
        Asignar memoria para un elemento (nuevopuntero) utilizando a (malloc(), calloc(), realloc())
        nuevopuntero->info ← dato;
        nuevopuntero ->sig ← NULL
        si ant = NULL entonces
            ant ← nuevopuntero
```

```

    nl ← ant
    sino
    ant -> sig ← nuevopuntero
    nl ← nuevopuntero
  fin si
fin mientras
fin

```

18.3.1 INSERCIÓN DE UN ELEMENTO EN UNA LISTA

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

INSERCIÓN DE UN NUEVO ELEMENTO EN LA CABEZA DE UNA LISTA

El proceso de inserción se puede resumir en este algoritmo:

- Asignar un nuevo nodo a un puntero `ptrnodo` que apunte al nuevo nodo `nuevonodo` que se va a insertar en la lista.
- Situar el nuevo elemento en el campo `dato (el)` del nuevo nodo `nuevonodo`.
- Hacer que el campo enlace `sig` del nuevo nodo `nuevonodo` apunte a la cabeza (primer nodo) de la lista original.
- Hacer que `cabeza` (puntero `primero`) apunte al nuevo nodo que se ha creado.

EJEMPLO 18.3 *Inserción en la cabeza de la lista*

```

Nodo * ptrnodo;
ptrnodo = & nuevonodo;
nuevonodo.dato = el;
nuevonodo.sig = cabeza;
cabeza = ptrnodo;

```

INSERCIÓN DE UN NUEVO NODO QUE NO ESTÁ EN LA CABEZA DE LISTA

Se puede insertar en el centro o al final de la lista. El algoritmo de la nueva operación insertar requiere los pasos siguientes:

- Asignar memoria al nuevo nodo apuntado por el puntero `nuevo`.
- Situar el nuevo elemento en el campo `dato (el)` del nuevo nodo.
- Hacer que el campo enlace `sig` del nuevo nodo `nuevo` apunte al nodo que va después de la posición del nuevo nodo `nuevo`.
- En la variable puntero `ant` hay que tener la dirección del nodo que está antes de la posición deseada (siempre existe) para el nuevo nodo. Hacer que `ant -> sig` apunte al nuevo nodo que se acaba de crear.

INSERCIÓN AL FINAL DE LA LISTA

La inserción al final de la lista es un caso particular de la anterior. La única diferencia es que el enlace `sig` de nuevo nodo siempre apunta a `NULL`.

EJEMPLO 18.4 *Inserción en otro lugar de la lista*

```

nuevo = (Nodo*) malloc (sizeof(Nodo));
nuevo->dato = el;
nuevo->sig = ptrnodo->sig;
ant->sig = nuevo;

```

18.3.2 ELIMINACIÓN DE UN NODO EN UNA LISTA

El algoritmo para eliminar un nodo que contiene un dato se puede expresar en estos pasos:

- Buscar el nodo que contiene el dato. Hay que tener la dirección del nodo a eliminar y la del nodo inmediatamente anterior.

- El puntero `sig` del nodo anterior ha de apuntar al `sig` del nodo a eliminar.
- Si el nodo a eliminar es el `Primero`, se modifica `Primero` para que tenga la dirección del nodo `sig` del nodo a eliminar
- Se libera la memoria ocupada por el nodo.

EJEMPLO 18.5 Eliminación de un nodo

```
ant->sig = ptrnodo->sig;
if (Primero == ptrnodo) Primero = ptrnodo->sig;
free (ptrnodo);
```

18.4 Lista doblemente enlazada

En una **lista doblemente enlazada**, cada elemento contiene dos punteros, aparte del valor almacenado en el elemento; un puntero apunta al siguiente elemento de la lista (`sig`) y el otro puntero apunta al elemento anterior de la lista (`ant`). Existe una operación de *insertar* y *eliminar* (borrar) en cada dirección.

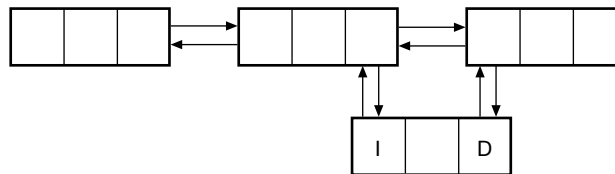


Figura 18.2 Inserción de un nodo en una lista doblemente enlazada.

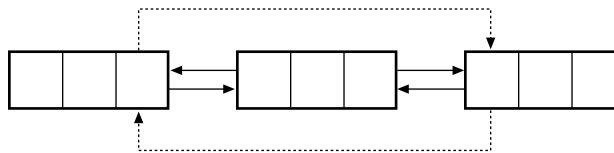


Figura 18.2 Eliminación de un nodo en una lista doblemente enlazada.

EJEMPLO 18.6 Declaración de una lista doblemente enlazada

Una lista doblemente enlazada con valores de tipo `int` necesita para ser declarada dos punteros (`sig`, `ant`) y el valor del campo elemento (`el`):

```
typedef int Item;
struct Unnodo
{
    Item el;
    struct Unnodo *sig;
    struct Unnodo *ant;
};
typedef struct Unnodo Nodo;
```

18.4.1 INSERCIÓN DE UN ELEMENTO EN UNA LISTA DOBLEMENTE ENLAZADA

El algoritmo empleado para añadir o insertar un elemento en una lista doble varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser: en la cabeza (elemento primero) de la lista; en el final de la lista (elemento último); antes de un elemento especificado, o bien después de un elemento especificado.

INSERCIÓN DE UN NUEVO ELEMENTO EN LA CABEZA DE UNA LISTA DOBLE

El proceso de inserción se puede resumir en este algoritmo:

- Asignar memoria a un nuevo nodo apuntado por `nuevo` que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista doble.
- Situar el nodo `nuevo` el elemento que se va a insertar en el campo `dato (el)` del nuevo nodo `nuevo`.
- Hacer que el campo enlace `sig` del nuevo nodo `nuevo` apunte a la cabeza (primer nodo) de la lista original, y que el campo enlace `ant` del nodo `cabeza` apunte al nuevo nodo `nuevo` si es que existe. En caso de que no exista no hacer nada.
- Hacer que `cabeza` (puntero de la lista) apunte al nuevo nodo que se ha creado.

EJEMPLO 18.7 *Inserción en cabeza de lista doble*

```
nuevo = (Nodo*) malloc (sizeof(Nodo));
nuevo->dato = el;
nuevo->sig = cabeza;
cabeza->ant = nuevo;
cabeza = nuevo;
```

INSERCIÓN DE UN NUEVO NODO QUE NO ESTÁ EN LA CABEZA DE LISTA

La inserción de un nuevo nodo en una lista doblemente enlazada se puede realizar en un nodo intermedio o final de ella. El algoritmo de la nueva operación insertar requiere las siguientes etapas:

- Asignar memoria al nuevo nodo apuntado por el puntero `nuevo`.
- Situar el nuevo elemento en el campo `dato (el)` del nuevo nodo `nuevo`.
- Hacer que el campo enlace `sig` del nuevo nodo `nuevo` apunte al nodo que va después de la posición del nuevo nodo `ptrnodo` (o bien a `NULL` en caso de que no haya ningún nodo después de la nueva posición). El campo `ant` del nodo siguiente `ptrnodo` al que ocupa la posición del nuevo nodo `nuevo` tiene que apuntar a `nuevo` si es que existe. En caso de que no exista no hacer nada.
- Tomar la dirección del nodo que está antes de la posición deseada para el nuevo nodo `nuevo`. Si esta dirección es la variable puntero `ant` y hacer que `ant->sig` apunte al nuevo nodo `nuevo`. El enlace `ant` del nuevo nodo `nuevo` ponerlo apuntando a `ant`.

EJEMPLO 18.8 *Inserción de un nodo en el centro de una lista doble*

En este caso siempre existe el nodo anterior `ant` y el nodo siguiente `ptrnodo`, con lo que la inserción supuesto que ya se han colocado los dos punteros es :

```
nuevo = (Nodo*) malloc (sizeof(Nodo));
nuevo->dato = el;
nuevo->sig = ptrnodo->sig;
ptrnodo->ant = nuevo;
ant->sig = nuevo;
```

18.4.2 ELIMINACIÓN DE UN ELEMENTO EN UNA LISTA DOBLEMENTE ENLAZADA

El algoritmo para eliminar un nodo que contiene un dato es similar al algoritmo de borrado para una lista simple. Ahora la dirección del nodo anterior se encuentra en el puntero `ant` del nodo a borrar. Los pasos a seguir son:

- Búsqueda del nodo que contiene el dato. Se ha de tener la dirección del nodo a eliminar y la dirección del anterior (`ant`).
- El puntero `sig` del nodo anterior (`ant`) tiene que apuntar al puntero `sig` del nodo a eliminar, `ptrnodo` esto en el caso de no ser el nodo primero de la lista. En caso de que sea el primero de la lista el puntero de la lista debe apuntar al puntero `sig` del nodo a eliminar `ptrnodo`.
- El puntero `ant` del nodo siguiente a borrar tiene que apuntar al puntero `ant` del nodo a eliminar, esto en el caso de no ser el nodo ultimo. En el caso de que el puntero a eliminar sea el último no hacer nada.

- Por último se libera la memoria ocupada por el nodo a eliminar `ptrnodo`.

EJEMPLO 18.9 Eliminación en una lista doble

```
ant->sig = ptrnodo->sig;
ptrnodo->sig->ant = ptrnodo->ant;
free (ptrnodo);
```

18.5 Listas circulares

En las listas lineales simples o en las dobles siempre hay un primer nodo y un último nodo que tiene el campo de enlace a nulo. Una lista circular, por propia naturaleza no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo a partir del cual se acceda a la lista y así poder acceder a sus nodos insertar, borrar etc.

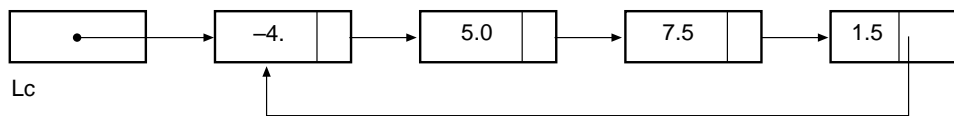


Figura 13.4 Lista circular.

INSERCIÓN DE UN ELEMENTO EN UNA LISTA CIRCULAR

El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar el elemento que inserta el nodo en la lista circular. En todo caso hay que seguir los siguientes pasos:

- Asignar memoria al nuevo nodo `nuevo` y almacenar el dato.
- Si la lista está vacía, enlazar el campo `sig` del nuevo nodo `nuevo` con el propio nuevo nodo, `nuevo` y poner el puntero de la lista circular en el nuevo nodo `nuevo`.
- Si la lista no está vacía se debe decidir el lugar donde colocar el nuevo nodo `nuevo`, quedándose con la dirección del nodo inmediatamente anterior `ant`. Enlazar el campo `sig` de nuevo nodo `nuevo` con el campo `sig` del nodo anterior `ant`. Enlazar el campo `sig` del nodo anterior `ant` con el nuevo nodo `nuevo`. Si se pretende que el nuevo nodo `nuevo` ya insertado sea el primero de la lista circular, mover el puntero de la lista circular al nuevo nodo `nuevo`. En otro caso no hacer nada.

EJEMPLO 18.10 Inserción en lista circular

```
nuevo = (Nodo*) malloc (sizeof(Nodo));
nuevo->dato = el;
if (primero==NULL)
{
    nuevo->sig = nuevo;
    primero = nuevo;
}
else {
    nuevo->sig = antanterior->sig;
    antanterior->sig = nuevo;
}
```

ELIMINACIÓN DE UN ELEMENTO EN UNA LISTA CIRCULAR

El algoritmo para eliminar un nodo de una lista circular es el siguiente:

- Buscar el nodo que contiene el dato quedándose con el nodo anterior `ant`.
- Se enlaza el campo `sig` del nodo anterior `ant` con el campo siguiente `sig` del nodo a borrar. Si la lista contenía un solo nodo se pone a NULL la lista.

- En caso de que el nodo a eliminar sea el referenciado por el puntero de acceso a la lista, `Lc`, y contenga más de un nodo se modifica `Lc` para que tenga la dirección del nodo anterior `ant` o bien el campo `sig` de `Lc`. (si la lista se quedara vacía hacer que `Lc` tome el valor `NULL`).
- Por último, se libera la memoria ocupada por el nodo.

EJEMPLO 18.11 *Eliminación en lista circular*

```
ant->sig = ptrnodo->sig;
if (Lc == Lc->sig)
    Lc=NULL;
else if (ptrnodo == Lc)
    Lc = ant->sig;
```

PROBLEMAS RESUELTOS

18.1. *Escriba una función que devuelva cierto si la lista está vacía y falso en otro caso, y otra que cree una lista vacía.*

Codificación

Si se supone siguiente declaración:

```
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro* sig;
}Nodo;
```

La codificación de la función `Esvacia` será:

```
Int Esvacia(Nodo * Primero)
{
    return( Primero == NULL);
}
```

La codificación de la función `VaciaL` será:

```
Void VaciaL(Nodo ** Primero)
{
    *Primero == NULL;
}
```

18.2. *Escriba una función entera que devuelva el número de nodos de una lista enlazada.*

Codificación

Si se supone la declaración del problema anterior se tiene:

```
int NumerNodos(Nodo *Primero)
{
    int k = 0;
    Nodo *p;
```

```

p = Primero;
while (p != NULL)
{
    k++;
    p = p->sig;
}
return(k);
}

```

18.3. Escriba una función que elimine el nodo que ocupa la posición i de una lista enlazada ordenada.

Análisis del problema

Para resolver el problema se necesita recorrer la lista contando el número de elementos que van pasando, y cortar el recorrido, cuando la lista esté vacía, o cuando se haya llegado a la posición que se busca. Una vez terminado el primer bucle de búsqueda en el caso de que haya que eliminar el elemento, se borra teniendo en cuenta si es o no el primer elemento de acuerdo con lo indicado en la teoría.

Codificación

Si se supone la declaración realizada en el problema 13,1 se tiene:

```

void EliminaPosicion (Nodo** Primero, int i)
{
    int k = 0;
    Nodo *ptr, *ant;

    ptr = *Primero;
    ant = NULL;
    while ( (k < i) && (ptr != NULL))
    {
        k++;
        ant = ptr;
        ptr = ptr->sig;
    }
    if(k == i)
    {
        if( ant == NULL)
            *Primero = ptr->sig;
        else
            ant->sig = ptr->sig;
        free(ptr);
    }
}

```

18.4. Escriba una función que reciba como parámetro una lista enlazada apuntada por *Primero*, un dato cualquiera *e* inserte en la lista enlazada un nuevo nodo con la información almacenada en *dato* y de tal forma que sea el primer elemento de la lista.

Análisis del problema

Los pasos que se seguirán son: asignar memoria a un nuevo puntero *nuevo*; situar el nuevo dato en el campo *el*; mover el campo *sig* de nuevo puntero *nuevo* al puntero *Primero* y hacer que *Primero* apunte a *nuevo*. Esta función trabaja correctamente, aún cuando la lista esté vacía, siempre que previamente se haya inicializado a *NULL*.

Codificación

Si se supone la siguiente declaración:

```
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro * sig;
}Nodo;
```

La codificación de la función será:

```
void InsertarprimeroLista(Nodo** Primero, Item dato)
{
    Nodo *nuevo ;
    nuevo = (Nodo*)malloc(sizeof(Nodo));
    nuevo -> el = dato;
    nuevo -> sig = *Primero;
    *Primero= nuevo;
}
```

- 18.5.** *Escriba una función que reciba como parámetro un puntero ant que apunte a un nodo de una lista enlazada e inserte el valor recibido en el parámetro dato como un nuevo nodo que esté inmediatamente después de ant (Inserción en el centro y final de una lista).*

Análisis del problema

Se crea un nuevo nodo apuntado por nuevo, donde se almacena el dato, para posteriormente poner como siguiente del nuevo nodo nuevo el siguiente de ant, para por último enlazar el siguiente de ant con nuevo.

Codificación

Si se supone la siguiente declaración:

```
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro* sig;
}Nodo;
```

La codificación de la función será:

```
void InsertarLista(Nodo* ant, Item dato)
{
    Nodo *nuevo;

    nuevo = (Nodo*)malloc(sizeof(Nodo));
    nuevo -> el = dato;
    nuevo -> sig = ant -> sig;
    ant -> sig = nuevo;
}
```

- 18.6.** *Escriba una función que reciba como datos un puntero al primer nodo de una lista enlazada y un dato a buscar y devuelva NULL si el dato no está en la lista y un puntero a la primera aparición del dato en otro caso.*

Análisis del problema

Mediante un bucle `for` controlado por la condición de fin de lista, se recorre la lista. En caso de encontrarlo se devuelve el puntero `pt`, en otro caso se devuelve `NULL`. Los parámetros son `Primero` que es puntero de cabeza de una lista enlazada, y `dato` que es el valor que se busca en la lista.

Codificación

Si se supone la siguiente declaración:

```
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro* sig;
}Nodo;
```

La codificación de la función será:

```
Nodo* BuscarLista (Nodo* Primero, Item dato)
{
    Nodo *ptr;

    for (ptr = Primero; ptr != NULL; ptr = ptr ->sig )
        if ( ptr->el == dato )
            return ptr;
    return NULL;
}
```

- 18.7.** *Escriba un programa que genere una lista enlazada de números aleatorios de tal manera que se almacenen en la lista en el orden en el que han sido generados. Posteriormente se presentará en pantalla toda la lista para después mostrar todos aquellos datos que ocupen una posición par en la lista.*

Análisis del problema

La solución se ha estructurado de la siguiente forma:

- Una función `InsertaListaDespues` que tiene como parámetros por referencia los punteros `Primero` y `ant`, que representan un puntero al primer elemento de la lista y otro al nodo inmediatamente anterior al último elemento de la lista, y como parámetro por valor `entrada` que representa el dato a insertar. Esta función inserta en la lista apuntada por `Primero` y después del puntero `ant` un nuevo nodo con la información dada en `entrada`, cambiando el puntero `ant`, y si es necesario el puntero `Primero`.
- Una función `NuevoNodo` que da un puntero a un nodo cuya información se le ha pasado en el parámetro `x`.
- Una función `EscribeLista` que recibe como parámetro un puntero a una lista enlazada y la presenta en pantalla.
- Una función `EscribeListapares` que recibe como parámetro un puntero a una lista enlazada y presenta en pantalla todos los elementos de la lista que ocupan una posición par.
- El programa principal se encarga de rellenar aleatoriamente la lista y realizar las llamadas correspondientes.

Codificación (Consultar la página web del libro)

18.8. *Escriba un programa que genere una lista aleatoria de números enteros, los inserte en una lista enlazada. Posteriormente nos presente la lista enlazada completa, y los elementos de la lista enlazada que son pares.*

Análisis del problema

Se declara una constante MX que será el máximo número entero aleatorio que se generará. El programa se ha estructurado de la siguiente forma:

- Una función `InsertaPrimero` insertará el dato que se ponga como entrada en una lista apuntada por el puntero `Primero`.
- Una función `NuevoNodo` dará un puntero a un nuevo registro en el que se ha almacenado la información que se le pase como dato.
- El programa principal, mediante un bucle `for` generará números aleatorios y los insertará en la lista hasta que se genere el número aleatorio cero. Mediante un bucle `for` se escriben todos los elementos de la lista enlazada, y mediante otro bucle `for` se escribe sólo aquellos elementos de la lista que sean pares.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 99
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro* sig;
}Nodo;

void InsertaPrimero(Nodo** Primero, Item entrada);
Nodo* NuevoNodo(Item x);

void main()
{
    Item d;
    Nodo *Primero, *ptr;
    int k;

    Primero = NULL;
    randomize();
    for (d = random(MX); d; ) // Termina cuando se genera el numero 0
    {
        InsertaPrimero(&Primero, d);
        d = random(MX);
    }
    printf("\n\n");
    printf(" se escriben todos los datos de la lista lista \n");
    for (k = 0, ptr = Primero; ptr; )
    {
        printf("%d ",ptr->el);
        k++;
        printf("%c",(k % 10?' ':'\n')); //cada 10 datos salta de línea */
        ptr = ptr->sig;
    }
    printf("\n\n");
```



```

printf(" se escriben los datos pares de la lista \n");
for (k = 0, ptr = Primero; ptr; )
{
    if (ptr->el%2 == 0)
    {
        printf("%d ", ptr->el);
        k++;
        printf("%c", (k%10?' ':'\n'));
    }
    ptr = ptr->sig;
}

void InsertaPrimero(Nodo** Primero, Item dato)
{
    Nodo *nuevo ;
    nuevo = NuevoNodo(dato);
    nuevo -> sig = *Primero;
    *Primero = nuevo;
}

Nodo* NuevoNodo(Item x)
{
    Nodo *a ;
    a = (Nodo*)malloc(sizeof(Nodo));
    a -> el = x;
    a -> sig = NULL;
    return a;
}

```

- 18.9.** *Escriba una función que reciba como parámetro una lista enlazada, y un dato, y borre de la lista enlazada la primera aparición del dato.*

Análisis del problema

Mediante un bucle mientras se encuentra si existe, el puntero `ptr` apunta al primer nodo que contiene el dato y en `ant` se queda con el puntero anterior. En caso de que no esté el dato en la lista (`ptr == NULL`) no se hace nada. En caso de que esté en la lista, se distinguen el caso de no ser el primero de la lista y el que lo sea, para por último liberar la memoria ocupada.

Codificación

```

void Suprime (Nodo** Primero, Item dato)
{
    Nodo* ptr, *ant;
    int enc = 0;

    ptr = *Primero;
    ant = ptr;
    while (( !enc) && (ptr != NULL))
    {
        enc = (ptr->el == dato);
        if (!enc)
        {

```

```

        ant = ptr;
        ptr = ptr -> sig;
    }
}
if (ptr != NULL)
{
    if (ptr == *Primero)
        *Primero = ptr->sig;
    else
        ant -> sig = ptr->sig;
    free(ptr);
}
}

```

18.10. *Escriba un programa que lea del teclado una lista de números enteros los inserte en una lista enlazada ordenada crecientemente, presente la lista enlazada ordenada, y pida una sucesión de datos del teclado que serán borrados de la lista ordenada.*

Análisis del problema

La solución se ha planteado de la siguiente forma:

- *Un programa principal se encarga de hacer las declaraciones y llamar a las distintas funciones en bucles `do while`. El fin de la entrada de datos viene dado por el centinela `-1`.*
- *La función `NuevoNodo` se encarga de crear un nodo donde almacenar el dato que recibe como parámetro, y coloca el campo siguiente a `NULL`.*
- *La función `Escribir` se encarga de presentar en pantalla la lista enlazada ordenada.*
- *La función `InsertarEnOrden` recibe como parámetro una lista enlazada ordenada y un dato y lo inserta dejándola de nuevo ordenada. Para realizarlo, primeramente crea el nodo donde almacenará el dato, si es el primero de la lista lo inserta, y en otro caso mediante un bucle `while` recorre la lista hasta encontrar donde colocar el dato. Una vez encontrado el sitio se realiza la inserción de acuerdo con el algoritmo correspondiente.*
- *La función `borrarEnOrden` se encarga de buscar la primera aparición de dato en una lista enlazada ordenada y borrarlo. Para ello realiza la búsqueda de la posición donde se encuentra la primera aparición del dato quedándose con el puntero `ptr` y con el puntero `ant` (anterior). Posteriormente realiza el borrado teniendo en cuenta que sea el primero de la lista o que no lo sea.*

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 100
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro* sig;
}Nodo;

void InsertarEnOrden(Nodo** Primero, Item Dato);
Nodo* NuevoNodo(Item x);
void Escribir(Nodo* Primero);
void BorrarEnOrden(Nodo** Primero, Item Dato);

```

```

void main()
{
    Item d;
    Nodo* Primero;
    Primero = NULL;
    randomize();
    do
    {
        printf(" dato -1 = fin\n");
        scanf("%d", &d);
        if (d != -1)
            InsertarEnOrden(&Primero, d);
    }
    while (d != -1);
    Escribir(Primero);
    do
    {
        printf(" dato a borrar -1 = fin\n");
        scanf("%d", &d);
        if (d != -1)
            BorrarEnOrden(&Primero, d);
        Escribir(Primero);
    }
    while (d != -1);
}

void InsertarEnOrden(Nodo** Primero, Item Dato)
{
    Nodo *nuevo, *ant, *p;

    nuevo = NuevoNodo(Dato);
    if (*Primero == NULL)
        *Primero = nuevo;
    else
        if (Dato <= (*Primero)->el)
        {
            nuevo -> sig = *Primero;
            *Primero = nuevo;
        }
        else
        {
            ant = p = *Primero;                                // no es el primero
            while ((Dato > p->el) && (p->sig != NULL) )
            {
                ant = p;
                p = p->sig;
            }
            if (Dato > p->el)                                     // falta por comprobar el ultimo
                ant = p;
            nuevo -> sig = ant -> sig;
            ant -> sig = nuevo;
        }
}

```

```
void BorrarEnOrden(Nodo** Primero, Item Dato)
{
    Nodo *ant, *ptr;
    int enc = 0;

    ant = NULL;
    ptr = *Primero;
    while ((! enc) && (ptr != NULL))
    {
        enc = (Dato <= ptr->el);
        if (! enc)
        {
            ant = ptr;
            ptr = ptr->sig;
        }
    }
    if ( enc )
        enc = ((ptr->el) == Dato);
    if ( enc )
    {
        if (ant == NULL)
            *Primero = ptr->sig;
        else
            ant->sig = ptr->sig;
        free(ptr);
    }
}

Nodo* NuevoNodo(Item x)
{
    Nodo *n ;

    n = (Nodo*)malloc(sizeof(Nodo));
    n -> sig = NULL;
    n -> el = x;
    return n;
}

void Escribir(Nodo* Primero)
{
    int k;

    printf("\n\t\t Lista Ordenada \n");
    for (k = 0; Primero; Primero = Primero->sig)
    {
        k++;
        printf("%d ", Primero->el);
        printf("%c", (k%10 ? ' ' : '\n'));
    }
    printf("\n\n");
}
```

- 18.11.** *Escriba una función que reciba como dato una lista enlazada y un dato y elimine todos los nodos de la lista que cumplan la condición de que su información sea estrictamente mayor que el valor dado en dato.*

Análisis del problema

La función usa tres punteros: `ptr` que es el puntero con el que se recorre la lista, `ant` que es el puntero inmediatamente anterior a `ptr`, y `p` que es el puntero usado para liberar memoria. La codificación se ha realizado de tal forma que el puntero `ptr` recorre la lista mediante un bucle `while`, cuando un nodo cumple la condición de ser eliminado, se borra de la lista, teniendo siempre en cuenta si es el primer elemento de la lista o no lo es. Si el nodo no cumple la condición de borrado simplemente se avanza en la lista.

Codificación

```
void EliminaMayores(Nodo **Primero, Item Dato)
{
    Nodo *ptr, *ant, *p;
    Ant = NULL;
    Ptr = *Primero;
    while (ptr != NULL)
        if (ptr->el > Dato)
            if (ant == NULL)
            {
                p = *Primero;
                *Primero = ptr->sig;
                ptr = *Primero;
                free(p);
            }
            else
            {
                ant->sig = ptr->sig;
                p = ptr;
                ptr = ptr->sig;
                free(p);
            }
            else
            {
                ant = ptr;
                ptr = ptr->sig;
            }
    }
}
```

- 18.12.** *Un conjunto es una secuencia de elementos todos del mismo tipo, sin duplicados. Escriba un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las operaciones: `EscribeConjunto`; `AnadeConjunto`; `PerteneceConjunto`; `BorraConjunto`; `CardinalConjunto`; `VaciaConjunto`; `EsVacioConjunto`; `RellenaConjunto`.*

Análisis del problema

El programa que se codifica representa los conjuntos como listas simplemente enlazadas sin ordenar. La implementación puede mejorarse en cuanto a eficiencia si se implementan como listas enlazadas ordenadas crecientemente, o bien como árboles binarios de búsqueda AVL. La solución se divide en los siguientes módulos:

- Un programa principal que se encarga de llamar a los distintos módulos de programa.
- La función `VaciaConjunto` que crea el conjunto vacío creando una lista vacía.
- La función `EsVacioConjunto` que decide si un conjunto es vacío.
- La función `RellenaConjunto` que añade aleatoriamente elementos a un conjunto.
- La función `CardinalConjunto` que nos dice cuantos elementos hay almacenados en el conjunto.
- La función `BorraConjunto` que se encarga de borrar un elemento del conjunto.
- La función `PerteneceConjunto` que decide si un elemento se encuentra en el conjunto.
- La función `AnadeConjunto` que se encarga de añadir un elemento al conjunto como primer elemento.
- La función `EscribeConjunto` que se encarga de escribir los elementos que se encuentran en el conjunto.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 99
typedef int Item;
typedef struct Registro
{
    Item el;
    struct Registro* sig;
} Nodo;

Nodo* NuevoNodo(Item x);
void EscribeConjunto(Nodo * Primero);
void AnadeConjunto(Nodo** Primero, Item dato);
int PerteneceConjunto (Nodo* Primero, Item dato);
void BorraConjunto (Nodo** Primero, Item dato);
int CardinalConjunto(Nodo *Primero);
void VaciaConjunto(Nodo **Primero);
int EsVacioConjunto(Nodo *Primero);
void RellenaConjunto(Nodo **Primero);

void main()
{
    Nodo *Primero;
    RellenaConjunto(&Primero);
    EscribeConjunto(Primero);
    ....
    ....
    ....
}

void RellenaConjunto(Nodo **Primero)
{
    Item d;
    *Primero = NULL;
    randomize();
    for (d = random(MX); d; )
    {
        if (!PerteneceConjunto(*Primero, d))
            AnadeConjunto(Primero,d);
        d = random(MX);
    }
}
```

```

    }
}

int EsVacioConjunto(Nodo *Primero)
{
    return (Primero == NULL);
}

void VaciaConjunto(Nodo **Primero)
{
    (*Primero)= NULL;
}

int CardinalConjunto(Nodo *Primero)
{
    Nodo *ptr;
    int k = 0;
    ptr = Primero;
    while ( ptr )
    {
        k++;
        ptr = ptr->sig;
    }
    return k;
}

Nodo* NuevoNodo(Item x)
{
    Nodo *a ;

    a = (Nodo*)malloc(sizeof(Nodo));
    a -> el = x;
    a -> sig = NULL;
    return a;
}

void EscribeConjunto(Nodo * Primero)
{
    int k; Nodo *ptr;

    printf("\n\n");
    printf(" se escriben todos los elementos del conjunto \n");
    for (k = 0, ptr = Primero; ptr ; )
    {
        printf("%d ", ptr->el);
        k++;
        if ( k % 10)
            printf("%c",' ');
        else
            printf("%c",' \n');
        ptr = ptr->sig;
    }
}

void AnadeConjunto(Nodo** Primero, Item dato)

```

```

{
    Nodo *nuevo;
    if(!PerteneceConjunto(*Primero, dato))
    {
        nuevo = (Nodo*)malloc(sizeof(Nodo));
        nuevo -> el = dato;
        nuevo -> sig = *Primero;
        *Primero = nuevo;
    }
}

int PerteneceConjunto (Nodo* Primero, Item dato)
{
    Nodo *ptr;
    for (ptr = Primero; ptr != NULL; ptr = ptr ->sig )
        if (ptr-> el == dato)
            return 1;
    return 0;
}

void BorraConjunto (Nodo** Primero, Item dato)
{
    Nodo* ptr, *ant;
    int enc = 0;
    ptr = *Primero; ant = ptr;
    while ((! enc) && (ptr != NULL))
    {
        enc = (ptr->el == dato);
        if (! enc)
        {
            ant = ptr;
            ptr = ptr -> sig;
        }
    }
    if (ptr != NULL)
    {
        if (ptr == *Primero)
            *Primero = ptr->sig;
        else
            ant -> sig = ptr->sig;
        free(ptr);
    }
}

```

18.13. Con la representación de Conjuntos realizada en el ejercicio anterior, añade las operaciones básicas: Unión, Intersección, Diferencia, Inclusión.

Análisis del problema

Se codifican a continuación las siguientes funciones

- **UnionDeConjuntos** que realiza la unión del conjunto C1 con el C2 en el Conjunto C3. Para realizarlo lo único que se hace es añadir todos los elementos del conjunto C1 y C2 al conjunto C3, previamente inicializado a NULL.

- `DiferenciaDeConjunto` que realiza la diferencia del conjunto C1 con el C2 dejando el resultado en C3. Por lo tanto C3 contendrá todos los elementos de C1 que no estén en C2.
- `InclusionDeConjuntos` que decide si el conjunto C1 está incluido en el C2. Para que esto ocurra deben estar todos los elementos de C1 en C2.
- `InterseccionDeConjuntos` que pone en C3 la intersección de los conjuntos C1 y C2. Por lo tanto C3 contendrá todos los elementos de C1 que estén a su vez en C2.

Codificación (Consultar la página web del libro)

18.14. *Escriba una función que reciba como parámetro dos listas enlazadas ordenas crecientemente y de como resultado otra lista enlazada ordenada que sea mezcla de las dos.*

Análisis del problema

Para mezclar dos listas enlazadas, se usa un nodo ficticio apuntado por el puntero `p`, para asegurar que todos los elementos se insertarán al final de la lista que será la mezcla. Para ello se lleva un puntero `u` que apuntará siempre al último elemento de la lista que debido al nodo ficticio siempre existirá. Al final de la mezcla se elimina el elemento ficticio. La mezcla de las dos listas se realiza avanzando con dos punteros `p1` y `p2` por las listas `L1` y `L2`. Un primer bucle `while` avanzará o bien por `L1` o bien por `L2` insertando en la lista mezcla, dependiendo de que el dato más pequeño esté en `L1` o en `L2`, hasta que una de las dos listas se termine. Los dos bucles `while` posteriores se encargan de terminar de añadir a la lista mezcla los elementos que queden o bien de `L1` o bien de `L2`.

Se usa la función `NuevoNodo` y las declaraciones de los problemas vistos anteriormente.

Codificación

```
void MezclarListasOrdenadas(Nodo *L1, Nodo *L2, Nodo **L3)
{
    Nodo *p1, *p2, *p, *u, *nn;

    nn = NuevoNodo(-32767);
    p = nn;
    u = nn;
    p1 = L1;
    p2 = L2;
    while (p1 && p2)
        if (p1->el < p2->el)
        {
            nn = NuevoNodo(p1->el);
            u->sig = nn;
            u = nn;
            p1 = p1->sig;
        }
        else
        {
            nn = NuevoNodo(p2->el);
            u->sig = nn;
            u = nn;
            p2 = p2->sig;
        }
    while ( p1 )
    {
        nn = NuevoNodo(p1->el);
        u->sig = nn;
        u = nn;
    }
    while ( p2 )
    {
        nn = NuevoNodo(p2->el);
        u->sig = nn;
        u = nn;
    }
}
```

```

    p1 = p1->sig;
}
while ( p2 )
{
    nn = NuevoNodo(p2->el);
    u->sig =nn;
    u = nn;
    p2 = p2->sig;
}
*L3 = p->sig;
free(p);
}

```

18.15. Implementar un programa C que tenga las siguientes opciones de una Lista Doblemente Enlazada. *InsertaPrincipioLD; VacíaLD; EsVacíaLD; generaPorElFinalLD; generaPorElPrincipioLD; InsertaAntLD; EscribeLista; InsertaListaDespuesLD*

Análisis del problema

El programa que se codifica declara una lista doblemente enlazada, y un programa principal se encarga de llamar a las distintas funciones que a continuación se especifica.

- *InsertaPrincipioLD*. Recibe como parámetro un puntero a una lista doble, un dato y realiza la inserción de un nuevo nodo en la lista como primer elemento cuya información es el valor recibido en dato.
- *VacíaLD*. Es una función que recibe como parámetro una lista doble y la pone a vacía.
- *EsvacíaLD*. Es una función que recibe como parámetro una lista doble y decide si está vacía.
- *GeneraPorElPrincipioLD*. Es una función que usando la función que usando la función *InsertaPrincipioLD*, genera números aleatorios y los inserta en una lista doble por el principio.
- *InsertaListaDespuesLD*. Es una función que recibe como parámetro un puntero *Primero* a una Lista enlazada Doble y un puntero *ant* que apunta al último nodo de la Lista Doble, e inserta un nuevo nodo como último elemento de la Lista Doble cuya información está dada por el valor de *d*. Además de realizar la inserción mueve adecuadamente los punteros *Primero* y *ant*.
- *GeneraPorElFinal*. Es una función que usando la función *InsertaListaDespuesLD*, genera números aleatorios y los inserta en una lista doble por el final.
- *EscribeLista*. Escribe los elementos de la lista doblemente enlazada.
- *InsertaAntLD*. Inserta después del nodo apuntado por *ant* que siempre existe un nuevo nodo cuya información es la recibida por dato.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 100
typedef int Item;
typedef struct unnodo
{
    Item el;
    struct unnodo* sig, *ant;
}Nodo;

void InsertaPrincipioLD(Nodo ** Primero, Item dato);
void VacíaLD(Nodo ** Primero);
int EsVacíaLD(Nodo *Primero);

```

```

void GeneraPorElFinalLD(Nodo **Primero);
void GeneraPorElPrincipioLD(Nodo **Primero);
void InsertaAntLD( Nodo* ant, Item dato);
void EscribeLista(Nodo * Primero);
void InsertaListaDespuesLD(Nodo **Primero,Nodo **ant, Item d);

void main(void)
{
    Nodo *Primero;
    GeneraPorElPrincipioLD( &Primero);
    EscribeLista(Primero);
}

void VacialD(Nodo ** Primero)
{
    *Primero = NULL;
}

int EsVacialD(Nodo *Primero)
{
    return (Primero == NULL);
}

void InsertaAntLD( Nodo* ant, Item dato)
{
    Nodo *ptr, *sig;

    ptr = (Nodo*)malloc(sizeof(Nodo));
    ptr->el = dato ;
    ptr->ant = ant;
    ptr->sig = ant->sig;
    sig=ptr->sig;
    if (sig != NULL)
        sig->ant = ptr;
    ptr->sig = ptr;
}

void InsertaPrincipioLD(Nodo ** Primero, Item dato)
{
    Nodo* nn;

    nn = (Nodo*)malloc(sizeof(Nodo));
    nn->el = dato ;
    nn->sig = *Primero;
    nn->ant = NULL;
    if (*Primero == NULL)
        *Primero = nn;
    else
        (*Primero) ->ant = nn;
    *Primero = nn;
}

void EscribeLista(Nodo * Primero)

```

```
{
    int k;
    Nodo *ptr;
    printf("\n\n");
    printf(" se escriben todos los datos de la lista lista  \n");
    for (k =0, ptr = Primero; ptr; )
    {
        printf("%d ",ptr->el);
        k++;
        if (k %10)
            printf("%c",' ');
        else
            printf("%c",' \n');
        ptr = ptr->sig;
    }
}

void InsertaListaDespuesLD(Nodo **Primero,Nodo **ant, Item d)
{
    Nodo *nuevo ;

    nuevo = (Nodo*)malloc(sizeof(Nodo));
    nuevo->el = d;
    nuevo->sig = NULL;
    if(*ant == NULL)
    {
        *ant = nuevo;
        *Primero = ant;
        nuevo->ant = NULL ;
    }
    else
    {
        (*ant)->sig = nuevo;
        nuevo->ant =*ant;
        *ant = nuevo;
    }
}

void GeneraPorElFinalLD(Nodo **Primero)
{
    Item d;
    Nodo *p,*ptr;

    p = NULL;
    ptr = NULL;
    randomize();
    for (d = random(MX); d; )
    {
        InsertaListaDespuesLD(&p, &ptr, d);
        d = random(MX);
    }
    *Primero =p;
}
```

```

void GeneraPorElPrincipioLD(Nodo **Primero)
{
    Item d;
    Nodo *p;

    p = NULL;
    randomize();
    for (d=random(MX); d; )
    {
        InsertaPrincipioLD(&p, d);
        d = random(MX);
    }
    *Primero=p;
}

```

18.16. *Escriba una función que reciba como parámetro un puntero a una Lista Doblemente Enlazada además de un valor almacenado en dato, y elimine la primera aparición de ese dato en la lista Doble.*

Análisis del problema

En primer lugar se procede a buscar la primera aparición del elemento en la Lista Doblemente Enlazada. Una vez que se haya encontrado, se resuelve el problema del borrado al comienzo de la lista moviendo el puntero *Primero y si es necesario el puntero ptr->sig. Posteriormente se resuelve el problema de borrado en el centro de la lista, para lo cual hay que mover los punteros ptr->ant->sig y ptr->sig->ant. Por último se resuelve el problema del borrado al final de la lista moviendo el puntero ptr->sig. Una vez que se han realizado los enlaces, se libera la memoria.

Codificación

```

void EliminaLD(Nodo **Primero, Item dato)
{
    Nodo* ptr;
    int enc = 0;

    ptr = *Primero; final de línea //busqueda
    while ((ptr!=NULL) && (!enc))
    {
        enc = (ptr->el == dato);
        if (!enc)
            ptr = ptr->sig;
    }
    if (ptr != NULL)
    {
        if (ptr == *Primero) // comienzo
        {
            *Primero = ptr->sig;
            if (ptr->sig != NULL)
                ptr->sig->ant = NULL;
        }
        else
        {
            if (ptr->sig != NULL) // centro
            {
                ptr->ant->sig = ptr->sig;
                ptr->sig->ant = ptr->ant;
            }
        }
    }
}

```



```

        {
            ant->sig = ptr->sig;
            ptr->sig->ant = ant;
        }
        free(ptr);
    }
}

```

18.18. *Escriba una función que inserte un número entero en una lista Doblemente enlazada y ordenada crecientemente.*

Análisis del problema

En primer lugar se llama a la función `NuevoNodoLD` que reservará la memoria donde almacenará el Dato y se inicializarán los campos `ant` y `sig` a `NULL` del puntero `nuevo` donde se ha almacenado el dato. Mediante un bucle `while` controlado por la variable lógica `enc` y el puntero `p`, que inicialmente se ponen a falso y a la propia lista, se busca la posición donde se debe insertar el nuevo dato. Además hay que quedarse en el puntero `ant` con la posición del nodo anterior. Una vez encontrada la posición, la inserción se puede producir en; a) principio y final b) caso principio y no final c) caso final y no principio d) caso no principio y no final. Se codifica también la función `NuevoNodoLD` y la función `GegeraOrdenadoLD`, que se encarga de generar aleatoriamente la lista doblemente enlazada ordenada.

Codificación

```

#define MX 100
typedef int Item;
typedef struct unnodo
{
    Item el;
    struct unnodo* sig, *ant;
}Nodo;
.....
.....

Nodo* NuevoNodoLD(Item x)
{
    Nodo *n;

    n = (Nodo*)malloc(sizeof(Nodo));
    n->sig = NULL;
    n->ant = NULL;
    n->el = x;
    return n;
}

void InsertarEnOrdenLD(Nodo** Primero, Item Dato)
{
    Nodo *nuevo, *ant, *p;
    int enc;
    nuevo = NuevoNodoLD(Dato);
    p = *Primero;
    ant = NULL;
    enc = 0;
    while ((! Enc ) && (p != NULL))
    {

```

```

        enc = (Dato <= p->el);
        if (!enc)
        {
            ant = p;
            p = p->sig;
        }
    }
    if (*Primero == NULL) // primero y ultimo
        *Primero = nuevo;
    else
        if (ant == NULL) // primero no ultimo
        {
            nuevo->sig = p;
            p->ant = nuevo;
            *Primero = nuevo;
        }
    else
        if (p == NULL) // ultimo no primero
        {
            nuevo->ant = ant;
            ant->sig = nuevo;
        }
        else // no ultimo no primero
        {
            nuevo->sig = p;
            nuevo->ant = ant;
            ant->sig = nuevo;
            p->ant = nuevo;
        }
}

void GeneraOrdenadaD(Nodo **Primero)
{
    Item d;
    Nodo *p;

    p = NULL;
    randomize();
    for (d = random(MX); d; )
    {
        InsertarEnOrdenLD(&p, d);
        d = random(MX);
    }
    *Primero = p;
}

```

18.19. *Se tiene una lista simplemente enlazada de números reales. Escriba una función para obtener una lista doblemente enlazada ordenada respecto del campo `el`, con los valores de la lista simple.*

Análisis del problema

Para resolver el problema planteado simplemente hay que cambiar la función `GeneraOrdenadaLD` hecha en el ejercicio 18.18, de tal manera que reciba además la lista simplemente enlazada como parámetro, y en lugar del bucle `for` recorrer la lista simple mediante un bucle `while` por ejemplo realizando la misma llamada.

Codificación

```

typedef int Item;
typedef struct unnodo
{
    Item el;
    struct unnodo* sig, *ant;
}Nodo;

typedef struct unnodols
{
    Item el;
    struct unnodols* sig;
}Nodols;
.....
.....
void GeneraOrdenadaLDaPartirSimple(Nodo**Primero,Nodols*Primerols)
{
    Nodo *pId;
    Nodols *pls

    pId = NULL;
    pls =Primerols;
    while (pls)
    {
        InsertarEnOrdenLD(&pId, pls->el);
        pls = pls->sig;
    }
    *Primero = p;
}

```

18.20. *Escriba una función que tenga como parámetro el puntero Primerols al primer nodo de una lista simplemente enlazada y retorne un puntero a una lista doble con los mismos campos que la lista enlazada simple pero en orden inverso.*

Análisis del problema

Para resolver el problema sólo que recorrer la lista enlazada simple e ir insertando en una lista doble por el final.

Codificación

```

typedef int Item;
typedef struct unnodo
{
    Item el;
    struct unnodo* sig, *ant;
}Nodo;

typedef struct unnodols
{
    Item el;
    struct unnodols* sig;
}Nodols;

```

```

....
....
....
void InsertaListaDespuesLD(Nodo **Primero, Nodo **ant, Item d)
{
    Nodo *nuevo ;

    nuevo = (Nodo*)malloc(sizeof(Nodo));
    nuevo->el = d;
    nuevo->sig = NULL;
    if(*ant == NULL)
    {
        *ant = nuevo;
        *Primero = *ant;
        nuevo->ant = NULL;
    }
    else
    {
        (*ant)->sig = nuevo;
        nuevo->ant = *ant;
        *ant=nuevo;
    }
}

nodo * GeneraPorElFinalLD(NodoIs *Primero)
{
    Item d;
    Nodo *p,*ptr;
    NodoIs ls;

    p = NULL;
    ptr = NULL;
    ls = Primero;
    while(ls != NULL)
    {
        InsertaListaDespuesLD(&p, &ptr, ls->el);
        ls = ls->sig;
    }
    return(p);
}

```

18.21. *Escriba las declaraciones y funciones necesarias para trabajar con una lista circular.*

Análisis del problema

El programa que se presenta se estructura de la siguiente forma:

- En primer lugar se realizan las declaraciones necesarias para tratar la Lista Circular.
- El Programa principal se encarga de realizar las llamadas correspondientes.
- VacíaLc. Es una función que nos crea una Lista Circular vacía.
- EsVacíaLc. Es una función que da verdadero cuando la Lista Circular está vacía.
- NuevoNodoLc. Es una función que devuelve un puntero a un nuevo nodo en el que se ha almacenado el dato x.
- InsertaListaCircular. Realiza la inserción en una lista circular del valor dato. Lo hace teniendo en cuenta que Primero es un puntero que apunta al último elemento que se añadió a la lista, y a continuación inserta un nuevo nodo en

la Lista Circular como último elemento, para lo cual aparte de realizar los correspondientes enlaces, debe mover el puntero `Primero` para que apunte siempre al último elemento que se añadió. De esta forma el primer elemento de la lista siempre estará en el nodo `Primero->sig`.

- `GeneraPorElFinalLc`. Crea una lista circular de números enteros aleatorios, realizando las inserciones con la función `InsertaListaCircular`.
- `EscribeListaLc`. Se encarga de escribir la lista circular. Si la Lista Circular está vacía no hace nada. En otro caso lo que hace es mediante un puntero `ptr`, se toma el primer elemento que estará siempre en el `sig` del puntero que apunte a la lista circular, y mediante un bucle escribe el dato, y avanza `ptr` hasta que haya dado una vuelta completa.
- `EliminaPrimeroLc`. Se encarga de eliminar el primer nodo de la lista circular que estará siempre en `Primero->sig`. Sólo hay que tener en cuenta que si la lista está vacía no se puede borrar. Si tiene un sólo dato la lista se quedará vacía y habrá que liberar memoria. Si tiene más de un dato habrá que mover el puntero `Primero->sig` a `Primero->sig->sig` y liberar la memoria del nodo que se ha puentado.
- `EliminarLc`. Se encarga de buscar la primera aparición de dato y borrarla de la lista circular. Lo hace de la siguiente forma. Si la lista esta vacía no hay nada que hacer. En otro caso con una variable lógica `enc` y con un puntero `ptr` realiza la búsqueda del dato, controlando no realizar un bucle infinito. Una vez encontrado el elemento se realiza el borrado teniendo en cuenta: Si la lista contiene un solo valor se quedará vacía. Si el nodo a borrar es el apuntado por `Primero`, habrá que mover este puntero, en otro caso no habrá que moverlo. Siempre que se borre un nodo habrá que puentarlo.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 100
typedef int Item;
typedef struct NuevoNodo
{
    Item el;
    struct NuevoNodo* sig;
}NodoLc;

void Vacialc(NodoLc ** Primero);
int EsVacialc(NodoLc *Primero);
NodoLc* NuevoNodoLc(Item x);
void InsertaListaCircular(NodoLc ** Primero,Item dato);
void GeneraPorElFinalLc(NodoLc **Primero);
void EscribeListaLc(NodoLc * Primero);
void EliminarLc (NodoLc** Primero,Item dato);
void EliminaPrimeroLc( NodoLc **Primero);

void main(void)
{
    NodoLc *Primero;

    GeneraPorElFinalLc(&Primero);
    EscribeListaLc(Primero);
}

void Vacialc(NodoLc ** Primero)
{
    *Primero = NULL;
}
```

```
int EsVaciaLc(NodoLc *Primero)
{
    return (Primero == NULL);
}

NodoLc* NuevoNodoLc(Item x)
{
    NodoLc *nn ;

    nn = (NodoLc*)malloc(sizeof(NodoLc));
    nn -> el = x;
    nn -> sig = nn;
    return nn;
}

void InsertaListaCircular(NodoLc ** Primero,Item dato)
{
    NodoLc* nn;

    nn = NuevoNodoLc(dato);
    if (*Primero != NULL)
    {
        nn -> sig = (*Lc) -> sig;
        (*Primero) -> sig = nn;
    }
    *Primero = nn;
}

void GeneraPorElFinalLc(NodoLc **Primero)
{
    Item d;
    NodoLc *p;

    p = NULL;
    randomize();
    for (d=random(MX); d; )
    {
        InsertaListaCircular(&p,d);
        d = random(MX);
    }
    *Primero=p;
}

void EscribeListaLc(NodoLc * Primero)
{
    NodoLc *ptr;
    int k = 0;

    ptr = Primero;
    if (ptr != NULL)
    {
        ptr = ptr->sig;
        do
```

```

    {
        k++;
        if(k%10==0)
        {
            printf("\n");
            printf("%d", ptr->el);
        }
        else
        {
            printf(" ");
            printf("%d", ptr->el);
        }
        ptr = ptr->sig;
    }
    while (ptr != Primero->sig);
}

void EliminaPrimeroLc( NodoLc **Primero)
{
    NodoLc *ptr, *p;
    ptr = *Primero;

    if (ptr!=NULL)
    {
        p=ptr->sig; // p hay que borrarlo
        if (p == ptr)
            *Primero = NULL;
        else
            ptr->sig = p->sig;
        free(p);
    }
}

void EliminarLc (NodoLc** Primero, Item dato)
{
    NodoLc* ptr,*p;
    int enc = 0;
    ptr = *Primero;
    if (ptr == NULL)
        return;
    // búsqueda mientras no encontrado y no de la vuelta
    while ((ptr->sig != *Primero) && (!enc))
    {
        enc = (ptr->sig->el == dato);
        if (!enc)
            ptr = ptr -> sig;
    }
    enc = (ptr->sig->el == dato); // aquí se debe encontrar el dato
    if (enc)
    {
        p = ptr->sig;
        if (*Primero == (*Primero)->sig)           // solo hay un dato

```

```

        *Primero = NULL;
    else
    {
        if (p == *Primero)
            *Primero = ptr;
        ptr->sig = p->sig;
    }
    free(p);
}
}

```

18.22. *Se tiene una Lista Circular de palabras. Escribir una función que cuente el número de veces que una palabra dada está en la lista.*

Análisis del problema

Para resolver el problema basta con declarar `item` de tipo cadena y recorrer la lista circular contando el número de apariciones que tiene. Para realizar las comparaciones se usa la función `strcmp()` que recibe como parámetro dos cadenas de caracteres y devuelve el valor 0 si son iguales. Si la lista está vacía devolverá el valor cero. En otro caso, mediante un bucle `while` controlado por “dar la vuelta a la lista”, se va comprobando y contando las igualdades entre cadenas con la función `strcmp()`.

Codificación

```

#include <stdio.h>
#include <string.h>

typedef char *Item;
typedef struct NuevoNodo
{
    Item el;
    struct NuevoNodo* sig;
}NodoLc;

int AparicoponesEnLc(NodoLc *Primero, char *cad)
{
    int cont = 0;
    NodoLc *ptr;

    ptr = Primero;
    if(ptr == NULL)
        return (cont);
    else
    {
        if (strcmp(ptr->el,cad) == 0)
            cont++;
        // mientras no de la vuelta
        while (ptr->sig != Primero)
        {
            ptr = ptr->sig;
            if(strcmp(ptr->el,cad) == 0)
                cont++;
        }
    }
}

```

```

    }
    return(cont);
}
}

```

18.23. Escriba una función que tenga como argumento una lista circular de números enteros. La función debe devolver el dato del nodo con mayor valor.

Análisis del problema

Para resolver el problema basta con recorrer la lista circular almacenando el valor máximo. Si la lista está vacía se devuelve un valor muy negativo equivalente a menos infinito. En otro caso, el mayor es `ptr->el`, (previamente `ptr` toma el valor de `Primero`) y mediante un bucle `while` controlado por “dar la vuelta a la lista”, se calcula el nuevo mayor por el algoritmo voraz clásico.

Codificación

```

Item MayorLc(NodoLc *Primero)
{
    Item Mayor;
    NodoLc *ptr;
    ptr=Primero;
    if(ptr == NULL)
        return (-32767);
    else
    {
        Mayor = ptr->el;
                                                                    // mientras no de la vuelta

        while (ptr->sig != Primero)
        {
            ptr = ptr->sig;
            if(Mayor <(ptr->el))
                Mayor = ptr->el;
        }
        return(Mayor);
    }
}

```

PROBLEMAS PROPUESTOS

- 18.1.** En una lista enlazada de números enteros se desea añadir un nodo entre dos nodos consecutivos con campos dato de distinto signo; el valor del campo dato del nuevo nodo que sea la diferencia en valor absoluto.
- 18.2.** Escribir una función para crear una lista doblemente enlazada de palabras introducidas por teclado. La función debe tener un argumento puntero `Ld` en el que se devuelva la dirección del nodo que está en la posición intermedia.
- 18.3.** Se tiene una lista de simple enlace, el campo dato es un registro(estructura) con los datos de un alumno: nombre, edad, sexo. Escribir una función para transformar la lista de tal forma que si el primer nodo es de un alumno de sexo masculino el siguiente sea de sexo femenino.
- 18.4.** Una lista circular de cadenas está ordenada alfabéticamente. El puntero `Lc` tiene la dirección del nodo alfabéticamente mayor, apunta al nodo alfabéticamente menor.

Escribir una función para añadir una nueva palabra, en el orden que le corresponda, a la lista.

- 18.5.** Dada la lista del ejercicio anterior escribir una función que elimine una palabra dada.
- 18.6.** Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.
- 18.7.** Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y el exponente.

Escribir un programa que permita dar entrada a polinomios en x , representándolos con una lista enlazada simple. A continuación obtener una tabla de valores del polinomio para valores de $x = 0.0, 0.5, 1.0, 1.5, \dots, 5.0$

- 18.8.** Teniendo en cuenta la representación de un polinomio propuesta en el problema anterior hacer los cambios necesarios para que la lista enlazada sea circular. El puntero de acceso debe de tener la dirección del último término del polinomio, el cuál apuntará al primer término.
- 18.9.** Según la representación de un polinomio propuesta en el problema 18.7, escribir un programa para realizar las siguientes operaciones:

Obtener la lista circular suma de dos polinomios.
Obtener el polinomio derivada.
Obtener una lista circular que sea el producto de dos polinomios.

- 18.19.** Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter. Una vez que se tiene la lista ordenarla alfabéticamente y escribirla por pantalla.
- 18.11.** Escribir un programa en el que dados dos archivos F1, F2 formados por palabras separadas por un blanco o fin de línea, se creen dos conjuntos con las palabras de F1 y F2 respectivamente. Posteriormente encontrar las palabras comunes y mostrarlas por pantalla.
- 18.12.** Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final), eliminar un pasajero de la lista. A la lista se accede por un puntero al primer nodo y otro al último nodo.
- 18.13.** Para representar un entero largo, de más de 30 dígitos, utilizar una lista circular teniendo el campo dato de cada nodo un dígito del entero largo. Escribir un programa en el que se introduzcan dos enteros largos y se obtenga su suma.
- 18.14.** Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.
- 18.15.** Escriba una función que tenga como argumento una lista circular de números enteros. La función debe devolver el dato del nodo con mayor valor.

Pilas y colas

En este capítulo se estudian en detalle las estructuras de datos pila y cola que son probablemente las más frecuentemente utilizadas en los programas ordinarios. Son estructuras de datos que almacenan y recuperan sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** (*Last-in, first-out*, último en entrar- primero en salir) y las colas como estructuras **FIFO** (*First-in, First-out*, primero en entrar- primero en salir). Entre las numerosas aplicaciones de las pilas destaca la evaluación de expresiones algebraicas, así como la organización de la memoria. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

19.1 Concepto de pila

Una **pila** (*stack*) es una estructura de datos que cumple la condición: “los elementos se añaden o quitan (borran) de la misma sólo por su parte superior (**cima**) de la pila”. Debido a su propiedad específica “*último en entrar, primero en salir*” se le conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*). Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **Insertar** (*push*) añade un elemento en la cima de la pila y la operación **Quitar** (*pop*) elimina o saca un elemento de la pila.

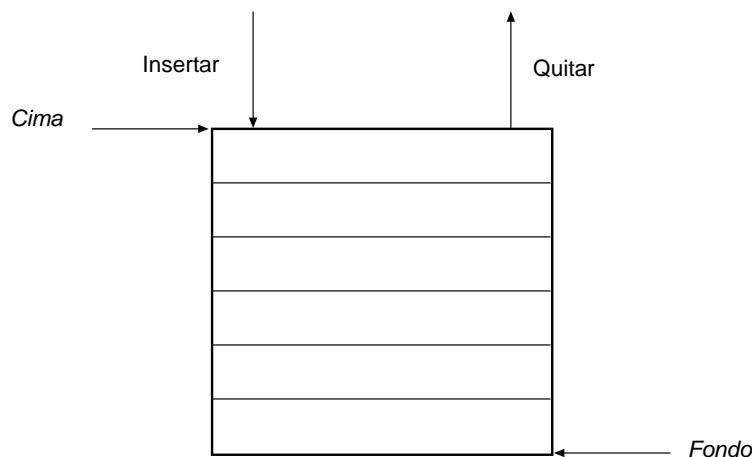


Figura 19.1 Operaciones básicas de una pila.

La pila se puede implementar mediante arrays en cuyo caso su dimensión o longitud es fija, y mediante punteros o listas enlazadas en cuyo caso se utiliza memoria dinámica y no existe limitación en su tamaño. Una pila puede estar *vacía* (no tiene elementos) o *llena* (en el caso de tener tamaño fijo, si no caben más elementos en la pila).

ESPECIFICACIÓN DE UNA PILA

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes:

<i>Tipo de dato</i>	Dato que se almacena en la pila.
<i>AnadeP (push)</i>	Insertar un dato en la pila.
<i>BorrarP (pop)</i>	Sacar (quitar) un dato de la pila.
<i>EsVacíaP</i>	Comprobar si la pila no tiene elementos.
<i>EstallenaP</i>	Comprobar si la pila está llena de elementos.
<i>PrimeroP</i>	Extrae el primer elemento de la pila sin borrarlo.

EL TIPO PILA IMPLEMENTADO CON ARRAYS

En C para definir una pila con arrays se utiliza una estructura. Los miembros de la estructura pila incluyen una lista (array) y un índice o puntero a la cima de la pila; además una constante con el máximo número de elementos limita la longitud de la pila. El método usual de introducir elementos en una pila es definir el *fondo* de la pila en la posición 0 del array, es decir, definir una *pila vacía* cuando su cima vale -1 (el *puntero de la pila* almacena el índice del array que se está utilizando como cima de la pila). La cima de la pila se va incrementando en uno cada vez que se añade un nuevo elemento, y se va decrementando en uno cada vez que se borra un elemento. Los algoritmos de introducir “insertar” (*push*) y quitar “sacar” (*pop*) datos de la pila utilizan el índice del array como puntero de la pila son:

Insertar (*push*). Verificar si la pila no está llena. Incrementar en uno el puntero de la pila. Almacenar el elemento en la posición del puntero de la pila.

Quitar (*pop*). Verificar si la pila no está vacía. Leer el elemento de la posición del puntero de la pila. Decrementar en uno el puntero de la pila.

En el caso de que el array que define la pila tenga *TamañoPila* elementos, el índice o puntero de la pila, estarán comprendidas en el rango 0 a *TamañoPila*-1 elementos, de modo que *en una pila llena* el puntero de la pila apunta a *TamañoPila*-1 y *en una pila vacía* el puntero de la pila apunta a -1 , ya que 0, teóricamente, será el índice del primer elemento.

EL TIPO PILA IMPLEMENTADO CON PUNTEROS

Para implementar una pila con punteros basta con usar una lista simplemente enlazada, con ello la pila estará vacía si la lista apunta a NULL. La pila teóricamente nunca estará llena. Los algoritmos de introducir “insertar” (*push*) y quitar “sacar” (*pop*) datos de la pila son:

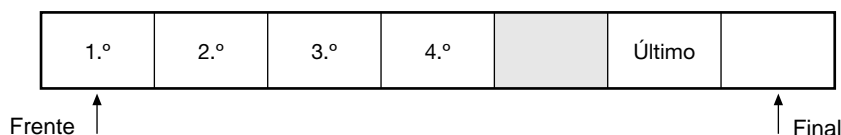
Insertar (*push*). Basta con añadir un nuevo nodo con el dato que se quiera insertar como primer elemento de la lista (pila).

Quitar (*pop*). Verificar si la lista (pila) no está vacía. Extraer el valor del primer nodo de la lista (pila). Borrar el primer nodo de la lista (pila).

19.2 Concepto de cola

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan, y por consiguiente, una cola es una estructura de tipo **FIFO** (*first-in/first-out*, *primero en entrar/primero en salir* o bien *primero en llegar/primero en ser servido*).

Las acciones que están permitidas en una cola son: Creación de una cola vacía; Verificación de que una cola está vacía; Añadir un dato al final de una cola; Eliminación de un dato de la cabeza de la cola.



EL TIPO COLA IMPLEMENTADO CON ARRAYS

La definición de una Cola ha de contener un *array* para almacenar los elementos de la cola, y dos marcadores o punteros (variables) que mantienen las posiciones *frente* y *final* de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador *final* apunta a una posición válida, entonces se añade el elemento a la cola y se incrementa el marcador *final* en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) *frente* y éste se incrementa en 1. Este procedimiento funciona bien hasta la primera vez que el puntero de *frente* alcanza el extremo del array quedando o bien vacío o bien lleno.

DEFINICIÓN DE LA ESPECIFICACIÓN DE UNA COLA

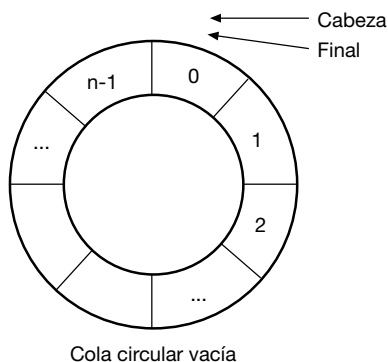
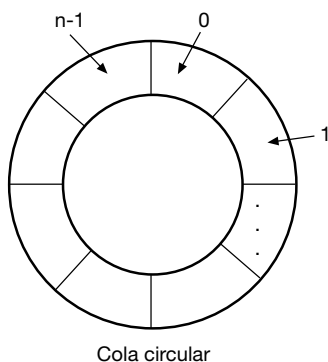
Se define en primer lugar el tipo genérico *TipoDato*. El **TDA Cola** contiene una lista cuyo máximo tamaño se determina por la constante *MaxTamC*. Se definen dos tipos de variables puntero o marcadores, *frente* y *final*. Estos son los punteros de cabecera y cola o final respectivamente.

<i>AnadeC</i>	Añade un elemento a la cola.
<i>BorrarC</i>	Borra el primer elemento de la cola.
<i>VaciaC</i>	Deja la cola sin ningún elemento
<i>EsvaciaC</i>	Decide si una cola está vacía.
<i>EstallenaC</i>	Decide si una cola está llena.
<i>PrimeroC</i>	Extrae el primer elemento de la cola.

Cuando un elemento se añade a la cola, se hace una prueba para comprobar si el marcador *final* apunta a una posición válida, a continuación se añade el elemento a la cola y el marcador *final* se incrementa en uno. Cuando se elimina un elemento de la cola, se realiza una prueba para comprobar si la cola está vacía, y si no es así, se recupera el elemento que se encuentra en la posición apuntada por el marcador de *frente* y el marcador de *frente* se incrementa en uno. Este procedimiento funciona bien hasta que el marcador *final* alcanza el tamaño máximo del *array*. Si durante este tiempo se han producido eliminaciones, habrá espacio vacío al principio del array. Sin embargo, puesto que el marcador *final* apunta al extremo del *array*, implicará que la cola está llena y ningún dato más se añadirá.

Existen diversas soluciones a este problema:

Retroceso	Consiste en mantener fijo a uno el valor de <i>frente</i> , realizando un desplazamiento de una posición para todas las componentes ocupadas cada vez que se efectúa una supresión.
Reestructuración	Cuando <i>final</i> llega al máximo de elementos se desplazan las componentes ocupadas hacia atrás las posiciones necesarias para que el principio coincida con el primera posición del array.
Mediante un array circular	Un <i>array</i> circular es aquel en el cual se considera que la primera posición sigue a la última.



La variable *frente* es siempre la posición del elemento que precede al primero de la cola y se avanza en el sentido de las agujas del reloj. La variable *final* es la posición en donde se hizo la última inserción. Después que se ha producido una inserción, *final* se mueve circularmente a la derecha. La implementación del movimiento circular “calcular siguiente” se realiza utilizando la *teoría de los restos*:

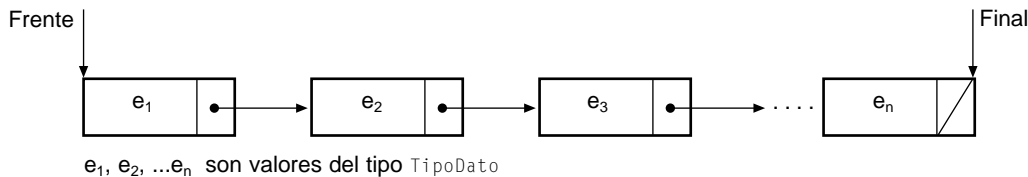
Mover Final adelante =	$(Final+1)\%MaxTamC$	Siguiente de Final
Mover Frente adelante =	$(Frente+1) \% MaxTamC$	Siguiente de Frente

Para implementar los algoritmos que formalizan la gestión de colas en un array circular hay que tener en cuenta que *Frente* apunta siempre a una posición anterior donde se encuentra el primer elemento de la cola y *Final* apunta siempre a la posición donde se encuentra el último de la cola. Por lo tanto la parte esencial de las tareas de gestión de una cola son:

- Creación de una cola vacía: hacer $Frente = Final = 0$.
- Comprobar si una cola está vacía: ¿es $Frente == Final$?
- Comprobar si una cola está llena: ¿es $(Final+1)\%MaxTamC == Frente$? No se confunda con cola vacía.
- Añadir un elemento a la cola: si la cola no está llena, añadir un elemento en la posición siguiente a *Final* y se establece: $Final = (Final+1)\%MaxTamC$.
- Eliminación de un elemento de una cola: si la cola no está vacía, eliminarlo de la posición siguiente a *Frente* y establecer $Frente = (Frente+1) \% MaxTamC$.

REALIZACIÓN DE UNA COLA CON UNA LISTA ENLAZADA

La implementación de una cola con una lista enlazada utiliza dos punteros para acceder a la lista. El puntero *Frente* y el puntero *Final*. El puntero *Frente* referencia al primer elemento de la cola. El puntero *Final* referencia al último elemento en ser añadido, el último que será retirado. Con esta representación no tiene sentido la operación que prueba si la cola está llena



Cola con lista enlazada

PROBLEMAS RESUELTOS

19.1 Escriba las primitivas de gestión de una pila implementada con un array.

Análisis del problema

Se define en primer lugar una constante *MaxTamaPila* de valor 100 valor máximo de los elementos que podrá contener la pila. Se define la pila como una estructura cuyos campos (miembros) serán el puntero *cima* que apuntará siempre al último elemento añadido a la pila y un array *A* cuyos índices variarán entre 0 y *MaxTamaPila*-1. Posteriormente se implementan las las primitivas

- *VaciaP*. Crea la pila vacía poniendo la *cima* en el valor -1.
- *EsvaciaP*. Decide si la pila está vacía. En este caso ocurrirá cuando su *cima* valga -1.
- *EstallenaP*. Si bien no es una primitiva básica de gestión de una pila; la implementación se realiza con un array conviene disponer de ella para prevenir posibles errores. En este caso la pila estará llena cuando la *cima* apunte al valor *MaxTamaPila*-1.
- *AnadeP*. Añade un elemento a la pila. Para hacerlo comprueba en primer lugar que la pila no esté llena, y en caso afirmativo, incrementa la *cima* en una unidad, para posteriormente poner en el array *A* en la posición *cima* el elemento.
- *PrimeroP*. Comprueba que la pila no esté vacía, y en caso de que así sea, dará el elemento del array *A* almacenado en la posición apuntada por la *cima*.
- *BorrarP*. Se encarga de eliminar el último elemento que entró en la pila. En primer lugar comprueba que la pila no esté vacía en cuyo caso, disminuye la *cima* en una unidad.
- *Pop*. Esta operación extrae el primer elemento de la pila y lo borra. Puede ser implementada directamente, o bien llamando a las primitivas *PrimeroP* y posteriormente a *BorrarP*.
- *Push*. Esta primitiva coincide con *AnadeP*.

Codificación

```
typedef int TipoDato;

/* archivo pilaarray.h */

#include <stdio.h>
#include <stdlib.h>
#define MaxTamaPila 100

typedef struct
{
    TipoDato A[MaxTamaPila];
    int cima;
}Pila;

void VaciaP(Pila* P);
void AnadeP(Pila* P,TipoDato elemento);
void BorrarP(Pila* P);
TipoDato PrimeroP(Pila P);
int EsVaciaP(Pila P);
int EstallenaP(Pila P);
void Pop(Pila* P, TipoDato elemento);
TipoDato Push(Pila *P);

void VaciaP(Pila* P)
{
    P -> cima = -1;
}

void AnadeP(Pila* P,TipoDato elemento)
{
    if (EstallenaP(*P))
    {
        puts("Desbordamiento pila");
        exit (1);
    }
    P->cima++;
    P->A[P->cima] = elemento;
}

void Pop(Pila* P,TipoDato elemento)
{
    AnadeP(P, elemento);
}

TipoDato Push(Pila *P)
{
    TipoDato Aux;
    if (EsVaciaP(*P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
}
```

```

    }
    Aux = P->A[P->cima];
    P->cima--;
    return Aux;
}

TipoDato PrimeroP(Pila P)
{
    TipoDato Aux;
    if (EsVaciaP(P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    Aux = P.A[P.cima];
    return Aux;
}

void BorrarP(Pila* P)
{
    if (EsVaciaP(*P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    P->cima --;
}

int EsVaciaP(Pila P)
{
    return P.cima == -1;
}

int EstallenaP(Pila P)
{
    return P.cima == MaxTamaPila-1;
}

```

- 19.2.** *Escribir un programa que usando las primitivas de gestión de una pila, lea datos de la entrada (-1 fin de datos) los almacene en una pila y posteriormente visualice dicha la pila.*

Análisis del problema

Si se supone que el archivo `pilaarray.cpp` contiene todas las primitivas de gestión de una pila, para resolver el problema bastará con declarar `TipoDato` como un entero incluir el archivo `pilaarray.cpp` anterior, y mediante un programa principal en un primer bucle `while` se leen los datos y se almacenan en una pila, para posteriormente en otro bucle extraer datos de la pila y presentarlos en pantalla.

Codificación

```

typedef char TipoDato
#include <pilaarray.cpp>
void main()

```

```

{
    Pila P;
    int x;

    VacíaP(&P);
    do
    {
        printf("dame dato -1=fin \n");
        scanf("%d",&x);
        if (x != -1)
            AnadeP(&P, x);
        while (x != -1);
    }
    printf("escritura de la pila\n");
    while(!EsVacíaP(P))
    {
        printf("%d \n",PrimeroP(P));
        BorrarP( &P );
    }
}

```

19.3. *Escriba las primitivas de gestión de una pila implementada con una lista simplemente enlazada.*

Análisis del problema

Se define la pila como una lista simplemente enlazada. Posteriormente se implementan las primitivas:

- **VacíaP.** Crea la pila vacía poniendo la pila P a NULL.
- **EsvacíaP.** Decide si pila vacía. Esto ocurrirá cuando P valga NULL.
- **AnadeP.** Añade un elemento a la pila. Para hacerlo, lo único que se debe hacer, es añadir un nuevo nodo que contenga como información el elemento que se quiera añadir y ponerlo como primero de la lista enlazada.
- **PrimeroP.** En primer lugar se comprobará que la pila (lista) no esté vacía, y en caso de que así sea dará el campo el almacenado en el primer nodo de la lista enlazada.
- **BorrarP.** Se encarga de eliminar el último elemento que entró en la pila. En primer lugar se comprueba que la pila no esté vacía en cuyo caso, se borra el primer nodo de la pila (lista enlazada).
- **Pop.** Esta operación extrae el primer elemento de la pila y lo borra. Puede ser implementada directamente, o bien llamando a las primitivas **PrimeroP** y posteriormente a **BorrarP**.
- **Push.** Esta primitiva coincide con **AnadeP**.
- **NuevoNodo.** Es una función auxiliar de la implementación que se encarga de reservar memoria para la operación **AnadeP**.
- **EstallenaP.** En esta implementación no tiene ningún sentido, ya que se supone que la memoria dinámica es en principio inagotable.

Codificación

```

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

typedef int TipoDato;
typedef struct unnodo
{
    TipoDato el;
    struct unnodo *sig;
}Nodo;

```



```

typedef Nodo Pila;

Nodo* NuevoNodo(TipoDato elemento);

void VaciaP(Pila** P);
void AnadeP(Pila** P, TipoDato elemento);
void BorrarP(Pila** P);
TipoDato PrimeroP(Pila *P);
int EsVacíaP(Pila *P);
void Pop(Pila** P, TipoDato elemento);
TipoDato Push(Pila **P);

Nodo* NuevoNodo(TipoDato elemento)
{
    Nodo *a ;
    a = (Nodo*)malloc(sizeof(Nodo));
    a -> el = elemento;
    a -> sig = NULL;
    return a;
}

void VaciaP(Pila** P)
{
    *P = NULL;
}

void AnadeP(Pila** P, TipoDato elemento)
{
    Nodo * nn;
    nn = NuevoNodo(elemento);
    nn->sig = (*P);
    *P = nn;
}

void Pop(Pila** P, TipoDato elemento)
{
    AnadeP(P, elemento);
}

TipoDato Push(Pila **P)
{
    TipoDato Aux;
    Pila *nn;

    if (EsVacíaP(*P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    Aux = (*P)->el;
    nn = *P;
    *P = nn->sig;
    free(nn);
}

```

```

    return Aux;
}

TipoDato PrimeroP(Pila *P)
{
    TipoDato Aux;

    if (EsVaciaP(P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    Aux = P->el;
    return Aux;
}

void BorrarP(Pila** P)
{
    Pila *nn;

    if (EsVaciaP(*P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    nn =(*P);
    (*P)= nn->sig;
    free(nn);
}

int EsVaciaP(Pila *P)
{
    return P == NULL;
}

```

19.4. Usando las primitivas de gestión de una pila de enteros escriba las siguientes funciones: *EscribePila* que recibe como parámetro una pila y la escribe, *CopiaPila* que copia una pila en otra. *DaVueltaPila* que da la vuelta a una pila.

Análisis del problema

Usando el archivo `pilalista.cpp` en el que se tiene ya la implementación de las primitivas de una pila, lo único que se debe hacer es implementar las siguientes funciones:

- *EscribePila* que recibe como parámetro por valor una pila y mediante un bucle `while`, se van extrayendo, borrando, y escribiendo los elementos de la pila.
- *CopiaPila* que recibe como parámetro por valor una pila `p` y devuelve en `Pcop` una copia exacta de la pila `P`. Para ello basta con volcar la pila `P` en una pila `Paux` auxiliar, para posteriormente volcar la pila `Paux` en la pila `Pcop`.
- *DaVueltaPila* que recibe como parámetro por valor la pila `P` y vuelca su contenido en la pila `Pcop`.

Codificación

```

void CopiaPila (Pila *P, Pila**Pcop)
{
    Pila *Paux;

```

```

    TipoDato e;

    VaciaP(&Paux);
    while (! EsVaciaP(P))
    {
        e = PrimeroP(P);
        BorrarP(&P);
        AnadeP(&Paux,e);
    }
    VaciaP(Pcop);
    while (! EsVaciaP(Paux))
    {
        e = PrimeroP(Paux);
        BorrarP(&Paux);
        AnadeP(Pcop,e);
    }
}

void DaVueltaPila (Pila *P, Pila**Pcop)
{
    TipoDato e;

    VaciaP(Pcop);
    while (!EsVaciaP(P))
    {
        e = PrimeroP(P);
        BorrarP(&P);
        AnadeP(Pcop,e);
    }
}

void EscribePila(Pila *P)
{
    TipoDato e;

    while (! EsVaciaP(P))
    {
        e = PrimeroP(P);
        BorrarP(&P);
        printf("%d\n", e);
    }
}

```

19.5. *Escriba las funciones MayorPila, MenorPila, MediaPila que calculan el elemento mayor menor y la media de una pila de enteros.*

Análisis del problema

Al igual que en el ejercicio anterior se usa el archivo pilalista.cpp en el que se tiene ya la implementación de las primitivas de una pila, lo único que resta es implementar las siguientes funciones:

- **MayorPila**, calcula el elemento mayor, inicializando la variable **Mayor** a un número muy pequeño, y mediante un bucle voraz controlado por ser vacía la pila se extraen los elementos de la pila reteniendo el mayor de todos.
- **MenorPila**, calcula el elemento menor. Se realiza de manera análoga a la función **MayorPila**.

- `MediaPila` que calcula la media de una pila, para lo cual basta acumular los datos que contiene la pila en un acumulador `Total` y con contador `k` contar los elementos que hay, para devolver el cociente real.

Codificación (Se encuentra en la página web del libro)

- 19.6.** Escriba las funciones `LiberarPila` y `SonIgualesPilas` que respectivamente libera todos los nodos de una pila implementada con listas y decide si dos pilas son iguales.

Análisis del problema

Al igual que en el ejercicio anterior se usa el archivo `pilalista.cpp` en el que se tiene ya la implementación de las primitivas de una pila, lo único que resta por hacer es implementar las siguientes funciones:

- `LiberarPila` que mediante un bucle mientras se encarga de ir extrayendo los elementos de la pila y mediante la función `BorrarP` irlos eliminando.
- `SonIgualesPilas`. Dos pilas son iguales si tienen el mismo número de elementos y además coinciden en el orden de colocación. Por lo tanto basta con un bucle mientras, controlado por haber datos en las dos pilas y haber sido todos los elementos extraídos anteriormente iguales, extraer un elemento de cada una de las pilas y seguir decidiendo sobre su igualdad. Al final del bucle debe ocurrir que las dos pilas estén vacías y además que la variable lógica que controla el bucle sea verdadera.

Codificación

```
void LiberarPila(Pila**P)
{
    while (!EsVaciaP(*P))
        BorrarP(P);
}

int SonIgualesPilas(Pila *P, Pila* P1)
{
    int sw = 1;
    TipoDato e,e1;

    while (! EsVaciaP(P) && !EsVaciaP(P1) &&  sw)
    {
        e = PrimeroP(P);
        BorrarP(&P);
        e1 = PrimeroP(P1);
        BorrarP(&P1);
        sw = (e == e1);
    }
    return (sw && EsVaciaP(P)&& EsVaciaP(P1));
}
```

- 19.7.** Escriba un programa que lea una frase y decida si es palíndroma. Una frase es palíndroma si se puede leer igual de izquierda a derecha y de derecha a izquierda. Ejemplo para no es palíndroma, pero alila si que lo es.

Análisis del problema

Para resolver el problema usaremos una función que nos lea una frase carácter a carácter poniéndola en una pila, cuando se haya terminado se da la vuelta la pila en otra pila `Pcop`. La frase será palíndroma si las dos pilas son iguales. En la codificación que se presenta se implementan además las funciones `DaVueltaPila` y `SonIgualesPilas`.

Codificación

```

void DaVueltaPila (Pila *P,Pila**Pcop)
{
    TipoDato e;
    VacíaP( Pcop );
    while ( ! EsVacíaP( P ))
    {
        e = PrimeroP( P );
        BorrarP( &P );
        AnadeP(Pcop, e);
    }
}

int SonIgualesPilas(Pila *P, Pila* P1)
{
    int sw = 1;

    TipoDato e, e1;
    while ( ! EsVacíaP(P) && !EsVacíaP(P1) && sw)
    {
        e = PrimeroP( P );
        BorrarP( &P );
        e1 = PrimeroP( P1 );
        BorrarP( &P1 );
        sw=(e == e1);
    }
    return (sw && EsVacíaP(P)&& EsVacíaP(P1));
}

int palindroma()
{
    Pila *P, *Pcop;
    char ch;
    puts(“ frase a comprobar que es palindroma”);
    VacíaP( &P );
    for( ;(ch = getchar()) != ‘\n’; )
        AnadeP(&P, ch);
    DaVueltaPila(P, &Pcop);
    return (SonIgualesPilas(P,Pcop));
}

```

19.8. ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es int:?

```

Pila *P;
int x=4, y;
VacíaP(&P);
AnadeP(&P,x);
printf(“\n%d “,PrimeroP(P));
BorrarP(&P);
AnadeP(&P,32);
y=PrimeroP(P);
BorrarP(P);

```

```

AnadeP(&P,y);

do
{
    printf("\n%d",PrimeroP(P));
    BorrarP(P);
}
while (!EsVaciaP(P));

```

Solución

Se vacía la pila y posteriormente se añade el dato 4. Se escribe el primero de la pila que es 4. Se borra el primer elemento de la pila con lo que se vuelve a quedar vacía. Se añade el número 32 a la pila, para después borrarlo, y luego añadirlo. El último bucle extrae el número 32 de la pila, lo escribe y después de borrarlo la pila se queda vacía con lo que se sale del bucle. Es decir la solución es:

```

4
32

```

- 19.9.** Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma: $X&Y$. Donde X es una cadena de caracteres e Y es la cadena inversa. El carácter $&$ es el separador y siempre se supone que existe.

Análisis del problema

Se usan tres pilas. En la primera se introducen todos los caracteres que estén antes que el carácter $&$. En la segunda se introducen todos los caracteres que estén después de $&$. Seguidamente se da la vuelta a la primera pila para dejar los caracteres en el mismo orden en el que se leyeron. Por último se devuelve el valor “son iguales las dos pilas”. Las funciones `DaVueltaPila` y `SonIgualesPilas` son las mismas del problema 19.7.

Codificación

```

int extrana()
{
    Pila *P, *P1,*Pcop;
    char ch;

    puts(" frase a comprobar que es extraña");
    VacíaP(&P);
    for( ;(ch = getchar()) != '&; )
        AnadeP(&P, ch);
    VacíaP(&P1);
    for( ;(ch = getchar()) != '\n'; )
        AnadeP(&P1, ch);
    DaVueltaPila(P, &Pcop);
    return (SonIgualesPilas(P1, Pcop));
}

```

- 19.10.** Escribir una función que haciendo uso del tipo `Pila` de caracteres, procese cada uno de los caracteres de una expresión que viene dada en una línea de caracteres. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

$$((a+b)*5) - 7$$

a esta otra expresión le falta un corchete:

$$2*[(a+b)/2.5 + x - 7*y$$

Análisis del problema

Para comprobar el equilibrio de paréntesis y corchetes, es preciso comprobar que tienen el mismo número de abiertos que cerrados y que además aparecen en el orden correspondiente. Se usa una pila por la que pasarán sólo los paréntesis abiertos y los corchetes abiertos en el orden en que aparecen. Cada vez que aparezca un paréntesis cerrado o un corchete cerrado, se extrae un elemento de la pila, comprobando su igualdad con el último que se ha leído, almacenado en una variable lógica `sw` el valor verdadero o falso dependiendo de que se satisfaga la igualdad. Por lo tanto si con un bucle `for` controlado por el fin de línea y por el valor verdadero de una variable `sw` de tipo lógico (previamente inicializada a verdadero), se leen los caracteres y realiza lo indicado cuando termine el bucle puede ser que `sw` sea falso, en cuyo caso la expresión no es correcta, o que se sea verdadero, en cuyo caso la expresión será correcta si la pila está vacía.

Codificación

```
int equilibrio()
{
    Pila *P;
    char ch,e,sw=1 ;

    puts(" frase a comprobar equilibrio de parentesis");
    VacíaP(&P);
    for( ;(ch = getchar()) != '\n' && sw; )
        if ((ch == '(') || (ch == '['))
            AnadeP(&P, ch);
        else
            if ((ch == ')') || (ch == ']'))
                if( ! EsVacíaP(P))
                {
                    e = PrimeroP( P );
                    BorrarP( &P );
                    sw = e==ch;
                }
            else
                sw = 0;
    if ( sw )
        sw = EsVacíaP(P);
    return (sw);
}
```

19.11. Escribir las declaraciones necesarias y las primitivas para gestionar una cola mediante un array circular.

Análisis del problema

Las declaraciones necesarias son una estructura que contiene dos punteros `frente` y `final` y un *array* que puede almacenar `MaxTamC` datos. `MaxTamC` es una constante previamente definida. `Final` apuntará siempre al último elemento que se añadió la cola, y `Frente` siempre a una posición antes de donde se encuentra el primer elemento, entendiendo que la posición anterior a la 0 es `MaxTamC-1`, y para el resto una menos. Análogamente la posición siguiente de `MaxTamC-1` es la 0, y para el resto es una unidad más. De esta manera la cola estará vacía cuando `Frente` y `Final` apuntan a la misma posición, y la cola estará llena cuando al calcular el siguiente valor de `Final` se tiene el `Frente`, ya que si se permitiera avanzar a `Final` se confundiría cola vacía con cola llena. Para calcular el siguiente valor tanto de `Frente` como de `Final` basta con hacer `Frente = (Frente+1)%MaxTamC`; igual método con `Final`. Las primitivas de gestión de la cola son:

- `VacíaC`. Crea una cola vacía, para lo cual basta con poner el `Frente` y el `Final` en la posición 0.
- `EsVacíaC`. Decide si una cola está vacía. Es decir, si `Frente == Final`.

- **EstallenaC.** Decide si la cola está llena. Es decir, si $(Final+1)\%MaxTamC == Frente$.
- **PrimeroC.** Extrae el primer elemento de la cola que se encuentra en $(Frente+1)MaxTamC$. Previamente a esta operación ha de comprobarse que la cola no esté vacía.
- **AnadeC.** Añade un elemento a la cola. Este elemento se añade a la posición del array $(Fina+1)\%MaxTamC$. Final también debe ponerse en esa posición. Previamente a esta operación ha de comprobarse si la cola está llena.
- **BorrarC.** Elimina el primer elemento de la cola. Para ello basta con hacer $Frente = (Frente+1)\%MaxTamC$. Previamente a esta operación ha de comprobarse que la cola no está vacía.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#define MaxTamC 100
typedef int TipoDato;
typedef struct
{
    int frente, final;
    TipoDato A[MaxTamC];
} Cola;

void VacíaC(Cola* C);
void AnadeC(Cola* C, TipoDato e);
void BorrarC(Cola* C);
TipoDato PrimeroC(Cola C);
int EsVacíaC(Cola C);
int EstallenaC(Cola C);

void VacíaC(Cola* C)
{
    C->frente = 0;
    C->final = 0;
}

void AnadeC(Cola* C, TipoDato e)
{
    if (EstallenaC( *C ))
    {
        puts("desbordamiento cola");
        exit (1);
    }
    C->final = (C->final + 1) % MaxTamC;
    C->A[C->final] = e;
}

TipoDato PrimeroC(Cola C)
{
    if (EsVacíaC(C))
    {
        puts("Elemento frente de una cola vacía");
        exit (1);
    }
    return (C.A[(C.frente+1) % MaxTamC]);
}
```



```

int EsVaciaC(Cola C)
{
    return (C.frente == C.final);
}

int EstallenaC(Cola C)
{
    return (C.frente == (C.final+1) % MaxTamC);
}

void BorrarC(Cola* C)
{
    if (EsVaciaC(*C))
    {
        puts("Eliminación de una cola vacía");
        exit (1);
    }
    C->frente = (C->frente + 1) % MaxTamC;
}

```

19.12. Usando las primitivas de gestión de colas escriba funciones para generar aleatoriamente una cola, escribir una cola, calcular el número de elementos de una cola, y eliminar de una cola todos los elementos mayores que un elemento que se pasa como parámetro.

Análisis del problema

Las funciones que se codifican son las siguientes:

GeneraColaAleatoriamente. Recibe como parámetro una cola. Declara una constante `Max` y añade a la cola números aleatorios comprendidos entre 0 y `Max-1`. El final de la entrada de datos viene dado por haber generado el número aleatorio 0. **EscribeCola.** Recibe como parámetro una cola y la presenta en pantalla. Para ello mediante un bucle `while` extrae elementos de la cola y los escribe.

NumeroDeElementosCola. Informa de elementos tiene una cola. Para resolver el problema, usa un contador que se inicializa a cero, y se incrementa en una unidad, cada vez que un bucle `while` extrae un elemento de la cola.

EliminaMayores. Recibe como parámetro una cola y un elemento `e`. Mediante un bucle `while` pone en una cola `C1` todos los elementos de la cola que se recibe como parámetro que cumplen la condición de ser menores o iguales que el elemento `e` que se recibe como parámetro.

Codificación (Se encuentra en la página web del libro)

19.13. Escriba las declaraciones necesarias y las primitivas de gestión de una cola implementada con listas enlazadas.

Análisis del problema

Se declaran en primer lugar todos los tipos de datos necesarios para una lista enlazada. Una cola será una estructura con dos punteros a la lista frente que apuntará al primer elemento de la cola y final que apuntará al último elemento.

- **VaciaC.** Crea una cola vacía, para lo cual basta con poner el `Frente` y el `Final` a `NULL`.
- **EsVaciaC.** Decide si una cola está vacía. Es decir si `Frente` y `Final` valen `NULL`.
- **EstallenaC.** Esta función no es ahora necesaria ya que teóricamente no hay límites.
- **PrimeroC.** Extrae el primer elemento de la cola que se encuentra en el nodo `Frente`. Previamente a esta operación ha de comprobarse que la cola no esté vacía.
- **AnadeC.** Añade un elemento a la cola. Este elemento se añade en un nuevo nodo que será el siguiente de `Final` en el caso de que la cola no esté vacía. Si la cola está vacía el `Frente` debe apuntar a este nuevo nodo. En todo caso el `final` siempre debe moverse al nuevo nodo.

- **BorrarC.** Elimina el primer elemento de la cola. Para hacer esta operación la cola no debe estar vacía. El borrado se realiza avanzando Frente al nodo siguiente, y liberando la memoria correspondiente.
- **EliminarC.** Esta primitiva libera toda la memoria que tenga una cola ya creada. Se realiza mediante un bucle controlado por el Final de lista, liberando la memoria ocupada por cada nodo en cada una de las iteraciones del bucle.

```
#include <stdio.h>
#include <stdlib.h>
typedef int TipoDato;
struct Nodo
{
    TipoDato el;
    struct Nodo* sig;
};

typedef struct
{
    Nodo * Frente;
    Nodo * Final;
}Cola;

void VaciaC(Cola* C);
void AnadeC(Cola* C,TipoDato el);
void EliminarC(Cola* C);
void BorrarC(Cola* C);
TipoDato PrimeroC(Cola C);
int EsVacíaC(Cola C);
Nodo* crearnodo(TipoDato el);

void VaciaC(Cola* C)
{
    C->Frente = NULL;
    C->Final = NULL;
}

Nodo* crearnodo(TipoDato el)
{
    Nodo* nn;

    nn = (Nodo*)malloc(sizeof(Nodo));
    nn->el = el;
    nn->sig = NULL;
    return nn;
}

int EsVacíaC(Cola C)
{
    return (C.Frente == NULL);
}

void AnadeC(Cola* C,TipoDato el)
{
    Nodo* a;
    a = crearnodo(el);
```

```

    if (EsVaciaC(*C))
        C->Frente = a;
    else
        C->Final->sig = a;
        C->Final = a;
}

void BorrarC(Cola* C)
{
    Nodo *a;

    if (!EsVaciaC(*C))
    {
        a = C->Frente;
        C->Frente = C->Frente->sig;
        if(C->Frente == NULL)
            C->Final == NULL;
        free(a);
    }
    else
    {
        puts("Error eliminación de una cola vacía");
        exit(-1);
    }
}

TipoDato PrimeroC(Cola C)
{
    if (EsVaciaC(C))
    {
        puts("Error: cola vacía");
        exit(-1);
    }
    return (C.Frente->el);
}

void EliminaC(Cola* C)
{
    for (; C->Frente;)
    {
        Nodo* n;

        n = C->Frente;
        C->Frente = C->Frente->sig;
        free(n);
    }
}

```

19.14. *Escribir una función que tenga como argumentos dos colas del mismo tipo. Devuelva cierto si las dos colas son idénticas*

Análisis del problema

Se usan para resolver el problema las primitivas de gestión de colas implementando una función `SonIgualesColas` que dará el valor verdadero cuando las dos colas tengan igual número de elementos y además estén colocadas en el mismo orden.

Codificación

```

int SonIgualescolas(Cola *C, Cola* C1)
{
    int sw=1;
    TipoDato e,e1;

    while (!EsVaciaC(C)&& !EsVaciaC(C1)&& sw)
    {
        e = PrimeroC(C);
        BorrarC(&C);
        e1 = PrimeroC(C1);
        BorrarP(&C1);
        sw =(e == e1);
    }
    return (sw && EsVaciaC(C)&& EsVaciaC(C1));
}

```

19.15. Considerar una cola de nombres representada por una array circular con 6 posiciones, el campo frente con el valor: Frente = 2, y los elementos de la Cola: Mar, Sella, Centurión. Escribir los elementos de la cola y los campos siguiente de Frente y Final según se realizan estas operaciones:

- Añadir Gloria y Generosa a la cola.
- Eliminar de la cola.
- Añadir Positivo.
- Añadir Horche a la cola.
- Eliminar todos los elementos de la cola.

Solución

- Tras la primera operación se escribirá Mar y Generosa, quedando la cola con Mar, Sella, Centurión Gloria, Generosa.
- Después de realizar la segunda operación se escribirá Sella y Generosa, quedando la cola con Sella, Centurión Gloria, Generosa.
- Después de añadir Positivo se escribirá Sella y Positivo y la cola contendrá los siguientes elementos Sella, Centurión Gloria, Generosa, Positivo.
- Al añadir Horche a la cola se producirá un error ya que la cola está llena interrumpiéndose la ejecución del programa.

19.16. Escriba una función que reciba como parámetro una cola de números enteros y nos devuelva el mayor y el menor de la cola.

Análisis del problema

Se usan las primitivas de gestión de colas implementadas con listas, lo único que hay que hacer es inicializar Mayor y menor al primer elemento de la cola, y mediante un bucle voraz controlado por si se vacía la cola, ir actualizando las variables mayor y menor.

Codificación

```

void Mayormenor(Cola *C, TipoDato * Mayor, TipoDato *menor)
{
    TipoDato M,m,e;

    M = -32767;
    m = 32367;
    while(!EsVaciaC(*C))

```

```

{
    e=PrimeroC(*C);
    BorrarC(C);
    if(M<e)
        M=e;
    if(m>e)
        m=e;
}
*Mayor=M;

*menor=m;
}

```

PROBLEMAS PROPUESTOS

19.1. Obtener una secuencia de 10 elementos reales, guardarlos en un array y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.

19.2. Una bicola es una estructura de datos lineal en la que la inserción y borrado se pueden hacer tanto por el extremo frente como por el extremo final. Suponer que se ha elegido una representación dinámica, con punteros, y que los extremos de la lista se denominan frente y final. Escribir la implementación de las operaciones:

```

InsertarFrente(), InsertarFinal(),
EliminarFrente() y EliminarFinal().

```

19.3. Considere una bicola de caracteres, representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y los elementos de la bicola:

```
frente = 5   final = 7   Bicola: A,C,E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos F y K por el final de la bicola.
- Añadir los elementos R, W y V por el frente de la bicola.
- Añadir el elemento M por el final de la bicola.
- Eliminar dos caracteres por el frente.
- Añadir los elementos K y L por el final de la bicola.
- Añadir el elemento S por el frente de la bicola.

19.4. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de

código para poner todos los elementos que son par de la pila en la cola.

19.5. Con un archivo de texto se quieren realizar las siguientes acciones: formara una lista de colas, de tal forma que cada nodo de la lista esté la dirección de una cola que tiene todas las palabras del archivo que empiezan por una misma letra. Visualizar las palabras del archivo, empezando por la cola que contiene las palabras que comienzan por *a*, a continuación las de la letra *b*, a sí sucesivamente.

19.6. Escribir un programa en el que se generen 100 números aleatorios en el rango -25 .. +25 y se guarden en una pila implementada mediante un array considerado circular. Una vez creada la cola, el usuario puede pedir que se forme otra cola con los números negativos que tiene la cola original.

19.7. Escribir un programa en el que se manejen un total de $n=5$ pilas: P_1 , P_2 , P_3 , P_4 y P_5 . La entrada de datos será pares de enteros (i, j) tal que $1 \leq \text{abs}(i) \leq n$. De tal forma que el criterio de selección de pila:

- Si i es positivo, debe de insertarse el elemento j en la pila P_i .
- Si i es negativo, debe de eliminarse el elemento j de la pila P_i .
- Si i es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso el programa debe de escribir el contenido de la n Pilas en pantalla.

19.8. Modificar el programa 19.7 para que la entrada sean triplas de números enteros (i, j, k) , donde i , j tienen el

mismo significado que en 19.8, y k es un número entero que puede tomar los valores -1 , 0 con este significado:

- -1 , hay que borrar todos los elementos de la pila.
- 0 , el proceso es el indicado en 19.8 con i y j .

19.9. Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en que un cliente paga en la caja, el carro de la compra que tiene queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

19.10. Se trata de crear una cola de mensajes que sirva como buzón para que los usuarios puedan depositar y recoger mensajes. Los mensajes pueden tener cualquier formato, pero deben contener el nombre de la persona a la que van dirigidos y el tamaño que ocupa el mensaje. Los usuarios pueden dejar sus mensajes en la cola y al recogerlos especificar su nombre por el que recibirán el primer mensaje que está a su nombre o una indicación de que no tienen ningún mensaje para ellos. Realizar el programa de forma que muestre una interfaz con las opciones indicadas y que antes de cerrarse guarde los mensajes de la cola en un fichero binario del que pueda recogerlos en la siguiente ejecución.

Árboles

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales* al contrario que los arrays y las listas enlazadas que constituyen *estructuras lineales*.

Los árboles son muy utilizados en informática para representar fórmulas algebraicas como un método eficiente para búsquedas grandes y complejas, aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, procesadores de texto y algoritmos de búsqueda.

En el capítulo se estudiará el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda. Asimismo se estudiarán algunas aplicaciones típicas del diseño y construcción de árboles

20.1 Árboles generales

Un árbol es un tipo estructurado de datos que representa una estructura jerárquica entre sus elementos. La definición de un árbol viene dada recursivamente de la siguiente forma: un árbol o es vacío o se considera formado por un nodo raíz y un conjunto disjunto de árboles llamados subárboles del raíz. Es posible representar gráficamente un árbol de diversas formas:

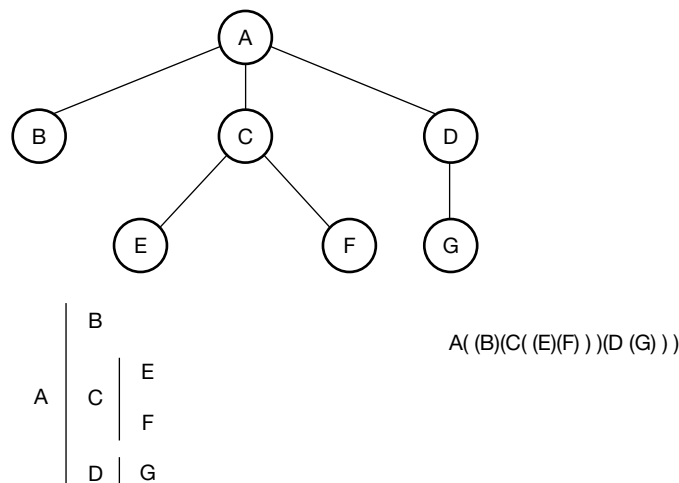


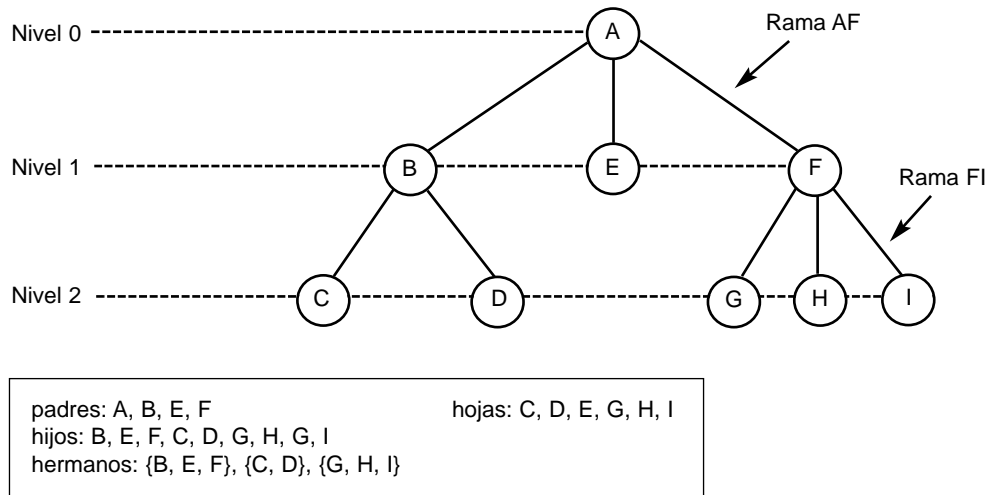
Figura 20.1 Representación de árboles.

TERMINOLOGÍA

Las siguientes definiciones forman parte de la terminología específica:

- *Nodos* son los elementos o vértices del árbol.
- Cada nodo excepto la raíz tiene un único *antecesor* o *ascendiente* denominado *padre*.
- *Hijo* es un nodo descendiente inmediato de otro nodo de un árbol.
- Se llama *grado de un nodo* al número de sus hijos.
- *Nodo hoja* es un nodo de grado 0.
- *Hermanos* son los nodos hijos del mismo padre.
- Cada nodo de un árbol tiene asociado un número entero *nivel* que se determina por el número de antecesores que tiene desde la raíz, teniendo en cuenta que el nivel de la raíz es cero.
- *Profundidad o altura* de un árbol es el máximo de los niveles de todos los nodos del árbol.
- *Un Bosque* es una colección de dos o más árboles.
- *Grado* de un árbol es el máximo de los grados de sus nodos.

EJEMPLO 20.1 En el siguiente árbol indique el nivel de cada nodo, el grado de algunos nodos, el grado del árbol, así como su profundidad.



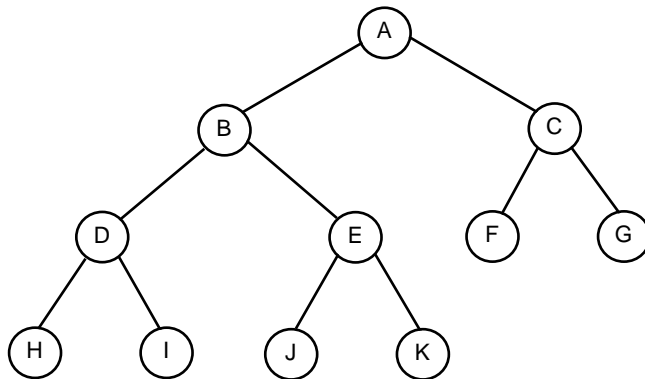
Tiene profundidad 5. El grado del nodo B es 2. El grado del árbol es 3 ya que el grado máximo de todos sus nodos lo da el nodo F que tiene grado 3.

20.2 Árboles binarios

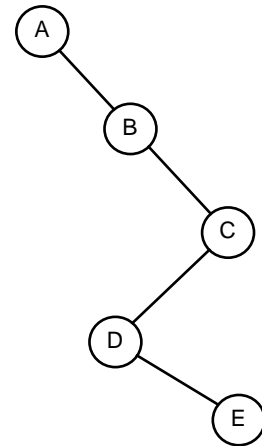
Un *árbol binario* es aquél en el cual cada nodo tiene como máximo grado dos.

- Un árbol binario es *equilibrado* cuando la diferencia de altura entre los subárboles de cualquier nodo es como máximo una unidad.
 - Un árbol binario está *perfectamente equilibrado*, si los subárboles de todos los nodos tienen todos la misma altura.
 - Un árbol binario se dice que es *completo* si todos los nodos *interiores*, es decir aquellos con descendientes, tienen dos hijos.
 - Un árbol binario se dice *lleno* si todas sus hojas están al mismo nivel y todo sus nodos interiores tienen cada uno dos hijos.
- Si un árbol binario es *lleno* entonces es *completo*.

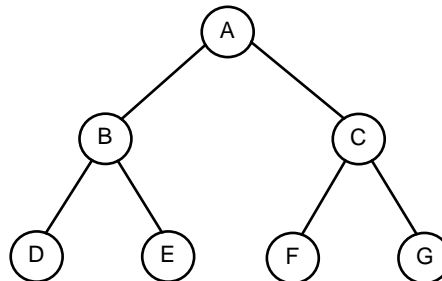
EJEMPLO 20.2 Dibuje tres árboles, uno completo, otro degenerado y otro lleno.



a) completo



b) degenerado



c) lleno

20.3 Estructura y representación de un árbol binario

La estructura de un árbol binario es aquella en la cual en cada nodo se almacena un dato y su hijo izquierdo e hijo derecho. En C puede representarse de la siguientes forma.

```

typedef int TipoElemento;           /* Puede ser cualquier tipo */
struct NodoA
{
    TipoElemento el;
    struct NodoA *hi, *hd;
};
typedef struct NodoA EArbolBin;
typedef EArbolBin *ArbolBinario;
  
```

20.4 Árboles de expresión

Una **expresión** es una secuencia de operadores y operandos debidamente relacionados que forman una fórmula. Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

- Cada hoja es **operando**.
- Los nodos raíz e internos son **operadores**.
- Los subárboles son **subexpresiones** en las que el nodo raíz es un operador.

CONSTRUCCIÓN DE ÁRBOLES DE EXPRESIÓN

Para la construcción de un árbol de expresión a partir de la notación infija se utilizan, como estructuras de datos, una pila de operadores y otra pila de operadores de punteros árbol. Un algoritmo de paso de la notación *infija* a *postfija* es el siguiente:

- Si se lee un operando se crea un árbol de un solo nodo y se mete en la pila de árboles.
- Si se lee un operador se pone en la pila de operadores, de acuerdo con la siguiente regla: el operador se pone en esta pila si tienen prioridad mayor que el que está en la cumbre de la pila o bien la pila está vacía. Si tiene prioridad menor o igual prioridad que el de la cima, se sacan los que hubiera en la pila de mayor o igual prioridad (hasta que quede uno de prioridad mayor o bien la pila esté vacía) y se coloca en ella éste último. El paréntesis abierto se considera como operador de prioridad máxima para obligar a que entre en la pila cuando se lee, y sólo puede salir de la pila cuando aparece un paréntesis derecho.
- Cuando se acaba la entrada de datos hay que sacar todos los operadores que hubiera en la pila.
- Al sacar un operador de la pila de operadores hay que extraer, de la de la pila de árboles, los dos últimos árboles (se considera sólo operadores binarios). Con éstos tres elementos, se forma un nuevo árbol cuya raíz almacena el operador y los punteros *hi*, *hd* apuntan a los dos últimos árboles extraídos de la pila de árboles. Posteriormente se coloca el nuevo árbol en la pila de árboles.
- El proceso termina cuando se acaba la entrada y la pila de operadores queda vacía. El árbol de expresiones que se está buscando se encuentra en la cima de la pila de árboles.

20.5 Recorridos de un árbol

Se denomina recorrido al proceso que permite acceder una sola vez a cada uno de los nodos del árbol. Existen diversas formas de efectuar el recorrido de un árbol binario:

Recorrido en anchura:

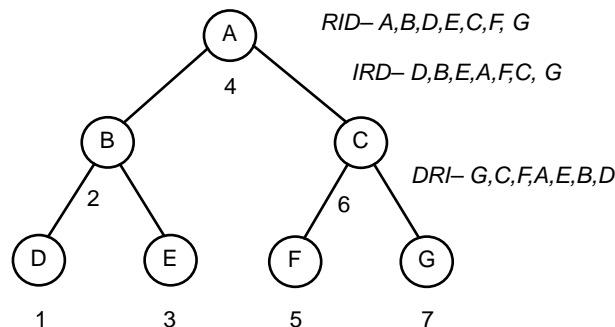
- Consiste en recorrer los distintos niveles (del inferior al superior), y dentro de cada nivel, los diferentes nodos de izquierda a derecha (o bien de derecha a izquierda).

Recorrido en profundidad:

- Preorden RID.** Visitar la raíz, recorrer en preorden el subárbol izquierdo, recorrer en preorden el subárbol derecho.
- Inorden IDR.** Recorrer inorden el subárbol izquierdo, visitar la raíz, recorrer inorden el subárbol derecho.
- Postorden IDR.** Recorrer en postorden el subárbol izquierdo, recorrer en postorden el subárbol derecho, visitar la raíz.

Existen otros tres recorridos más en profundidad pero apenas se usan: RDI, DRI, DIR.

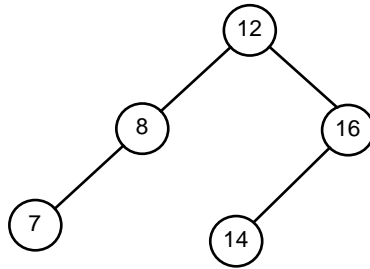
EJEMPLO 20.3 Expresar los recorridos Preorden, Enorden y Postorden del siguiente árbol:



20.6 Árbol binario de búsqueda

Un **árbol binario de búsqueda** es aquel en el cual, dado un nodo cualquiera del árbol, todos los datos almacenados en el subárbol izquierdo son menores que el dato almacenado en este nodo, mientras que todos los datos almacenados en el subárbol derecho son mayores que el dato almacenado en este nodo. En caso de igualdad de claves deben almacenarse en una estructura de datos auxiliar que salga de cada nodo.

EJEMPLO 20.4 El siguiente árbol binario es de búsqueda.



20.7 Operaciones en árboles binarios de búsqueda

Las operaciones mas usuales sobre árboles binarios de búsqueda son: *búsqueda* de un nodo; *inserción* de un nodo; *borrado* de un nodo.

BÚSQUEDA

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

- Si el árbol está vacío la búsqueda termina con fallo.
- La clave buscada se compara con la clave del nodo raíz.
- Si las claves son iguales, la búsqueda se detiene con éxito.
- Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

INSERCIÓN

La operación de *inserción* de un nodo es una extensión de la operación de búsqueda. El algoritmo es:

- Asignar memoria para una nueva estructura nodo.
- Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará siempre como un nuevo nodo hoja.
- Enlazar el nuevo nodo al árbol. Para ello en el proceso de búsqueda hay que quedarse con el puntero que apunta a su padre y enlazar el nuevo nodo a su padre convenientemente. En caso de que no tenga padre(árbol vacío), se pone el árbol apuntando al nuevo nodo.

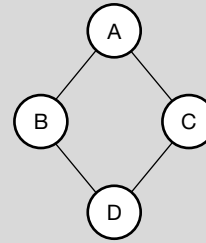
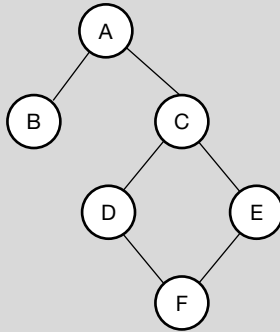
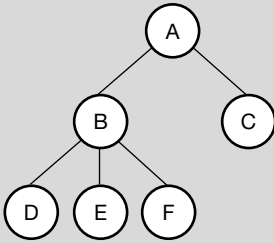
BORRADO

La operación de *borrado* de un nodo es una extensión de la operación de búsqueda. Después de haber buscado el nodo a borrar hay que tener en cuenta:

- Si el nodo es hoja, se suprime, asignando nulo al puntero de su antecesor.
- Si el nodo tiene único hijo. El nodo anterior se enlaza con el hijo del que se quiere borrar.
- Si tiene dos hijos. Se sustituye el valor almacenado en el nodo por el valor, inmediato superior (o inmediato inferior). Que se encuentra en un avance a la derecha (izquierda) del nodo a borrar y todo a la izquierda (derecha), hasta que se encuentre NULL. Posteriormente se borra el nodo que almacena el valor inmediato superior (o inmediato inferior) que tiene como máximo un hijo.
- Por último hay que liberar el espacio en memoria ocupado el nodo.

PROBLEMAS RESUELTOS

20.1. Explicar por qué cada una de las siguientes estructuras no es un árbol binario.

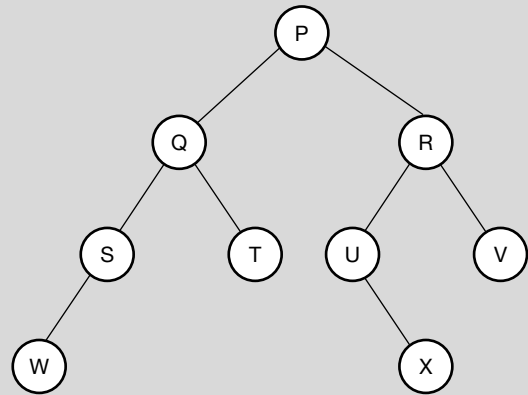


Solución

La primera no es un árbol binario ya que el nodo cuyo contenido es B tiene tres hijos y el máximo número de hijos de un árbol binario es dos. La segunda porque hay dos caminos distintos para ir al nodo F y por tanto no se expresa la jerarquía de la definición de árbol. La tercera por la misma razón que la segunda.

20.2. Considérese el árbol siguiente:

- ¿Cuál es su altura?
- ¿Está el árbol equilibrado?. ¿Por qué?
- Listar todos los nodos hoja.
- ¿Cuál es el predecesor inmediato (padre) del nodo U?
- Listar los hijos del nodo R.
- Listar los sucesores del nodo R.



Solución

- Su altura es cuatro.
- El árbol está equilibrado ya que la diferencia de las alturas de los subárboles izquierdo y derecho es como máximo uno.
- Los nodos hoja son: W, T, X, V.
- El predecesor inmediato (padre) del nodo U es el nodo que contiene R.
- Los hijos del nodo R son U y V.
- Los sucesores del nodo R son U, V, X.

20.3. Para el árbol del ejercicio anterior realizar los siguientes recorridos: RDI, DRI, DIR

Solución

Recorrido RDI : P, R, V, U, X, Q, T, S, W.
 Recorrido DRI : V, R, X, U, P, T, Q, S, W.
 Recorrido DIR : V, X, U, R, T, W, S, Q, P.

20.4. Escriba las declaraciones necesarias para trabajar con árboles binarios de números enteros y las funciones CrearNodo, Construir, Hijo Izquierdo e Hijo Derecho.

Análisis del problema

- Para hacer la declaración basta con declarar el tipo elemento como un entero, y definir una estructura que almacene elementos de ese tipo y dos punteros a la propia estructura.
- La función `CrearNodo` es una función que se encarga de recibir un dato como parámetro, y devuelve un nodo de tipo árbol con su hijo izquierdo e hijo derecho apuntando a `NULL`.
- La función `Construir` recibe como parámetro un dato así como dos árboles, y retorna un árbol cuya raíz es un nodo que contiene el dato y cuyos hijos izquierdo y derecho son los punteros árboles que recibe como parámetro.
- La función `Hi` (Hijo Izquierdo), recibe como parámetro un árbol y devuelve su Hijo Izquierdo si es que lo tiene.
- La función `Hd` (Hijo Derecho), recibe como parámetro un árbol y devuelve su Hijo Derecho si es que lo tiene.

Codificación

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

typedef int TipoElemento;
struct nodo
{
    TipoElemento el;
    struct nodo *hi, *hd;
};
typedef struct nodo Nodo;

Nodo* CrearNodo(TipoElemento el);
Nodo* Construir(TipoElemento el, Nodo * hi, Nodo * hd);
Nodo* Hi( Nodo * a);
Nodo* Hd( Nodo * a);

Nodo* CrearNodo(TipoElemento el)
{
    Nodo* t;

    t = (Nodo*) malloc(sizeof(Nodo));
    t -> el = el;
    t ->hi = t -> hd = NULL;
    return t;
}

Nodo* Construir(TipoElemento el, Nodo * hi, Nodo * hd)
{
    Nodo *nn;
    nn = CrearNodo(el);
    nn->hi = hi;
    nn->hd = hd;
    return( nn );
}

Nodo* Hi( Nodo * a)
{
    if(a)
        return(a->hi);
}
```

```

        else
        {
            printf(" error en Hi \n");
            return(NULL);
        }
    }

Nodo* Hd( Nodo * a)
{
    if(a)
        return(a->hd);
    else
    {
        printf(" error en Hd \n");
        return(NULL);
    }
}

void main ()
{
    ....
    ....
}

```

- 20.4.** *Escribir las sentencias necesarias para construir un árbol cuya raíz sea el número 9, cuyo hijo izquierdo sea el 6 y cuyo hijo derecho sea el 12.*

Codificación

```

Nodo * hi,*hd, *a;
....
....
hi=Construir(6, NULL, NULL);
hd=Construir( 12, NULL, NULL);
a=Constgruir(9,hi,hd);
....
....

```

- 20.5.** *Escribir funciones para construir un árbol vacío, decidir si un árbol es vacío, y decidir si un árbol es una hoja.*

Análisis del problema

Usando una declaración de árbol se tiene las siguientes funciones:

VaciaA : crea el árbol vacío (NULL).

EsVacioA: proporciona el valor verdadero si el árbol que recibe como parámetro está vacío y falso en otro caso.

EsHoja: decide si un árbol es una hoja; es decir, si su hijo izquierdo e hijo derecho son vacíos.

Codificación

```

int EsVacioA(Nodo *a)
{
    return(a == NULL);
}

```

```

void VaciaA(Nodo **a)
{
    (*a) = NULL;
}

int Eshoja(Nodo *a)
{
    if (a)
        return( (a->hi == NULL)&& a->hd == NULL);
    else
        return (0);
}

```

20.6. *Escribir funciones que digan el numero de hojas de un árbol, el número de nodos de un árbol, el número de nodos que no son hojas, la altura de un árbol, y los nodos que hay en un cierto nivel.*

Análisis del problema

Las funciones solicitadas se codifican recursivamente:

- **Altura.** La función se calcula de la siguiente forma: un árbol está vacío su altura es cero, y si no está vacío su altura es uno mas que el máximo de las alturas de cada uno de sus hijos.
- **NdeHojas.** Para calcular el número de hojas de un árbol, basta con observar, que si un árbol es vacío su número de horas es cero. Si no está vacío, hay dos posibilidades, que el árbol sea una hoja en cuyo caso vale 1, o que no lo sea con lo que para calcularlas basta con sumar las hojas que tenga el hijo izquierdo y el hijo derecho.
- **NdeNodos.** Si un árbol es vacío el número de nodos que tiene es cero, y si no está vacío el número de nodos es uno más que la suma del número de los nodos que tenga su hijo izquierdo y su hijo derecho.
- **NdeNodosInternos.** Se calcula restando los valores de las funciones **NdeNodos** y **NdeHojas**.
- **NodosNivel.** Si el árbol está vacío los nodos que hay en ese nivel es cero. En otro caso, hay que contar o escribir el nodo si está en el nivel 1, y en caso de que no lo esté hay que calcular los nodos existentes en el nivel inmediatamente siguiente (restar uno al nivel que buscamos) de su hijo izquierdo e hijo derecho.

Codificación

```

int Altura(Nodo *a)
{
    if (!a)
        return(0);
    else
    {
        int ai,ad;

        ai = Altura(a->hi);
        ad = Altura(a->hd);
        if (ai < ad)
            return (1 + ad);
        else
            return (1 + ai);
    }
}

int NdeHojas(Nodo *a)
{
    if (!a)

```



```

        return(0);
    else
        return (NdeHojas(a->hi) + NdeHojas(a->hd));
    }

int NdeNodos(Nodo *a)
{
    if(!a)
        return(0);
    else
        return(1+ NdeNodos(a->hi) + NdeNodos(a->hd));
}

int NdeNodosInternos(Nodo *a)
{
    return (NdeNodos(a)-NdeHojas(a));
}

void NodosNivel(Nodo *a, int n)
{
    if (a)
        if (n == 1)
            printf("%d\n", a->el);
        else
        {
            NodosNivel(a->hi,n - 1);
            NodosNivel(a->hd,n - 1);
        }
}

```

20.7. *Escriba funciones que copien un árbol en otro, y que den la imagen especular.*

Análisis del problema

- **Copia.** Es una función que recibe como parámetro un árbol *a* y da una copia exacta de él. Para resolver el problema se hace lo siguiente: si el árbol es vacío una copia de él es el propio árbol vacío. En otro caso hay que copiar en un nuevo nodo el dato almacenado en la raíz y después copiar el hijo izquierdo e hijo derecho.
- **Espejo.** El espejo de un árbol vacío es el propio árbol vacío. El espejo de un árbol no vacío (imagen especular) se obtiene cambiando entre sí el hijo izquierdo y el hijo derecho. Por lo tanto la función es análoga a la de *Copia*, excepto que se cambian los parámetros de llamada (el hijo izquierdo pasa a ser hijo derecho y recíprocamente).

Codificación

```

void Copiar (Nodo *a, Nodo **Acop)
{
    if (a)
    {
        (*Acop) = CrearNodo(a->el);
        Copiar(a->hi,&(*Acop)->hi);
        Copiar(a->hd,&(*Acop)->hd);
    }
}

void Espejo (Nodo *a, Nodo **Aesp)
{

```

```

    if (a)
    {
        (*Aesp) = CrearNodo(a->el);
        Espejo(a->hi, & (*Aesp)->hd);
        Espejo(a->hd, & (*Aesp)->hi);
    }
}

```

20.8. *Escriba funciones para hacer los recorridos recursivos en profundidad Inorden, Preorden, Postorden.*

Análisis de problema

De los seis posibles recorridos en profundidad IDR, IRD, DIR, DRI, RID, RDI se pide IDR, IRD RID.

- *Inorden.* Se recorre el hijo izquierdo, se visita la raíz y se recorre el hijo derecho. Por lo tanto la función debe codificarse de la siguiente forma: Si el árbol es vacío no se hace nada. En otro caso, se recorre recursivamente el hijo izquierdo, se escribe la raíz, y posteriormente se recorre recursivamente el hijo derecho.
- *Preorden.* Se visita la raíz. Se recorre el hijo izquierdo, y después se recorre el hijo derecho. Por lo tanto la codificación es análoga a la de *Inorden*, pero cambiando el orden de llamadas.
- *Postorden.* Se recorre el hijo izquierdo, se recorre el hijo derecho, y posteriormente se visita la raíz.

Codificación

```

void Inorden(Nodo *a)
{
    if (a)
    {
        Inorden(a->hi);
        printf("%d ", a->el);
        Inorden(a->hd);
    }
}

void Preorden(Nodo *a)
{
    if (a)
    {
        printf("%d ", a->el);
        Preorden(a->hi);
        Preorden(a->hd);
    }
}

void Postorden(Nodo *a)
{
    if (a)
    {
        Postorden(a->hi);
        Postorden(a->hd);
        printf("%d ", a->el);
    }
}

```

20.9. Se dispone de un árbol binario de elementos de tipo entero. Escriba funciones que calculen:

- a) La suma de sus elementos.
- b) La suma de sus elementos que son múltiplos de 3.

Análisis del problema

Para resolver el problema basta con implementar las dos funciones efectuando al hacer un recorrido del árbol las correspondientes operaciones.

Codificación

```
int Suma (Nodo*a)
{
    if(a)
        return( a->el + Suma(a->hi) + Suma(a->hd));
    else
        return(0);
}

int SumaMultimos (Nodo*a)
{
    if(a)
        if (a->el%3)
            return( a->el + SumaMultimos(a->hi)+SumaMultimos(a->hd));
        else
            return( SumaMultimos(a->hi) + SumaMultimos(a->hd));
    else
        return(0);
}
```

20.10. Escribir una función booleana *Identicos* que permita decir si dos árboles binarios son iguales.

Análisis del problema

Dos árboles son idénticos si tiene la misma estructura y contienen la misma información en cada uno de sus distintos nodos.

Codificación

```
int Identicos (Nodo *a, Nodo * a1)
{
    if (a)
        if (a1)
            return((a->el == a1->el) && Identicos(a->hi,a1->hi) && Identicos(a->hd,a1->hd));
        else // a ≠ NULL y a1 es NULL
            return(0);
        else
            if (a1) // a es NULL y a1 no
                return(0);
            else
                return(1);
}
```

20.11. Construir una función recursiva para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado (el campo clave es de tipo entero).

Análisis del problema

Basta con hacer un recorrido del árbol y escribir los que cumplan la condición dada.

Codificación

```
void RecorreMayores(Nodo *a, TipoElemento el)
{
    if (a)
    {
        if(a->el > el)
            printf("%d ", a->el);
        RecorreMayores(a->hi, el);
        RecorreMayores(a->hd, el);
    }
}
```

20.12. Dado un árbol binario de búsqueda diseñe una función que liste los nodos del árbol ordenados descendentemente.

Análisis del problema

Basta con hacer el recorrido DRI del árbol.

Solución

```
void Escribe(Nodo *a)
{
    if (a)
    {
        Escribe(a->hd);
        printf("&d ", a->el);
        Escribe(a->hi);
    }
}
```

20.13. Escribir un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:

- La altura de cada nodo del árbol.
- La diferencia de altura entre rama izquierda y derecha de cada nodo.

Análisis del problema

El programa que se presenta tiene las siguientes funciones:

- **Altura.** Calcula la altura del árbol que se le pasa como parámetro.
- **GeneraAleatoriamente.** Genera aleatoriamente un árbol binario de búsqueda usando la función **AnadeA** (Codificada en un ejercicio posterior 20.17) que añade número enteros a un árbol binario de búsqueda.
- **EscribeA.** Escribe el árbol de acuerdo con el recorrido DIR.
- **MuestraP.** Muestra en pantalla para cada nodo las condiciones pedidas usando un recorrido IRD.

Codificación (Se encuentra en la página web del libro)

20.14. Escriba una función que realice el recorrido en anchura de un árbol binario.

Análisis del problema

Para hacer el recorrido en anchura se usa el esquema de arriba abajo y de izquierda a derecha. También se codifica el esquema de arriba abajo y de derecha a izquierda, mediante las funciones `AnchuraID` y `AnchuraDI`. Para ambas funciones se usa una cola de árboles por la que pasarán todos los árboles que apunten a los nodos del árbol.

- `AnchuraID`. Se crea en primer lugar la cola vacía. Si el árbol original es vacío no se hace nada y en otro caso se añade el propio árbol a la cola. Ahora mediante un bucle mientras controlado por la condición es vacía la cola, se extrae el árbol que está como primer elemento, se borra de la cola, se escribe la información almacenada en el nodo raíz, y posteriormente se añade el árbol hijo izquierdo del nodo raíz a la cola si está no vacío y posteriormente el árbol hijo derecho del nodo raíz a la cola si está no vacío.
- `AnchuraDI`. Es análoga a la anterior, pero la final se cambia el orden de añadir a la cola el árbol hijo izquierdo y el árbol hijo derecho.

En la codificación que se presenta, se incluye la declaración de árbol la de cola, y se omiten las primitivas de gestión de una cola.

Codificación

```
typedef int TipoElemento;
struct nodo
{
    TipoElemento el;
    struct nodo *hi, *hd;
};
typedef struct nodo Nodo;

// gestion de Cola
struct NodoC
{
    Nodo *el;
    struct NodoC* sig;
};

typedef struct
{
    NodoC * Frente;
    NodoC * Final;
}Cola;

// Las primitivas de gestión d e una cola son omitidas

void AnchuraID(Nodo *a)
{
    Cola C;
    Nodo *a1;
    VacíaC(&C);
    if (a)
    {
        AnadeC(&C,a);
        while (!EsVacíaC(C))
        {
            a1 = PrimeroC(C);
```

```

        BorrarC(&C);
        printf("%d \n", a1->el);
        if (a1->hi != NULL)
            AnadeC(&C, a1->hi);
        if (a1->hd != NULL)
            AnadeC(&C, a1->hd);
    }
}

void Anchura DI(Nodo *a)
{
    Cola C; Nodo *al;

    VacíaC(&C);
    if (a)
    {
        AnadeC(&C, a);
        while (!EsVacíaC(C))
        {
            al = PrimeroC(C);
            BorrarC(&C);
            printf("%d \n", al->el);
            if (al->hd != NULL)
                AnadeC(&C, al->hd);
            if (al->hi != NULL)
                AnadeC(&C, al->hi);
        }
    }
}

```

20.15. *Escriba las funciones para recorrer un árbol en inorden y preorden iterativamente.*

Análisis del problema

Una manera de resolver el problema es usar la técnica general de eliminación de la recursividad. En este caso se usa una pila de árboles, por la que pasarán punteros a todos los nodos del árbol de la siguiente forma.

- **InordenNoRecursivo.** Usa una pila de árboles no escritos. Inicialmente se pone la pila a vacío y después mediante un bucle repetir que termina cuando la pila está vacía y cuando el último hijo derecho del nodo del que se ha escrito su raíz (y por tanto también su hijo izquierdo) está vacío. Hay que estar seguros de que cuando se escriba un nodo ya se ha escrito su hijo izquierdo. Para ello el primer bucle `while` se desplaza (baja) por los hijos izquierdos añadiendo los punteros correspondientes a la pila hasta asegurar que el primer árbol almacenado en la cumbre e la pila ya se ha escrito su hijo izquierdo, y falta por escribir su raíz y su hijo derecho. Por tanto, a cada vuelta del bucle `do while` y después de haberse ejecutado el bucle `while` (posterior) hay en la cumbre de la pila de árboles no escritos un puntero a un árbol del que se ha escrito su hijo izquierdo y falta por escribir su raíz y el hijo derecho. Por lo tanto después del bucle `while` hay que extraer un dato de la pila, escribir su raíz para posteriormente tratar su hijo derecho, añadiéndolo a la pila repitiendo el proceso general.
- **PreordenNoRecursivo.** Usa una pila de árboles escritos. Inicialmente se pone la pila a vacío y después se usa un bucle repetir que termina cuando la pila está vacía y cuando el último hijo derecho del nodo del que se ha escrito su hijo derecho y su raíz está vacío. Hay que asegurar que cuando se escriba un hijo izquierdo ya se ha escrito su raíz. Para ello el primer bucle `while` se desplaza (baja) por los hijos izquierdos escribiendo la información de la raíz y añadiendo los punteros correspondientes a la pila hasta asegurar que el primer árbol almacenado en la cumbre de la pila ya se ha escrito su hijo izquierdo y su raíz, y falta por escribir su hijo derecho. Por tanto a cada vuelta del bucle `do while` y después de haberse ejecutado el bucle `while` (posterior) hay en la cumbre de la pila de árboles escritos, un

puntero a un árbol del que se ha escrito su raíz y su hijo izquierdo y falta por escribir el hijo derecho. Al terminar el bucle `while` hay que extraer un dato de la pila, para posteriormente tratar su hijo derecho, añadiéndolo a la pila repitiendo el proceso general.

Se presentan a continuación las declaraciones necesarias y las dos funciones pedidas. Se usa una pila de árboles que no está incluida.

Codificación (*Se encuentra en la página web del libro*)

20.16. *Escriba una función recursiva y otra iterativa que se encargue de buscar la información dada en un elemento en un árbol binario de búsqueda.*

Análisis del problema

Se usa una definición de árboles como la usada los ejercicios anteriores, y se implementa la función `BuscarA` y `BuscarAIterativo` que busca el nodo que contenga el elemento.

- `BuscarA`. Realiza la búsqueda recursiva de la siguiente forma: si el árbol está vacío la búsqueda termina con fallo. Si no está vacío, la búsqueda termina con éxito si la información almacenada en la raíz coincide con la que se está buscando, en otro caso habrá que continuar la búsqueda o por el hijo izquierdo o bien por el hijo derecho dependiendo de la comparación entre el elemento que se busca y la información almacenada en el nodo.
- `BuscarAIterativo`. Realiza la búsqueda devolviendo un puntero al nodo anterior además del puntero al nodo que se busca. Para ello se usa un interruptor `enc` que se inicializa a falso, y mediante un bucle *mientras no* `enc` y *no se haya terminado el árbol* hacer: si coinciden los contenidos poner `enc` a verdadero, en otro caso se desplaza a la izquierda o la derecha quedándose con su anterior dependiendo del contenido de la raíz y del elemento que se esté buscando.

Codificación

```
Nodo* BuscarA (Nodo* p, TipoElemento el)
{
    if (!p)
        return 0;
    else if (el == p -> el)
        return p;
    else if (el < p -> el)
        return BuscarA (p -> hi, el);
    else
        return BuscarA (p -> hd, el);
}

Nodo*BuscarAIterativo(Nodo * a, Nodo** ant, TipoElemento el)
{
    int enc;
    Nodo *anterior;

    enc = 0;
    anterior = NULL;
    while (!enc && (a!=NULL) )
    {
        if (a->el == el)
            enc = 1;
        else
        {
            anterior = a ;
```

```

        if (el < a->el)
            a = a->hi;
        else
            a = a->hd;
    }
}
*ant = anterior;
return a;
}

```

20.17. *Escriba una función recursiva y otra iterativa que se encargue de insertar la información dada en un elemento en un árbol binario de búsqueda.*

Análisis del problema

Se usa una definición de árboles como la de los ejercicios anteriores, y se implementa la función `AnadeA` y `AnadeAIterativo` que insertan el nodo que contenga el elemento.

- `AnadeA`. Realiza la inserción recursiva de la siguiente forma: si el árbol está vacío se inserta el nuevo nodo como una hoja de un árbol llamando a la función `CrearNodo`. Si no está vacío si la información almacenada en la raíz coincide con la que está buscando habría que tratar las claves repetidas almacenándolas en una estructura de datos auxiliar (no se hace), en otro caso habrá que continuar la inserción o por el hijo izquierdo o bien por el hijo derecho dependiendo de la comparación entre el elemento que vamos a insertar y la información almacenada en el nodo.
- `AnadeAIterativo`. Se llama a la función `BuscarAIterativo` programada en el ejercicio anterior 20.16 y después se inserta el nodo si la búsqueda terminó en fallo teniendo en cuenta que puede ser el raíz total o bien un hijo izquierdo o bien un hijo.
- `CreaNodo`. Es una función auxiliar que recibe como parámetro un elemento y devuelve un puntero a un árbol que es una hoja y que contiene la información del elemento.

Codificación

```

Nodo* CrearNodo(TipoElemento el)
{
    Nodo* t;

    t = (Nodo*) malloc(sizeof(Nodo));
    t -> el = el;
    t -> hi = t -> hd = NULL;
    return t;
}

void AnadeA (Nodo** a, TipoElemento el)
{
    if (!(*a))
        *a = CrearNodo(el);
    else
        if (el == (*a)->el)
            printf( " valor %d repetido no se inserta\n",el);
        else
            if (el < (*a) -> el)
                AnadeA (&((*a) -> hi), el);
            else
                AnadeA (&((*a) -> hd), el);
}

```



```

void AnadeAIterativo(Nodo** a, TipoElemento el)
{
    Nodo *nn, *a1, *ant;

    a1 = BuscarAIterativo(*a, &ant, el);
    if (a1==NULL)
    {
        nn = CrearNodo(el);
        if (ant == NULL)
            *a = nn;
        else
            if( el < ant->el)
                ant->hi = nn;
            else
                ant->hd = nn;
    }
    else
        printf(" nodo duplicado no se inserta \n");
}

```

20.18. *Escriba funciones iterativas y recursivas para borrar un elemento de un árbol binario de búsqueda.*

Análisis del problema

El borrado que se implementa es el explicado en la teoría, usando la técnica del predecesor inmediato que se encuentra uno a la izquierda y todo a su derecha. Las funciones que lo codifican son:

- **BorrarARecursivo.** Realiza la búsqueda del nodo a borrar recursivamente, y una vez encontrado el nodo considera los tres casos. No tiene hijo izquierdo, en cuyo caso se enlaza el puntero con su hijo derecho. No tiene hijo derecho, en cuyo caso se enlaza el nodo con su hijo izquierdo. Tiene dos hijos, en cuyo caso se llama a una función `recorred` que se encarga de buscar el sucesor inmediato, copia la información en el nodo que se quiere borrar y cambia el nodo a borrar que es el que es ahora el predecesor inmediato. Por último se libera memoria.
- **Recorred.** Es una función recursiva que hace lo indicado anteriormente y realiza el enlace con el hijo izquierdo del predecesor inmediato.
- **BorrarAIterativo.** Realiza a búsqueda del nodo a borrar iterativamente. En el caso de éxito en la búsqueda considera los tres casos considerados en el `BorrarARecursivo`, pero ahora, al realizar los dos primeros casos (no tiene hijo izquierdo, o no tiene hijo derecho) ha de tener en cuenta si el nodo a borrar es el raíz del árbol (`ant==NULL`) a la hora de realizar los enlaces. Para el caso del borrado del nodo con dos hijos, la búsqueda del predecesor inmediato se realiza iterativamente mediante la condición no tiene hijo derecho. Una vez encontrado, se intercambian la información y se procede al borrado del nodo predecesor inmediato.
- **BorrarA.** Realiza la búsqueda recursivamente de acuerdo con la función `BorrarARecursivo`, pero en lugar de llamar a la función `recorred`, realiza el borrado iterativamente tal y como lo hace la función `BorrarAIterativo`.

Codificación

```

void BorrarARecursivo (Nodo** a, TipoElemento el)
{
    if (!(*a))
        printf("!! Registro con clave %d no se encuentra !!. \n",el);
    else
        if (el < (*a)->el)
            BorrarA(&(*a)->hi, el);
        else
            if (el > (*a)->el)

```

```

        BorrarA(&(*a)->hd,el);
    else
    {
        Nodo* ab;                                /* Nodo a borrar */
        ab = (*a);
        if (ab->hi == NULL)
            (*a) = ab->hd;
        else
            if (ab->hd == NULL)
                (*a) = ab->hi;
            else
                /* mayor de los menores */
                recorred(a, &(*a)->hi,&ab);
        free(ab);
    }
}

void recorred(Nodo **a, Nodo **por, Nodo**ab)
{
    if((*por)->hd)
        recorred(a,&(*por)->hd,ab);
    else
    {
        (*a)->el = (*por)->el;                    // copia
        *ab = *por;                                // nuevo nodo a borrar
        *por = (*por)->hi;                        // enlace
    }
}

void BorrarAIterativo(Nodo** a, TipoElemento el)
{
    Nodo *a1,*ant,*ab ;

    int enc = 0;
    ant = NULL;
    a1 = *a;
    while (! enc &&(a1 != NULL))
    {
        if (el == a1->el)
            enc = 1;
        else
        {
            ant = a1;
            if(el < a1->el)
                a1 = a1->hi;
            else
                a1 = a1->hd;
        }
    }
    if(a1)
    {
        ab = a1;
        if (ab->hi == NULL)

```

```

        if (ant == NULL)
            (*a) = ab->hd;
        else
            if (el < ant->el)
                ant->hi = ab->hd;
            else
                ant->hd = ab->hd;
    else
        if(ab->hd == NULL)
            if (ant == NULL)
                (*a) = ab->hi;
            else
                if (el < ant->el)
                    ant->hi = ab->hi;
                else ant->hd = ab->hi;
    else // tiene dos hijos
    {
        Nodo* a1, *p;
        p = ab;
        a1 = ab->hi;
        while (a1->hd)
        {
            p = a1;
            a1 = a1->hd;
        }
        ab->el = a1->el; // Se copia dato
        if (p == ab)
            ab->hi = a1->hi; //p->hi no enlaza bien por ser una copia
        else
            p->hd = a1->hi;
        ab = a1;
    }
    free(ab);
}

else
    printf ("Elemento no encontrado\n");
}

void BorrarA (Nodo** r, TipoElemento el)
{
    if (!(*r))
        printf("!! Registro con clave %d no se encuentra !!. \n",el);
    else
        if (el < (*r)->el)
            BorrarA(&(*r)->hi, el);
        else
            if (el > (*r)->el)
                BorrarA(&(*r)->hd,el);
            else
            {
                Nodo* ab; // Nodo a borrar */
                ab = (*r);
                if (ab->hi == NULL)

```

```

    (*r) = ab->hd;
else
    if (ab->hd == NULL)
        (*r) = ab->hi;
    else
        /* mayor de los menores */
        {
            Nodo* a, *p;
            p = ab;
            a = ab->hi;
            while (a->hd)
            {
                p = a;
                a = a->hd;
            }
            ab->el = a->el;
            if (p == ab)
                ab->hi = a->hi;

            else
                p->hd = a->hi;
            ab = a;
        }
    free(ab);
}
}

```

//p->hi no enlaza bien por ser una copia

PROBLEMAS PROPUESTOS

20.1. Para cada una de las siguientes listas de letras

- Dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado.
- Realizar recorridos en *inorden*, *preorden* y *postorden* del árbol y mostrar la secuencia de letras que resultan en cada caso.

(i)	M,	Y,	T,	E,	R
(ii)	R,	E,	M,	Y,	T
(iii)	T,	Y,	M,	E,	R
(iv)	C,	O,	R,	N,	F,
	L,	A,	K,	E,	S

20.2. Dibujar los árboles binarios que representan las siguientes expresiones:

- $(A+B)/(C-D)$
- $A+B+C/D$
- $A-(B-(C-D)/(E+F))$
- $(A+B)*((C+D)/(E+F))$
- $(A-B)/((C*D)-(E/F))$

20.3. El recorrido preorden de un cierto árbol binario produce ADFGHKLPQRWZ y el recorrido *en inorden* produce GFHKDLAWRQPZ. Dibujar el árbol binario.

20.4. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

I (seguido de un carácter)	: Insertar un carácter
B (seguido de un carácter)	: Buscar un carácter
RE	: Recorrido en orden
RP	: Recorrido en preorden
RT	: Recorrido postorden
SA	: Salir

20.5. Escribir una función *booleana* a la que se le pase un puntero a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y falso en caso contrario.

20.6. Crear un archivo de datos en el que cada línea contenga la siguiente información

Columnas	1-20	Nombre
	21-31	Número de la Seguridad Social
	32-78	Dirección

Escribir un programa que lea cada registro de datos de un árbol, de modo que cuando el árbol se recorra utilizando recorrido en orden, los números de la seguridad social se ordenen en orden ascendente. Imprimir una cabecera “DATOS DE EMPLEADOS ORDENADOS POR NUMERO SEGURIDAD SOCIAL”. A continuación se han de imprimir los tres datos utilizando el siguiente formato de salida.

Columnas:	1-11	Número de la Seguridad Social	25-44
		Nombre;	58-104
			Dirección.

- 20.7.** Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición. Hacer uso de la estructura árbol binario de búsqueda, cada nodo del árbol que tenga una palabra y su frecuencia.
- 20.8.** Crear un archivo de datos en el que cada línea contenga la siguiente información: Nombre 30 caracteres; Número de la Seguridad Social 10 caracteres; Dirección 24 caracteres. Escribir un programa que lea cada registro de datos en un árbol, de modo que cuando el árbol se recorra en orden los números de la Seguridad Social se almacenen en orden ascendente. Imprimir una cabecera “DATOS DE EMPLEADOS ORDENADOS POR

NUMERO DE LA SEGURIDAD SOCIAL” y a continuación imprimir los datos del árbol con el formato Columnas: 1-10 Número de la Seguridad Social; 20-50 Nombre; 55-79 Dirección

- 20.9.** Diseñar un programa interactivo que permita dar altas, bajas, listar, etc. en un árbol binario de búsqueda.
- 20.10.** Dados dos árboles binarios de búsqueda indicar mediante un programa si los árboles tienen o no elementos comunes.
- 20.11.** Un árbol binario de búsqueda puede implementarse con un *array*. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición *I* del array, su hijo izquierdo se encuentra en la posición $2*I$ y su hijo derecho en la posición $2*I + 1$. Diseñar a partir de esta representación las correspondientes funciones para gestionar interactivamente un árbol de números enteros. (Comente el inconveniente de esta representación de cara al máximo y mínimo número de nodos que pueden almacenarse).
- 20.12.** Una matriz de *N* elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordene ascendentemente la cadena de caracteres.
- 20.13.** Escriba un programa C que lea una expresión correcta en forma infija y la presente en notación postfija.

Compilación de programas C en UNIX y LINUX

La forma de compilar programas C en el entorno UNIX varía considerablemente entre las diferentes plataformas UNIX. Las versiones de Linux y FreeBSD 3.4 de UNIX usan el potente compilador GNU. Para conocer la versión disponible se ejecuta la orden:

```
$ gcc -version
2.7.2.3
$
```

La orden `cc` es la más usada en las plataformas de UNIX para compilar programas C, como se muestra en la siguiente sesión de una versión de BSD:

```
$ type cc
cc is a tracked alias for /usr/bin/cc
$ ls -li /usr/bin/cc
7951 -r-xr-xr-x 2 root wheel 49680 Dec 20 00:46 /usr/bin/cc

$ type gcc
gcc is a tracked alias for /usr/bin/gcc
$ ls -li /usr/bin/gcc
7951 -r-xr-xr-x 2 root wheel 49680 Dec 20 00:46 /usr/bin/gcc
$
```

Otras plataformas UNIX proporcionan sus propios compiladores de C y C++, los cuales difieren substancialmente en las opciones que permiten del compilador de GNU, así como en los mensajes que se producen y su capacidad de optimización. A continuación se verán algunas de las diferencias.

A.1 Orden (comando) de compilación cc

La mayoría de las plataformas UNIX invocan sus compiladores de C con el nombre `cc`. Las plataformas Linux y FreeBSD tienen el nombre de comando `gcc`, además del nombre `cc`. Algunas veces el compilador de GNU es instalado como `gcc` en plataformas comerciales para distinguirlo del estándar. Por ejemplo, HP incluye un compilador no ANSI con su sistema ope-

rativo HPUX, que es denominado el compilador “envuelto” (este compilador es suficiente para reconstruir un nuevo kernel para HPUX). El compilador ANSI debe ser adquirido por separado y, cuando se instala, reemplaza al comando `cc`.

Sin embargo, dentro de la misma plataforma, hay también hay otras opciones. HPUX 10.2 soporta el compilador `cc` y el compilador conforme con POSIX (estándar) `c89`. La plataforma IBM AIX 4.3 soporta un compilador “extendido” de C, `cc`, y un compilador de ANSI C, `xlc` o `c89`. La diferencia entre los compiladores `xlc` y `c89` en AIX son las opciones por defecto configuradas. Las opciones, relativamente estandarizadas, son:

OPCIÓN -C

Esta opción es probablemente la más estandarizada universalmente. La opción `-c` indica que el compilador debería producir un archivo (fichero) objeto (*fichero.o*) pero sin intentar enlazar para obtener un ejecutable. Esta opción se usa cuando se compilan varios módulos fuentes separados que serán enlazados juntos en una etapa posterior por medio del enlazador. Por ejemplo, se ha editado el archivo fuente `ecuacion.c`, la compilación con el comando `cc` y la opción `-c`:

```
$ cc -c ecuacion.c
```

El resultado de la compilación es un listado con los errores sintácticos del programa. O bien, de no haber errores, el archivo con el código objeto `ecuacion.o`. Una vez generado el código objeto, se enlaza y se genera el archivo ejecutable:

```
$ cc ecuacion.o
```

El siguiente ejemplo muestra como se compila y enlaza en un solo paso:

```
$ cc hello.c
```

Esta orden, de paso único, traduce el archivo fuente escrito en C `hello.c`; el resultado de la compilación, si no hay errores, es el archivo ejecutable `a.out`. El nombre de fichero `a.out` es el nombre por defecto de un ejecutable que se genera como salida del compilador y del enlazador (*link*). Esta práctica se remonta al menos a 1970 cuando UNIX estaba escrito en lenguaje ensamblador sobre el PDP-11. El nombre de los archivos de salida por defecto del enlazador de Digital Equipment (DEC) también es `a.out`.

El programa C se puede escribir en varios módulos y cada uno estar guardado en un archivo. La compilación puede hacerse archivo tras archivo y después enlazarse para formar el archivo ejecutable. Por ejemplo, la aplicación de cálculo de nóminas se escribe en los archivos independientes: `nomina1.c`, `nomina2.c` y `nomina3.c`. La compilación de cada archivo fuente:

```
$ cc -c nomina1.c
$ cc -c nomina2.c
$ cc -c nomina3.c
```

A continuación se enlazan los tres archivos objetos generados (una vez que no hay errores sintácticos) como sigue:

```
$ cc nomina1.o nomina2.o nomina3.o
```

el resultado es el archivo ejecutable `a.out`. La orden `cc` con la opción `-c`, ejecutado para cada archivo fuente, produce, respectivamente, los archivos `nomina1.o`, `nomina2.o` y `nomina3.o`. Después, la orden `cc` acepta cada archivo objeto como entrada y produce el archivo ejecutable final con el nombre `a.out`. A continuación, se puede ejecutar el programa generado.

OPCIÓN -O

Esta opción es también bastante estándar. La opción `-o` permite al usuario especificar el nombre del archivo de salida. Por ejemplo, para el archivo `ecuacion.c` podría hacerse:

```
$ cc -c ecuacion.c -o mat_ecuacion.o
```

La opción `-c` indica que se va a producir un archivo objeto y la opción `-o` nombrará el archivo objeto de salida como `mat_ecuacion.o`.

La opción `-o` puede usarse también para nombrar el archivo ejecutable. Por ejemplo, el archivo ejecutable que se genera, a continuación, se nombra `prog_ecuacion`:

```
$ cc mat_ecuacion.o -o prog_ecuacion
```

OPCIÓN -G (DEPURACIÓN)

Esta opción estándar indica al compilador que debe generarse información de depuración en la salida de la compilación. Esta información de depuración hace que sea posible que el depurador haga referencia al código fuente y a los nombres de las variables, así como el análisis de un archivo `core` tras abortar un programa. Incluya esta opción cuando se necesite depurar un programa interactivamente o realizar un análisis *post-mortem* de un archivo `core`. Hay que asegurarse de usar esta opción con todos los módulos objetos que vayan a ser inspeccionados por el depurador.

OPCIÓN -D (DEFINE)

Esta opción estándar del compilador de C permite definir un símbolo de macro desde la línea de comandos del compilador. Frecuentemente es utilizada sobre todo desde el archivo `makefile` pero no está limitada a esta práctica. Por ejemplo:

```
$ cc -c -D POSIX_C_SOURCE=199309L hello.c
```

define la macro constante en `_POSIX_C_SOURCE` con el valor `199309L`. Esta definición de macro tiene el efecto de elegir un estándar particular `POSIX` de entre los ficheros incluidos en la compilación. Se pueden definir macros adicionales en la misma línea de órdenes

```
$ cc -c -D_POSIX_C_SOURCE=199309L -DDEBUG hello.c
```

En este ejemplo se han definido dos macros para el archivo `hello.c`, la primera `_POSIX_C_SOURCE`, y a continuación la macro `DEBUG` (sin valor), con el fin de deshabilitar la generación de código en las innovaciones a la macro `assert(3)` dentro del programa.

OPCIÓN -I (INCLUSIÓN)

La opción estándar `-I` permite especificar directorios adicionales para buscar *archivos de inclusión* `include`. Por ejemplo, si se tienen archivos adicionales `include` localizados en un directorio inusual tal como `/usr/local/include`, se podría añadir la opción `-I` como sigue:

```
$ cc -c -I/usr/local/include hello.c
```

Pueden añadirse más de una opción `-I` en la línea de comandos, y los directorios serán recorridos en el orden dado. Por ejemplo, si se ejecuta el comando:

```
$ cc -c -I/usr/local/include -I/opt/include gestion.c
```

Si el programa fuente (`gestion.c`) contiene la directiva `#include "file.h"`, entonces muchos compiladores (no-GNU) de UNIX procesarán la directiva buscando, primero, en el directorio actual, después en todos los directorios dados por la opción `-I` y finalmente en el directorio `/usr/include`. Los mismos compiladores (no-GNU) de UNIX procesarán la directiva de C `#include <file.h>` de la misma forma, excepto que no buscan en el directorio actual. Sin embargo, el compilador de GNU extiende algo la opción `-I` como sigue:

- `-I-`, los directorios que preceden a una opción `-I-` son recorridos solamente para las directivas de la forma `#include "file.h"`.
- Los directorios proporcionados con las opciones `-I` que siguen a una opción `-I-` se recorren para las dos formas `#include "file.h"` y `#include <file.h>`.
- Si no aparece ninguna opción `-I-` en la línea de comandos, entonces el comportamiento es el mismo que para los compiladores no GNU de C.

Un ejemplo de todo esto es el comando de compilación siguiente:

```
$ gcc -c -I/usr/tipodato/include -I- -I/opt/oracle/include convo.c
```

La ejecución del comando del ejemplo permite a la directiva del preprocesador de C `#include "pila.h"` incluir el archivo `/usr/tipodato/include/pila.h`. Esta otra directiva `#include <sqlca.h>`, recorre los directorios que siguen a la opción `-I-`, entonces incluiría al fichero `/opt/oracle/include/sqlca.h`. Esto ocurre porque la forma `<file.h>` no es buscada en los directorios que preceden a la opción `-I-`.

OPCIÓN -E (EXPANDIR)

Esta opción es relativamente estándar entre los compiladores de C de UNIX. Permite modificar la línea de comandos para hacer que el compilador envíe el código preprocesado en C a la salida estándar sin llegar a compilar el código. Esto es útil para controlar las directivas de preprocesamiento y las macros de C. La salida de lo que será compilado puede ser redirigida a otro archivo para que después se examine con un editor.

```
$ cc -c -E hello.c > cpp.out
```

En el ejemplo anterior, la opción `-E` hace que los archivos `include` y el programa sean preprocesados y redirigidos hacia el archivo `cpp.out`. Después, se puede examinar el archivo `cpp.out` con un editor para determinar como será el código final en C. Esto es útil especialmente cuando se trata de depurar el efecto de macros en C que en ocasiones provocan errores de compilación difíciles de diagnosticar.

OPCIÓN -O (OPTIMIZAR)

Esta opción no es estándar entre los compiladores. Algunos compiladores requieren que un argumento siga a la `-O`, otros no y otros aceptarán opcionalmente un argumento. FreeBSD acepta lo siguiente:

- O y -O1 especifican optimización de nivel 1.
- O2 especifica optimización de nivel 2 (optimización mayor).
- O3 especifica optimización de nivel 3 (más que -O2).
- O0 especifica sin optimización.

Para el compilador de GNU, estas opciones pueden estar repetidas, y la última es la que establece el nivel final de optimización. Por ejemplo:

```
$ gcc -c -O3 -O0 ellipse.c
```

compila sin optimizar porque al final aparece `-O0`.

En contraste con el compilador GNU, el compilador de HP soporta las siguientes opciones de para niveles crecientes de optimización:

- Optimización por defecto +O0
- Nivel 1 de optimización +O1
- Nivel 2 de optimización +O2 (equivale a -O, sin argumentos, de FreeBSD)
- Nivel 3 de optimización +O3
- Nivel 4 de optimización +O4

El compilador de IBM AIX 4.3 soporta las opciones `-O`, `-O2` y `-O3` para niveles crecientes de optimización. Todo ello acentúa la necesidad de revisar para cada sistema las opciones del compilador en la página de `cc` del manual correspondiente.

La optimización analiza el código compilado, código objeto, para aumentar la eficiencia en la ejecución de las instrucciones. Cuanto mayor es el nivel de optimización mejor es el código ejecutable producido, por contra, mayor es el tiempo de compilación.

Compilación de programas C en WINDOWS

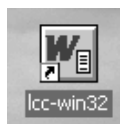
Existen diversos entornos de compilación de programas C para Windows. Uno muy popular, gratuito y de gran fiabilidad, diseñado en la University of Virginia es ***lcc-win32***. Es un entorno de programación completo que permite editar, compilar y ejecutar programas escritos en lenguaje C. El compilador se adapta a las especificaciones ANSI C Y se puede descargar gratuitamente de Internet en la dirección Web:

<http://www.cs.virginia.edu/~lcc-win32>

o bien en

<http://www.q-software-solutions.com>

La instalación de ***lcc-win32*** crea el icono de acceso:



Acceso a ***lcc-win32***

Seleccionando el icono *Wedit* se accede al entorno integrado de desarrollo, típico de sistemas Windows



B.1 Editar un programa

Este entorno de programación permite editar sin necesidad de utilizar otra aplicación. La forma más sencilla y rápida de editar un programa es la siguiente: pulse File en la parte superior del menú principal; a continuación New y en el menú que se despliega File. Escriba el nombre del programa y acepte con el botón Ok:



Ahora comience a teclear las sentencias del programa fuente; una vez terminado guarde el programa fuente mediante *Ctrl+S*.

B.2 Compilación

Seleccione el menú Compiler y la opción Compile. El entorno *lcc* le sugiere unas acciones relativas al proyecto en el cual se va a agrupar el programa; elija *New Project*. El compilador se pone en marcha y en la pantalla inferior se muestran los errores de compilación. En el programa se corrigen los errores; se vuelve a guardar (*Ctrl+S*) y de nuevo se compila. El proceso de depuración se repite hasta que no haya más errores.

B.3 Ejecución

Seleccione el menú Compiler y la opción Execute. La ejecución abre una nueva ventana en la que se muestran los resultados de la ejecución del programa.

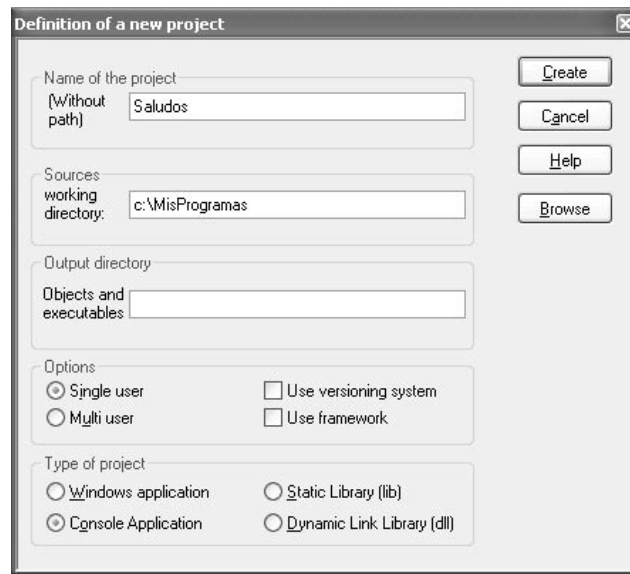
B.4 Crear un proyecto

Las aplicaciones escritas en C se componen de un número determinado de funciones y de una función principal (*main()*). Normalmente, estas funciones se agrupan en dos o más archivos. Cada archivo (*archivo fuente*) se compila y depura de forma individual, o sea se realiza una *compilación independiente*.

Para gestionar la *compilación independiente* se crea un proyecto. En el proyecto se agrupan los distintos archivos que forman la aplicación; cada archivo se compila y depura independientemente. Una vez compilado cada archivo, se enlazan para formar el archivo ejecutable.

Los pasos a seguir para crear un proyecto y ejecutar el programa:

1. Seleccione *Project* en el menú principal y a continuación elija Create.
2. Teclee el nombre del proyecto y el camino (*path*) donde se ubicará.
Proyecto *Saludos*, situado en *C:\MisProgramas*:
Pulse la opción Create.
3. A continuación, pulse No, o bien Next en las nuevas pantallas que le presenta la aplicación *lcc*. Llegará a una pantalla en la cual pulsará End; es la última por la que navegará para crear el *esqueleto* del proyecto.
4. Edite cada archivo: para ello pulse en File, y opción New File. Una vez creado el archivo se debe añadir al proyecto, para ello pulse Project y después *Add/Delete files*.



Archivo entrada.c:

```
#include <stdio.h>
void entradaNombre(char nom[])
{
    printf(" Tu nombre: ?");
    gets(nom);
}
```

5. Compile el archivo. Pulse *Project*, elija *Check syntax*. Si hay errores de compilación, se realizan las correcciones en el archivo: se guarda (*Ctrl+S*) y se vuelve a realizar la compilación con *Check syntax*.

6. Edite cada archivo fuente, repitiendo los pasos 4 y 5.

Archivo progSuerte.c:

```
#include <stdio.h>
void entradaNombre(char nom[]);

void main()
{
    char nombre[41];
    entradaNombre(nombre);
    printf("\n Hola %s, felices vacaciones \n", nombre);
}
```

7. En el menú principal pulse *Project*, elija *Make* para enlazar los archivos que forman el proyecto y crear el archivo ejecutable. Por último, *Execute* ejecutará el programa.

Recursos Web de programación

LIBROS

Existen numerosos libros de C para cualquier nivel de aprendizaje. Recogemos en este apartado aquellos que consideramos más sobresalientes para su formación y posterior vida profesional de programador en C.

American National Standards Institute (ANSI) . *Programming Language C*, ANSI X3.159- 1989. Manual de especificaciones del lenguaje ANSI C, conocido como **C89**

International Organization for Standardization (ISO). ISO/IEC 9899:1990... (**C89**) y ISO/IEC 9899:1999 (**C99**)

Deitel, P. J. y Deitel, H. M. *C: How to Program*. Prentice-Hall, 1994. Excelente libro cuyos autores son muy reconocidos en el mundo editorial tanto anglosajón como español , donde se han traducido muchas de sus obras.

Fischer, Alice E., Eggert, David W., Ross, Stephen M. *Applied C: An Introduction and More*. Boston (USA): McGraw-Hill, 2001. Libro excelente que proporciona una perspectiva teórico-práctica sobresaliente.

Feuer, Alan R. *The C Puzzle Book*. Addison-Wesley, 1998.

Es una excelente elección para todos los programadores que desean ampliar su conocimiento básico del lenguaje de programación C y totalmente compatible con la versión estándar ANSI C. Construido sobre reglas y experiencias prácticas es una obra muy completa para la comprensión de la sintaxis y semántica de C.

Harbison, Samuel P., Tartan Laboratories. *C: A Reference Manual*, 4/e. Prentice Hall, 1995. Este libro contiene en detalle todo el lenguaje de programación en C. La claridad, los útiles ejemplos y discusiones de compatibilidad con C++ lo diferencia, esencialmente, de otras referencias. Existen numerosos compiladores de C en los diferentes sistemas operativos más utilizados: Windows, Unix y Linux. Para los lectores estudiantes una buena opción es el empleo de Linux, un sistema operativo gratuito de gran potencia y con facilidad de “descarga” del sistema y del compilador de la Red.

Horton, Ivor. *Beginning C. Third edition*. New York: Aprress, 2004. Magnífico libro para el aprendizaje del lenguaje C dirigido a principiantes y con un enfoque práctico **Jones, Bradley y Aitken, Peter.** *C in 21 Days*. Sixth Edition. Indianápolis, USA: Sams, 2003. Magnífico y voluminoso libro práctico de programación en C

Joyanes, Luis, Castillo, Andres, Sánchez, Lucas y Zahonero Martínez, Ignacio. *Programación en C. Libro de problemas*. Madrid: McGraw-Hill, 2002. Libro complementario de esta obra, con un fundamento eminentemente teórico-práctico y con gran cantidad de ejemplos, ejercicios y problemas resueltos.

Joyanes Aguilar, Luis, y Zahonero Martínez, Ignacio. *Programación en C*. 2ª edición. McGraw-Hill, 2005. Libro eminentemente didáctico pensado para cursos profesionales o universitarios de programación en C. Complementario en el aspecto teórico de este libro.

Kernighan, Brian y Ritchie, Dennis M. *The C programming Language*. 2/e. Prentice Hall, 1988.

Este libro es la referencia definitiva de los autores del lenguaje. Imprescindible para el conocimiento con profundidad del lenguaje C. Traducido al castellano como *El Lenguaje de Programación C, segunda edición (ANSI-C)*, Prentice-Hall, 1991.

Kernighan, Brian W. y Pike, Rob. *The Unix Programming Environment*. Prentice-Hall, 1984, traducido al español como *El entorno de programación Unix (Prentice-hall, 1987)*. Describe y explica el sistema operativo Unix a nivel de usuario y de programador de aplicaciones no distribuidas (un poco anticuado para las versiones actuales, pero excelente).

Kelley, Al. *A Book on C*. Addison-Wesley, 1997. Libro sencillo para el aprendizaje de C

Koenig, Andrew. *C Traps and Pitfalls*. Addison-Wesley, 1988. Es un magnífico libro para aprender a programar a nivel avanzado en C y C++ tanto para profesionales como para estudiantes.

Oualline, Steve. *Practical C Programming*. O'Reilly & Associates, 1997. Libro muy interesante con una gran cantidad de reglas prácticas y consejos eficientes para progresar adecuadamente en el mundo de la programación.

Plauger, P. J. *C The Standard Library*. Prentice-Hall, 1992

Un excelente manual de referencia de sintaxis del lenguaje _ANSI C. Referencia obligada como elemento de consulta para el programador en su trabajo diario.

Sedgewick, Robert. *Algorithms in C*. Addison-Wesley, 3/e, 1997.

Excelente libro para el conocimiento y aprendizaje del diseño y construcción de algoritmos. Es una obra clásica que el autor ha realizado para otros lenguajes como C++.

Summit, Steve y Lafferty, Deborah. *C Programming Faqs: Frequently Asked Questions*. Addison-Wesley, 1995.

Contiene más de 400 preguntas y dudas frecuentes sobre C junto con las respuestas correspondientes. Aunque este recurso contiene mucha información útil, el libro es más un almacén de preguntas y respuestas que una referencia completa.

Tondo, Clovis L., Gimpel, Scott E., C Programming Kernighan, Brian W. *The C Answer Boock: Solutions to the Exercices in the C Programming Language, Second Edition.*, Prentice-Hall, 1993. Contiene las explicaciones completas de todos los ejercicios de la segunda edición del libro de Kernighan y Ritchie. Es ideal para utilizar en cualquier curso de C. Un estudio cuidadoso de este libro le ayudará a comprender ANSI C y mejorará sus destrezas de programación.

Van Der Linden, Peter. *Expert C Programming*, 1994. En esta obra se recogen todo tipo de reglas y consejos de programación para sacar el mayor rendimiento posible a la programación en C.

SITIOS DE INTERNET

REVISTAS

<i>C/C++ Users Journal</i>	www.cuj.com
<i>Dr. Dobb's Journal</i>	www.ddj.com
<i>MSDN Magazine</i>	msdn.microsoft.com/msdnmag
<i>Sys Admin</i>	www.samag.com
<i>Software Development Magazine</i>	www.sdmagazine.com
<i>UNIX Review</i>	www.review.com
<i>Windows Developer's Journal</i>	www.wdj.com
<i>C++ Report</i>	www.creport.com
<i>Journal Object Oriented Programming</i>	www.joopmag.com

PÁGINAS WEB IMPORTANTES DE C/C++

ccp.servidores.net/cgi-lib/buscador
www.msaj.com/msjquerry.html
Revista Microsoft Systems Journal
www.shareware.com
Software shareware
msdn.microsoft.com/developer
Página oficial de Microsoft sobre Visual C++

www.borland.com

Página oficial del fabricante Inprise/Borland

www.lysator.liu.se/c/

The Development of the C Language

[//en.wikibooks.org/wiki/Programming:C](http://en.wikibooks.org/wiki/Programming:C)

Programaming C en Wikibooks

Historia de C en la enciclopedia Wikipedia (10 páginas excelentes)

[//en.wikipedia.org/wiki/C_programming_language](http://en.wikipedia.org/wiki/C_programming_language)

Página web de Bjarne Stroustrup

<http://www.research.att.com/~bs/C++.html>

Página de Dennis M. Ritchie

www.cs.bell-labs.com/who/dmr/index.html

Preguntas y respuestas frecuentes sobre C (FAQ)

www.faqs.org/faqs/C-faq/faq

www.faqs.org/faqs/C-faq/faq/index.html (de Steve Summit)

TUTORIALES

www.help.com/cat/2/259/hc/index-9.html

www.lysator.liu.se/c

www.anubis.dkung.dk/JTC1/SC22/WG14

www.uib.es/c-calculo/manuals/altrese/cursc.htm

www.help.com/cat/259/hc/index-9.html

PREGUNTAS Y RESPUESTAS FRECUENTES SOBRE C (FAQ)

www.eskimo.com/~scs/C-faq/top.html

www.faqs.org/faqs/C-faq/faq

www.help.com/cat/2/259/hc/index-9.html

www.lysator.liu.se/c

www.anubis.dkung.dk/JTC1/SC22/WG14

www.uib.es/c-calculo/manuals/altrese/cursc.htm

www.help.com/cat/259/hc/index-9.html

<http://www.parashift.com/c++-faq-lite/>

cplusplus.com

<http://www.cplusplus.com/>

C99

www.comeaucomputing.com/techtalk/c99

REVISTAS DE INFORMÁTICA / COMPUTACIÓN DE PROPÓSITO GENERAL Y/O C EN SECCIONES ESPECIALIZADAS DE PROGRAMACIÓN Y EN PARTICULAR DE C/C++

PC Magazine

Linux Magazine

PC World

Java Report

www.ppcmag.com

www.linux-mag.com

www.pcworld.com

www.javareport.com

<i>Sigs</i>	www.sigs.com
<i>Java Pro</i>	www.java-pro.com
<i>PC Actual</i>	www.pc-actual.com
<i>PC World España</i>	www.idg.es/pcworld
<i>Dr.Dobb's (en español)</i>	www.mkm-pi.com
<i>Visual C++ Developer Journal.</i>	www.vcdj.com/

COMPILADORES

Thefreecountry.com
www.thefreecountry.com/compilers/cpp.shtml

Compilador GCC de GNU/Linux (Free Software Foundation)
[//gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/](http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/)

CompiladoresWin32 C/C++ de Willus.com
www.willus.com/ccomp.shtml

Compiladores e interpretes C/C++
www.latindevelopers.com/res/C++/compilers

Compilador Lxx-Win32 C de Jacob Navia
www.cs.virginia.edu/~lcc-win32/

El Rincón del C
www.elrincondec.com/compile

Visual Studio Beta 2005
[//msdn2.microsoft.com/library/default.aspx](http://msdn2.microsoft.com/library/default.aspx)

ORGANIZACIONES INFORMÁTICAS ESPECIALIZADAS EN C/C++

ACCU (Association of C and C++ Users).
www.accu.org/

ANSI(American National Standards Institute).
www.ansi.org

Comité ISO/IEC JTC1/SC22/WG14-C.
anubis.dkuug.dk/JTC1/SC22/WG14/
 Comité encargado de la estandarización y seguimiento del C.

Comité ISO/IEC JTC1/SC22/WG21-C++.
anubis.dkuug.dk/jtc1/sc22/wg21/
 Comité encargado de la estandarización y seguimiento del C++.

ISO (International Organization for Standardization).
www.iso.ch/
 Organización de aprobación de estándares de ámbito internacional (*entre ellos de C/C++*)

ISO/IEC JTC1/SC22/WG14-C
 Grupo de trabajo de estandarización internacional de la versión C99

ESTANDARES DE C

K&R (*The C Programming Language*, 1978)

ANSI C (Comité ANSI X3.159-1989, 1989)

ANSI C (adoptado por ISO como ISO/IEC 9899 :1990, 1990)

C99 (ISO 9899:1999)

MANUALES DE C ESTÁNDAR Y LIBRERIAS DE C

www.dinkunware.com/ (biblioteca de C)

www.open-std.org/jtc1/sc22/wg14/www/c99Rationalev5.10.pdf (Revisión 5.10 de Abril de 2003, manual de más de 200 páginas)

www.sics.se/~pd/ISO-C-FDIS.1999-04.pdf, (manual de más de 500 páginas)

Índice

- A
- Acceso aleatorio, 295-297
 - fseek(), 246
 - ftell(), 246
 - SEEK_SET, 246
 - SEEK_CUR, 246
 - SEEK_END, 246
- almacén libre, 212
- algoritmo, 13
 - análisis, 14
 - características, 13
 - definición, 13
 - de ordenación, 161
 - de búsqueda, 161
 - diseño, 14
- ámbito, 98
 - de programa, 98
 - de una función, 98
 - del archivo fuente, 98
- apuntadores (punteros), 191
- árbol, 369
 - binario, 370
 - borrado, 373
 - de búsqueda, 372, 373
 - de expresión, 371
 - general, 369
 - inserción, 373
 - recorrido, 372
 - terminología, 400
- archivos binarios, 246
 - fwrite(), 244
 - fread(), 244
 - fopen(), 244
 - fclose(), 244
 - fputc(), 244
 - ordenación, 274
 - mezcla directa, 276
 - fusión, 274
 - organización, 274
 - directos (aleatorios), 270
 - registro,s 267
- archivo de cabecera 25
- archivo indexado,s 272
 - función *hash*, 271
 - colisiones, 271
- archivo secuencial, 268
- argc, 248
- argv, 248
- arrays, 137
 - almacenamiento, 138
 - arrays de punteros, 193
 - asignación de memoria, 212
 - cadena de texto, 138
 - como parámetros, 140
 - de caracteres, 138
 - declaración, 137
 - inicialización, 138
 - inicializa,r 138
 - multidimensionales, 139
 - subíndices, 137
 - tamaño, 138
- arrays y punteros, 193
- arreglos (véase arrays)
- ASCII, 225
- aserción, 17
- asignación, 37
 - de cadena,s 226
- asignación dinámica de memoria 211-213
- asociatividad 43
- B
- Biblioteca de ejecución, 10
- Bits, 3
- Bucle, 17, 71
 - bandera, 72
 - condición, 72
 - controlado por centinela, 72
 - controlado por indicador, 72
 - infinito, 72
 - invariante 17,
- buffer de entrada / salida, 243
- búsqueda, 161
 - secuencial, 163
 - binaria, 163
- C
- C, Lenguaje de programación, 10
- C++, 11, 296
- cadena de caracteres, 225
 - inicialización, 226
 - funciones de cadena, 228
 - conversión de cadenas, 229
- punteros a cadenas, 226
- Calidad del software, 17
 - Integridad, 17, 297
 - Robustez, 17, 297
- callo(), 213
- campos de bit, 179
- carácter nulo, 225
- CD-ROM, 3
- char, 225
- cintas, 5
- clase, 297
- código fuente, 8
- código objeto, 9
- cola, 348-350
 - concepto, 348
 - especificación, 349
 - implementación, 349
- compilación, 9
- computadora, 1
 - multimedia, 6
- const puntero, 195
- constantes, 29
 - carácter, 29
 - de cadena, 29
 - declaradas, 30
 - definidas, 29
 - enumeradas, 29
 - literales, 29
 - reales, 20
 - simbólicas, 29
- comentario, 27
- compatibilidad, 17
- corrección, 17
- conversión de cadenas a números
 - atof (), 229
 - atoi(), 229
 - atol (), 229
- cortafuego, 6
- D
- datos tipos TAD, 295
- depuración, 15
- diagrama de flujo, 14
- directivas del preprocesador, 25
- diseño del algoritmo, 14
- divide y vencerás, 14

doble cola, 366
documentación, 15
DVD, 5

E

Editor, 9
ejecución de un programa, 9-10, 14
enlazador, 9
ensamblador, 8
entrada y salida por archivos, 243
 flujos, 243
 entrada estándar, 243
 salida estándar, 243
 apertura de un archivo, 244
 fopen(), 244
 binario, 246
 texto, 244
 FILE, 243
 modo de apertura, 244
 fclose(), 244
 EOF, 244
 funciones de lectura y escritura, 244
 acceso aleatorio, 246
enum, 176
enumeraciones 29, 176
EOF, 244
Errores
 De compilación, 9
 De tiempo de ejecución, 9
Especificación de tipo, 298
expresión condicional `?:`, 41
estructuras
 de control, 55, 47
 selectivas, 55
 repetitivas, 71
estructura (`struct`), 173-175
 miembro, 173
 nombre, 173
 inicialización, 174
 operador punto (`.`), 174
 operador flecha (`->`), 174
 asignación, 174
 estructuras anidadas, 174
 arrays, 157
evaluación en cortocircuito, 40
expresiones, 37
extensibilidad, 17

F

FIFO, 348
flujos de entrada / salida, 243
apertura, 244
flux(), 246
free(), 213
funciones, 95
 aritméticas, 38

concepto, 26, 95
declaración, 96
definidas por el usuario, 26
estructura, 95
parámetros, 97
puntero a función
 prototipos, 96
 en línea, 98
funciones de cadena
 memcpy(), 228
 strcat(), 228
 strchr(), 228
 strcmp(), 228
 stricmp(), 228
 strcpy(), 228
 strncpy(), 228
 strcspn(), 228
 strlen(), 228
 strncat(), 228
 strncmp(), 228
 strnset(), 228
 strpbrk(), 228
 strrchr(), 228
 strspn(), 228
 strevt(), 228
 strstr(), 229
 strtok(), 229
de entrada salida
 printf(), 245
 scanf(), 245
funciones de lectura escritura
de ficheros
 fputc(), 244
 putc(), 244
 getc(), 245
 fgetc(), 244
 fputs(), 244
 gets(), 226
 fprintf(), 245
 fscanf(), 245
 feof(), 245
 rewind(), 245

H

Hardware, 1, 2, 4
Heap, 211
HTML, 6

I

identificador, 27
integridad, 17
interfaz, 3
Internet, 6
intérprete, 9
invariante, 17
IMP, 6

L

lectura de cadenas
 getch(), 226
 getche(), 226
 gets(), 226
 putchar(), 226
 puts(), 226
 scanf, 226
lectura de ficheros
 fread(), 246
lenguaje
 de alto nivel, 7
 de bajo nivel, 7
 interpretado, 9
 máquina, 7
 traductores, 8
LIFO, 347
lista
 circular, 316
 enlazada, 311
 doblemente enlazada, 314
lista circular, 316
 eliminación, 316
 inserción, 316
lista doblemente enlazada, 314
 declaración, 314
 eliminación, 315
 inserción, 315
lista enlazada, 312
 cabecera, 312
 clasificación, 312
 eliminación, 313
 inserción, 313
 operaciones, 312
Literal de cadena, 225

M

macros, 393
main(), 348
mallo(), 212
manual
 de mantenimiento, 15
 de usuario, 15
modelo de memoria, 212
montículo (*heap*), 212

N

NULL, 225

O

operador, 37
 (), 42
 ->, 41
 ., 41

- [], 42
- &, 41
- *, 41
- aritmético, 38
- asignación, 37, 41
- asociatividad, 43
- condicional, 41
- conversión de tipos, 40
- coma, 42
- de bits, 40
- de desplazamiento de bits, 41
- de dirección, 41
- de incremento, 39
- de decremento, 39
- especiales, 40
- lógicos, 40
- prioridad, 43
- relacionales, 39
- sizeof, 42
- ordenación, 161
 - por burbuja, 161
 - por inserción, 163
 - por *shell*, 163
 - por selección, 162
 - rápida (*QuickSort*), 163

P

- P2P, 6
- PDA, 6
- palabras
 - reservadas, 27
 - reservadas ANSI C,
- parámetros, 87
- paso por referencia, 195
- paso por valor, 195
- periférico, 3
- pila
 - cima, 347
 - concepto, 347
 - especificaciones, 348
 - implementación, 348
- POP, 6
- postcondición, 16
- precondición, 16
- preprocesador, 25
- prioridad, 43
- procedimientos, 95
- programa, 14
 - codificación, 14
 - compilación, 14
 - depuración, 14
 - documentación, 42
 - ejecución, 14
 - elementos, 27
 - mantenimiento, 14
 - verificación, 15
- programación, 16

- estructurada, 16
- teorema, 16
- pseudocódigo, 5
- puntero a archivo, 244
- punteros (apuntadores)
 - aritmética de punteros, 194
 - arrays, 193
 - arrays de punteros concepto, 191
 - declaración, 191
 - inicialización estática, 192
 - indirección, 192
 - nulos NULL, 6, 192
 - operador &, 192
 - operador*, 192
 - puntero a estructura, 193
 - puntero a función, 196
 - puntero constante, 195
 - puntero doble, 192
 - punteros de cadenas, 227
 - paso por referencia, 195
 - void, 192
 - y arrays, 195

R

- realloc(), 213
- redes, 5
 - WAN, 5, 6
 - LAN, 5
- reutilización, 17
- robustez, 16
- recursividad, 123
 - funciones, 123
 - infinita, 123
 - versus iteración, 124

S

- segmento de código, 212
- segmento de datos, 212
- sentencias, 55
 - break, 57
 - continue,
 - do while, 74
 - for, 73
 - if anidadas, 56
 - if con dos alternativas, 56
 - if con una alternativo, 55
 - nula, 55
 - return, 95
 - switch, 57
 - while, 57
- sistema operativo, 6
- sizeof, 42
- SMTP, 6
- software, 1, 2, 6
- stack (pila), 347

- static, 100
- stderr, 243
- stdin, 243
- stdio.h, 243
- stdout, 243
- stream, 243
- string.h, 227
- struct, 173, 174
 - miembro, 174
 - nombre, 174
 - inicialización, 174
 - operador punto (.), 174
 - operador flecha (->), 174
 - asignación, 174
 - arrays, 174

T

- tabla, 137
- tipos
 - abstractos TAD, 295
- enumerados, 176
 - definidos por usuario, 177
 - void, 27
- transportabilidad, 16
- typedef, 177

U

- UAL, 2, 4
- UCP, 2, 4
- UC, 2, 4
- unión (union), 176
 - definición, 176
 - miembros, 176
 - memoria, 177
 - acceso 177

V

- variable, 30
 - automáticas, 99
 - declaración, 30
 - definición dinámicas estáticas, 100
 - externas, 99
 - globales, 99
 - nombres, 30
 - puntero, 191
 - registro, 100
- verificación, 15
- void, 27

W

- WWW, 6

¡Estudia a tu propio ritmo y aprueba tu examen con Schaum!

Los Schaum son la herramienta esencial para la preparación de tus exámenes.

Cada Schaum incluye:

- Teoría de la asignatura con definiciones, principios y teoremas claves.
- Problemas resueltos y totalmente explicados, en grado creciente de dificultad.
- Problemas propuestos con sus respuestas.

Hay un mundo de Schaum a tu alcance...¡BUSCA TU COLOR!

Info
Informática

Eco
Economía &
Empresa

Mat
Matemáticas

Inge
Ingeniería

Cien
Ciencias

www.mhe.es/joyanes

www.mcgraw-hill.es

The McGraw-Hill Companies