

¡¡APRUEBA TU EXAMEN CON SCHAUM!!

Programación en C++ Un enfoque práctico

Schaum

Luis Joyanes Aguilar • Lucas Sánchez García

-  REDUCE TU TIEMPO DE ESTUDIO
-  MÉTODO DE APRENDIZAJE PROGRESIVO PARA PROGRAMAR EN C++
-  EJERCICIOS RESUELTOS COMENTADOS CON CÓDIGO DISPONIBLE EN WEB

Utilízalo para las siguientes asignaturas:

METODOLOGÍA DE LA PROGRAMACIÓN

FUNDAMENTOS DE PROGRAMACIÓN



PROGRAMACIÓN I Y II

INTRODUCCIÓN A LA PROGRAMACIÓN

PROGRAMACIÓN ORIENTADA A OBJETOS

ESTRUCTURAS DE DATOS I

Programación en C++
Un enfoque práctico

Serie Shaum

Programación en C++

Un enfoque práctico

Serie Schaum

**LUIS JOYANES AGUILAR
LUCAS SÁNCHEZ GARCÍA**

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.

Programación en C++. Un enfoque práctico. 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.



DERECHOS RESERVADOS © 2006, respecto a la primera edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1.^a planta
Basauri, 17
28023 Aravaca (Madrid)

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

ISBN: 84-481-4643-3
Depósito legal: M.

Editor: Carmelo Sánchez González
Compuesto en Puntographic, S. L.
Impreso en

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prólogo	XIII
Capítulo 1. Programación orientada a objetos versus programación estructurada: C++ y algoritmos.....	
Introducción	1
1.1. Concepto de algoritmo	1
1.2. Programación estructurada.....	3
1.3. El paradigma de orientación a objetos	4
1.3.1. Propiedades fundamentales de la orientación a objetos	5
1.3.2. Abstracción.....	5
1.3.3. Encapsulación y ocultación de datos.....	6
1.3.4. Generalización y Especialización.....	6
1.3.5. Polimorfismo	7
1.4. C++ Como lenguaje de programación orientada a objetos	7
Ejercicios	8
Solución de los ejercicios	8
Ejercicios propuestos	14
Capítulo 2. Conceptos básicos de los programas en C++	
Introducción	15
2.1. Estructura general de un programa en C++	15
2.1.1. Directivas del preprocesador	16
2.1.2. Declaraciones globales	16
2.1.3. Función <code>main()</code>	17
2.1.4. Funciones definidas por el usuario	18
2.1.5. Comentarios.....	18
2.2. Los elementos de un programa en C++	19
2.2.1. Identificador.....	19
2.2.2. Palabras reservadas.....	19
2.2.3. Comentarios.....	19
2.2.4. Signos de puntuación y separadores	20
2.2.5. Archivos de cabecera	20
2.3. Tipos de datos en C++	20
2.3.1. Enteros (<code>int</code>).....	20
2.3.2. Tipos en coma flotante (<code>float/double</code>)	21
2.3.3. Caracteres (<code>char</code>).....	21
2.3.4. El tipo de dato <code>bool</code>	23
2.4. Constantes	23
2.4.1. Constantes literales	23
2.5. Variables	25
2.6. Duración de una variable	25
2.6.1. Variables locales	25
2.6.2. Variables globales	26
2.6.4. Variables dinámicas y de objetos	26

2.7. Entradas y salidas	26
2.7.1. Salida (<code>cout</code>)	27
2.7.2. Entrada (<code>cin</code>).....	27
2.8. Espacios de nombres (<code>namespaces</code>).....	28
Ejercicios.....	29
Solución de los ejercicios.....	31
Ejercicios propuestos	34
 Capítulo 3. Operadores y expresiones	 35
Introducción	35
3.1. Operadores y expresiones.....	35
3.2. Operador de asignación	35
3.3. Operadores aritméticos	36
3.4. Operadores de incrementación y decrementación	39
3.5. Operadores relacionales.....	39
3.6. Operadores lógicos	40
3.7. Operadores de manipulación de bits	41
3.7.1. Operadores de desplazamiento de bits (<code>>></code> , <code><<</code>).....	41
3.7.2. Operadores de asignación adicionales	42
3.7.3. Operadores de direcciones	42
3.8. Operador condicional <code>?</code>	42
3.9. Operador coma ,	43
3.10. Operadores especiales (<code>()</code> , <code>[]</code> y <code>::</code>)	44
3.11. El operador <code>sizeof</code>	44
3.12. Conversiones de tipos	45
3.13. Prioridad y asociatividad	45
Ejercicios.....	46
Problemas.....	47
Solución de los ejercicios.....	48
Solución de los problemas	49
Ejercicios propuestos	55
 Capítulo 4. Estructuras de control selectivas (<code>if</code>, <code>if-else</code>, <code>switch</code>).....	 57
Introducción	57
4.1. Estructuras de control.....	57
4.2. La sentencia <code>if</code>	57
4.3. Sentencia <code>if</code> de dos alternativas: <code>if-else</code>	59
4.4. Sentencias <code>if-else</code> anidadas.....	61
4.5. Sentencia de control <code>switch</code>	62
4.6. Expresiones condicionales: el operador <code>?:</code>	63
4.7. Evaluación en cortocircuito de expresiones lógicas	64
Ejercicios.....	65
Problemas.....	66
Solución de los ejercicios.....	67
Solución de los problemas	69
Ejercicios propuestos	75
Problemas propuestos.....	76
 Capítulo 5. Estructuras de control repetitivas (<code>while</code>, <code>for</code>, <code>do-while</code>).....	 77
Introducción	77
5.1. La sentencia <code>while</code>	77
5.2. Repetición: el bucle <code>for</code>	81
5.4. Repetición: el bucle <code>do...while</code>	83
5.5. Comparación de bucles <code>while</code> , <code>for</code> y <code>do-while</code>	84

5.6. Bucles anidados.....	86
Ejercicios.....	86
Problemas	88
Solución de los ejercicios.....	89
Solución de los problemas	91
Ejercicios propuestos	100
Problemas propuestos.....	101
 Capítulo 6. Funciones y módulos	103
Introducción	103
6.1. Concepto de función.....	103
6.2. Estructura de una función	104
6.3. Prototipos de las funciones	106
6.4. Parámetros de una función	108
6.5. Funciones en línea (<i>inline</i>)	112
6.6. Ámbito (alcance)	114
6.7. Clases de almacenamiento.....	116
6.8. Concepto y uso de funciones de biblioteca	119
6.9. Miscelánea de funciones	119
6.10. Sobrecarga de funciones (polimorfismo)	121
6.11. Plantillas de funciones	123
Problemas	124
Ejercicios	124
Solución de los ejercicios.....	126
Solución de los problemas	127
Problemas propuestos.....	133
Ejercicios propuestos	133
 Capítulo 7. Arrays (arreglos, listas o tablas)	135
Introducción	135
7.1. Arrays (arreglos)	135
7.2. Inicialización de arrays	136
7.3. Arrays multidimensionales.....	138
7.4. Utilización de arrays como parámetros	141
Ejercicios	147
Problemas	148
Solución de los ejercicios.....	149
Solución de los problemas	151
Ejercicios propuestos	157
Problemas propuestos.....	158
 Capítulo 8. Registros (estructuras y uniones)	161
Introducción	161
8.1. Estructuras	161
8.2. Uniones.....	168
8.3. Enumeraciones	170
8.4. Sinónimo de un tipo de dato <code>typedef</code>	172
8.5. Campos de bit.....	172
Ejercicios	174
Problemas	175
Solución de los ejercicios.....	176
Solución de los problemas	178
Problemas propuestos.....	181
Ejercicios propuestos	181

Capítulo 9. Cadenas.....	183
Introducción	183
9.1. Concepto de cadena	183
9.2. Lectura de cadenas	185
9.3. La biblioteca string.h.....	189
9.4. Conversión de cadenas a números	191
Ejercicios.....	193
Problemas	194
Solución de los ejercicios.....	195
Solución de los problemas	198
Ejercicios propuestos	201
Problemas propuestos.....	202
 Capítulo 10. Punteros (apuntadores)	203
Introducción	203
10.1. Concepto de puntero (apuntador)	203
10.2. Punteros <i>NULL</i> y <i>void</i>.....	206
10.3. Punteros y arrays	206
10.4. Punteros de cadenas.....	208
10.5. Aritmética de punteros	208
10.6. Punteros como argumentos de funciones	212
10.7. Punteros a funciones.....	213
10.8. Punteros a estructuras	216
Ejercicios.....	217
Solución de los ejercicios.....	219
Problemas	219
Solución de los problemas	222
Ejercicios propuestos	225
Problemas propuestos.....	226
 Capítulo 11. Gestión dinámica de la memoria	227
Introducción	227
11.1. Gestión dinámica de la memoria	227
11.2. El operador new	228
11.3. El operador delete	233
Ejercicios.....	235
Problemas	236
Solución de los ejercicios.....	237
Solución de los problemas	239
Ejercicios propuestos	242
Problemas propuestos.....	243
 Capítulo 12. Ordenación y búsqueda.....	245
Introducción	245
12.1. Búsqueda	245
12.1.1. Búsqueda secuencial	245
12.1.2. Búsqueda binaria.....	247
12.2. Ordenación.....	248
12.3. Ordenación por burbuja.....	248
12.4. Ordenación por selección	250
12.5. Ordenación por inserción	251
12.6. Ordenación Shell	253
Problemas	254

Ejercicios	254
Solución de los ejercicios	255
Solución de los problemas	256
Ejercicios propuestos	257
Problemas propuestos.....	258
Capítulo 13. Clases y objetos	259
Introducción	259
13.1. Clases y objetos	259
13.2. Definición de una clase	259
13.3. Objetos.....	261
13.4. Constructores	264
13.5. Destructores.....	265
13.6. Clase compuesta	268
13.7. Errores de programación	269
Ejercicios.....	270
Problemas	271
Solución de los ejercicios.....	272
Solución de los problemas	273
Ejercicios propuestos	274
Problemas propuestos.....	275
Capítulo 14. Herencia y polimorfismo	277
Introducción	277
14.1. Herencia.....	277
14.1.1. Tipos de herencia	280
14.1.2. Clases abstractas.....	286
14.2. Ligadura.....	287
14.3. Funciones virtuales	287
14.4. Polimorfismo	287
Ejercicios.....	288
Problemas	289
Solución de los ejercicios.....	290
Solución de los problemas	290
Problemas propuestos.....	292
Capítulo 15. Plantillas, excepciones y sobrecarga de operadores	293
Introducción	293
15.1. Generecidad	293
15.2. Excepciones	296
15.2.1. Lanzamiento de excepciones	296
15.2.2. Manejadores de excepciones.....	297
15.2.3. Diseño de excepciones	297
15.3. Sobre carga	299
15.3.1. Sobre carga de operadores unitarios	300
15.3.2. Conversión de datos y operadores de conversión forzada de tipos	301
15.3.3. Sobre carga de operadores binarios	303
Problemas	306
Ejercicios	306
Solución de los ejercicios.....	307
Solución de los problemas	308
Problemas propuestos.....	310
Ejercicios propuestos	310

Capítulo 16. Flujos y archivos	311
Introducción	311
16.1. Flujos (<i>streams</i>)	311
16.2. Las clases <i>istream</i> y <i>ostream</i>	312
16.3. Las clases <i>ifstream</i> y <i>ofstream</i>	316
16.3.1. Apertura de un archivo	316
16.3.2. Funciones de lectura y escritura en ficheros	317
Solución de los ejercicios	320
Problemas	320
Ejercicios	320
Solución de los problemas	321
Problemas propuestos	322
Capítulo 17. Listas enlazadas	323
Introducción	323
17.1. Fundamentos teóricos	323
17.2. Clasificación de las listas enlazadas	324
17.3. Operaciones en listas enlazadas	325
17.3.1. Declaración de los tipos nodo y puntero a nodo y clase nodo	325
17.3.2. Declaración de clase lista, inicialización o creación	327
17.3.3. Insertar elementos en una lista	328
17.3.4. Buscar elementos de una lista	330
17.3.5. Eliminar elementos de una lista	331
17.4. Lista doblemente enlazada	332
17.4.1. Inserción de un elemento en una lista doblemente enlazada	334
17.4.2. Eliminación de un elemento en una lista doblemente enlazada	335
17.5. Listas circulares	336
Problemas	338
Ejercicios	338
Solución de los ejercicios	339
Solución de los problemas	342
Ejercicios propuestos	344
Problemas propuestos	345
Capítulo 18. Pilas y colas	347
Introducción	347
18.1. Concepto de pila	347
18.2. Concepto de cola	353
Ejercicios	359
Problemas	360
Solución de los ejercicios	361
Solución de los problemas	362
Problemas propuestos	364
Ejercicios propuestos	364
Capítulo 19. Recursividad	367
Introducción	367
19.1. La naturaleza de la recursividad	367
19.2. Funciones recursivas	368
19.3. Recursión versus iteración	370
19.4. Recursión infinita	370
Ejercicios	371
Problemas	372
Solución de los ejercicios	373

Solución de los problemas	374
Ejercicios propuestos	377
Problemas propuestos.....	377
Capítulo 20. Árboles	379
Introducción	379
20.1. Árboles generales	379
20.2. Árboles binarios.....	380
20.3. Estructura y representación de un árbol binario	381
20.5. Recorridos de un árbol	383
20.6. Árbol binario de búsqueda	384
20.7. Operaciones en árboles binarios de búsqueda	385
Ejercicios	388
Problemas	389
Solución de los ejercicios.....	390
Solución de los problemas	392
Ejercicios propuestos	394
Problemas propuestos.....	395
Índice	396

Prólogo

C++ es heredero directo del lenguaje C que a su vez se deriva del lenguaje B. El lenguaje de programación C fue desarrollado por **Dennis Ritchie** de AT&T Bell Laboratories y se utilizó para escribir y mantener el sistema operativo UNIX. C es un lenguaje de propósito general que se puede utilizar para escribir cualquier tipo de programa, pero su éxito y popularidad está especialmente relacionado con el sistema operativo UNIX. Los sistemas operativos son los programas que gestionan (administran) los *recursos de la computadora*. Ejemplos bien conocidos de sistemas operativos además de UNIX son MS/DOS, OS Mac, OS/2, MVS, Linux, Windows 95/98, NT, 2000, XP 2000, o el recientemente presentado Vista de Microsoft que vendrá a sustituir al actual Windows XP.

La especificación formal del lenguaje C es un documento escrito por Ritchie, titulado *The C Reference Manual*. En 1997, **Ritchie** y **Brian Kernighan**, ampliaron ese documento y publicaron un libro referencia del lenguaje *The C Programming Language* (también conocido por el K&R). Aunque C es un lenguaje muy potente y sigue siendo muy utilizado en el mundo universitario y también en el profesional, el hecho de haber sido diseñado al principio de los setenta y que la naturaleza de la programación ha cambiado radicalmente en la década de los ochenta y de los noventa, exigía una actualización para subsanar sus “deficiencias”.

Bjarne Stroustrup de AT&T Bell Laboratories desarrolló C++ al principio de la década de los ochenta. Stroustrup diseñó C++ como un mejor C. En general, C estándar es un subconjunto de C++ y la mayoría de los programas C son también programas C++ (la afirmación inversa no es verdadera). C++ además de añadir propiedades a C, presenta características y propiedades de *programación orientada a objetos*, que es una técnica de programación muy potente y que se verá en la segunda parte de este libro.

Se han presentado varias versiones de C++ y su evolución ha incorporado nuevas y potentes propiedades: herencia, genericidad, plantillas, funciones virtuales, excepciones, etc. C++ ha ido evolucionando año a año y como su autor ha explicado: «evolucionó siempre para resolver problemas encontrados por los usuarios y como consecuencia de conversaciones entre el autor, sus amigos y sus colegas»¹.

C++ comenzó su proyecto de estandarización ante el comité ANSI y su primera referencia es *The Annotated C++ Reference Manual* [Ellis 89]². En diciembre 1989 se reunió el comité X3J16 del ANSI por iniciativa de Hewlett Packard. En junio de 1991, la estandarización de ANSI pasó a formar parte de un esfuerzo de estandarización ISO. En 1995 se publicó un borrador de estándar para su examen y en 1998 se aprobó el estándar C++ internacional por las organizaciones ANSI e ISO, conocido como **ISO/IEC 14882** o simplemente **C++ Estándar** o **ANSI C++**. Stroustrup publicó en 1997 la tercera edición de su libro *The C++ Programming Language* y, en 1998, una actualización que se publicó como *edición especial*³. El libro que tiene en sus manos, sigue el estándar ANSI/ISO C++.

C++ en el aprendizaje de algoritmos y programación orientada a objetos

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, así como un curso de *programación orientada a objetos*.

¹ [Stroustrup 98], p. 12

² Existe versión española de Addison-Wesley/Díaz de Santos y traducida por los profesores Manuel Katrib y Luis J. Yanes.

³ Esta obra fue traducida por un equipo de profesores universitarios que dirigió y coordinó el profesor Luis J. Yanes, coautor de esta obra.

Los doce primeros capítulos constituyen un curso inicial de algoritmos y programación y puede ser utilizado 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. Los restantes capítulos (12 al 20) pueden ser utilizados para cursos de introducción a estructuras de datos y a programación orientada a objetos y puede ser utilizado en asignaturas tales como *Estructuras de datos I*, *Introducción a la programación orientada a objetos* o similares. C++ es uno de los lenguajes universales 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 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 la colección los dos 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, encontrará el lector todos los códigos fuente incluidos y no incluidos en la obra y que podrá descargar de Internet. Pretendemos no sólo 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?

Programación en C++. Un enfoque práctico, está pensado y diseñado para enseñar métodos de escritura de programas en C++, tanto estructurados como orientados a objetos, útiles y eficientes y como indica el subtítulo de la obra, de un modo totalmente práctico. Se pretende también enseñar tanto la sintaxis como el funcionamiento del lenguaje de programación C++ junto con las técnicas de programación y los fundamentos de construcción de algoritmos básicos, junto el diseño y construcción de clases y restantes propiedades fundamentales de la programación orientada a objetos. El contenido se ha escrito pensando en un lector que tiene conocimientos básicos de algoritmos y de programación en C/C++, y que desea aprender a conocer de modo práctico técnicas de programación, tanto estructuradas como orientadas a objetos. La gran profusión de ejemplos resueltos permitirá al lector sin conocimientos de programación básica y del lenguaje C/C++ seguir el curso incluido en el libro, aunque en este caso le recomendamos no afronte la resolución de los ejercicios y problemas hasta tanto no haya estudiado y asimilado bien los conceptos teóricos y prácticos de cada capítulo.

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, es conveniente utilizar en paralelo con el aprendizaje una computadora que tenga instalados un compilador de C++ y un editor de texto para preparar sus archivos de código fuente. Normalmente, hoy día, la mayoría de los compiladores vienen con un Entorno Integrado de Desarrollo (EID) que contiene un compilador, un editor y un depurador de puesta a punto de sus programas. 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), **ANSI C++**, 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 el estándar citado. Los autores hemos utilizado el compilador **Dev C++** por considerar que es uno de los más utilizados hoy día, tanto por alumnos autodidactas como por alumnos de carreras de ciencias y de ingeniería, en sus prácticas regladas o en sus hogares, al ser un compilador gratuito, de libre disposición, que funciona en entornos Windows y totalmente compatible con ANSI C++.

Aunque el libro está concebido como un libro de problemas, se ha incluido la teoría necesaria para que el libro pueda ser considerado también un libro de teoría con los conocimientos mínimos necesarios para conseguir alcanzar un nivel medio de programación. No obstante si necesita o desea adquirir un nivel más profundo de teoría de programación en C++, tanto de algoritmos, como de programación estructurada u orientada a objetos, le proponemos un libro complementario de éste, y de uno de sus coautores (Luis J. Yanes) *Programación en C++: Algoritmos, estructura de datos y objetos*, 1.^a edición, publicado en el año 2000. En la actualidad el autor trabaja en la 2.^a edición que espera estar publicada en el primer semestre de 2006. Esta nueva edición tendrá, además de las actualizaciones correspondientes, el enunciado de todos los ejercicios y problemas, resueltos y propuestos en esta obra de la *Colección Schaum* de modo que ambos libros se configuran como un conjunto complementario teórico-práctico para el aprendizaje de programación en C++.

En 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 a medida que vaya leyendo el libro, compile, ejecute y depure de errores sus programas, tanto los resueltos y propuestos como los que usted diseñe, tratando de entender la lógica del algoritmo y la sintaxis del lenguaje en cada ejercicio que realice.

Compiladores y compilación de programas en C++

Existen numerosos compiladores de C++ en el mercado. Desde ediciones gratuitas y *descargables* a través de Internet hasta profesionales, con costes diferentes, comercializados por diferentes fabricantes. Es difícil dar una recomendación al lector porque casi todos ellos son buenos compiladores, muchos de ellos con Entornos Integrados de Desarrollo (**EID**). Si usted es estudiante, tal vez la mejor decisión sea utilizar el compilador que le haya propuesto su profesor y que utilice en su Universidad, Instituto Tecnológico o cualquier otro Centro de Formación, donde estudie. Si usted es un lector autodidacta y está aprendiendo por su cuenta, existen varias versiones gratuitas que puede descargar desde Internet. Algunos de los más reconocidos son: **Dev-C++** de **Bloodshed** que cumple fielmente el estándar ANSI/ISO C++ (utilizado por los autores del libro para editar y compilar todos los programas incluidos en el mismo) y que corre bajo entornos Windows; **GCC** de GNU que corre bajo los entornos Linux y Unix. Existe muchos otros compiladores gratuitos por lo que tiene donde elegir. Tal vez un consejo más: procure que sea compatible con el estándar ANSI/ISO C++.

Bjarne Stroustrup (creador e inventor de C++) en su página oficial⁴ ha publicado el 9 de febrero de 2006, “una lista incompleta de compiladores de C++”, que le recomiendo lea y visite. Por su interés incluimos, a continuación, un breve extracto de su lista recomendada:

Compiladores gratuitos

- Apple C++
- Borland C++
- Dev-C++ de Bloodshed
- GNUIntel C++ para Linux
- Microsoft Visual C++ Toolkit 2003
- Sun Studio...

⁴ <http://public.research.att.com/~bs/compilers.html>. El artículo “An incomplete list of C++ compilers” lo suele modificar Stroustrup y en la cabecera indica la fecha de modificación. En nuestro caso, consultamos dicha página mientras escribíamos el prólogo en la primera quincena de marzo, y la fecha incluida es la de 9 de febrero de 2006.

Compiladores comerciales

Borland C++
Compaq C++
HP C++
IBM C++
Intel C++ para Windows, Linux y algunos sistemas empotrados
Microsoft C++

Stroustrup recomienda un sitio de compiladores gratuitos de C y C++ (Compilers.net).

NOTA PRÁCTICA DE COMPATIBILIDAD C++ ESTÁNDAR

En el artículo antes citado de Stroustrup, recomienda que compile con su compilador el siguiente simple programa fuente C++. Si usted compila bien este programa, no tendrá problemas con C++ estándar. En caso contrario, aconseja buscar otro compilador que sea compatible.

```
#include<iostream>
#include<string>

using namespace std;

int main( )
{
    string s;
    cout << "Por favor introduzca su nombre seguido por ''Intro''\n";
    return 0; // esta sentencia return no es necesaria
}
```

¿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; enunciados de *Ejercicios y Problemas*, y a continuación, la *Solución de los ejercicios* y la *Solución de los problemas*, donde se incluyen el análisis del problema y la codificación en C++; por último, todos los capítulos contienen una colección de ejercicios y problemas propuestos, cuyo objetivo es facilitar al lector la medición de su aprendizaje.

Con el objetivo de facilitar la edición, ejecución y puesta a punto de los programas al lector, esta edición incluye una novedad: todos los códigos fuente del libro, tanto de los numerosos ejemplos como de ejercicios y problemas se han incluido en la página web oficial del libro (www.mhe.es/joyanes). Por esta razón, numerosos listados de códigos fuente de los ejercicios y problemas resueltos de diferentes capítulos, sólo se han incluido en la página Web, y no se han editado en el libro. Se pretende con esta estructura dos cosas: primera, que el libro no resultara voluminoso en número de páginas; segunda, que el lector pudiera comprobar cuando considerara oportuno la solución propuesta, sin más que bajar dicho código de la página web.

Capítulo 1. Programación orientada a objetos versus programación estructurada: C++ y algoritmos. Explica y describe los conceptos fundamentales de algoritmos, de programación estructurada y de programación orientada a objetos, junto con las propiedades más significativas de C++.

Capítulo 2. Conceptos básicos de los programas en C++. Se introduce al lector en la estructura general y elementos básicos de un programa en C++. Se describen los tipos de datos, constantes, variables y la entrada/salida; así como el nuevo concepto de espacio de nombres (*namespaces*).

Capítulo 3. 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 con versiones de tipos, junto con reglas de prioridad y *asociatividad* de los operadores en las expresiones y operaciones matemáticas.

Capítulo 4. Estructuras de control selectivas (if, if-else, 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 5. Estructuras de control repetitivas (while, for, do-while). 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 6. Funciones y módulos. Examina las funciones en C++, una parte importante de la programación y enseña la 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. Se introduce al lector en el concepto de funciones de biblioteca, junto con la propiedad de sobrecarga y plantillas de funciones.

Capítulo 7. Arrays o arreglos (listas y tablas). Se explica un método sencillo pero potente de almacenamiento de datos. Se aprende cómo agrupar datos similares en arrays o “arreglos” (listas y tablas) numéricas.

Capítulo 8. Registros (estructuras y uniones). Se describen conceptos básicos de estructuras, uniones y enumeraciones: declaración, definición, iniciación, uso y tamaño. El concepto de tipo de dato definido por el usuario (`typedef`) y de campos de bit, se explican también en este capítulo.

Capítulo 9. 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 10. Punteros (apuntadores). Presenta una de las características más potentes y eficientes del lenguaje C++, los punteros. Se describe el concepto, los punteros NULL y void y los diferentes punteros de cadena, como argumentos de funciones, punteros a funciones y punteros a estructuras, junto con la aritmética de punteros.

Capítulo 11. Gestión dinámica de la memoria. Se describe la gestión dinámica de la memoria y los operadores asociados a esta tarea: `new` y `delete`. Se proporcionan reglas de funcionamiento de estos operadores y reglas para asignación de memoria.

Capítulo 12. Ordenación y búsqueda. Se describen en el capítulo métodos básicos de búsqueda de información (secuencial y binaria) y de ordenación de listas y vectores (burbuja, selección, inserción y shell).

Capítulo 13. Clases y objetos. Los conceptos fundamentales de la programación orientada a objetos se estudian en este capítulo: clases, objetos, constructores, destructores y clases compuestas.

Capítulo 14. Herencia y polimorfismo. Las propiedades de herencia y polimorfismo constituyen, junto con las clases, la espina dorsal de la programación orientada a objetos.

Capítulo 15. Plantillas, excepciones y sobrecarga de operadores. La propiedad de genericidad se implementa mediante plantillas (*templates*). Las excepciones son un mecanismo muy potente para programación avanzada en C++ y en otros lenguajes. Por último se describe la propiedad ya estudiada de sobrecarga; en este caso, la sobrecarga de operadores.

Capítulo 16. Flujos y archivos. Las estructuras de datos básicas de almacenamiento en dispositivos externos, flujos y archivos, son objeto de estudio en este capítulo. Las clases `istream`, `ostream`, `ifstream` y `ofstream`, se analizan y describen prácticamente.

Capítulo 17. 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 variía 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 18. 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 19. Recursividad. La naturaleza de la recursividad y su comparación con la iteración se muestran en el capítulo. Esta propiedad es una de las más potentes existentes en programación. Su comprensión y buen uso es muy importante para cualquier programador.

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 EN LA WEB

En el sitio oficial del libro, www.mhe.es/joyanes, podrá encontrar los siguientes apéndices que podrá descargar libremente:

- A. C++ frente a (*versus*) C
- B. Guía de sintaxis de ANSI/ISO C++
- C. Operadores y expresiones (prioridad)
- D. Código de caracteres ASCII
- E. Palabras reservadas ANSI/ISO C++
- F. Biblioteca estándar de funciones ANSI/ISO C++
- G. Biblioteca estándar de clases ANSI/ISO C++
- H. Glosario
- I. Recursos (Libros/Revistas/Tutoriales) de programación en la Web
- J. Bibliografía

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 en C++ sea todo lo rápida y eficiente que deseamos. Así mismo deseamos destacar de modo muy especial, a todos nuestros lectores, profesores, maestros y alumnos de Latinoamérica, que utilizan nuestros libros y nos ayudan con sus comentarios continuos para mejorar cada una de nuestras obras y ediciones. 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 total, si así lo consideran oportuno.

En Carchejo (Jaén) y en Madrid, marzo de 2006.

LOS AUTORES

CAPÍTULO 1

Programación orientada a objetos *versus* programación estructurada: C++ y algoritmos

Introducción

El aprendizaje de la programación requiere el conocimiento de técnicas y metodologías de *programación estructurada*. Aunque a finales del siglo XX y, sobre todo en este siglo XXI, la *programación orientada a objetos* se ha convertido en la tecnología de software más utilizada; el conocimiento profundo de algoritmos y estructuras de datos, en muchos casos con el enfoque estructurado, facultará al lector y futuro programador los fundamentos técnicos necesarios para convertirse en un brillante programador de C++, en general, y programador orientado a objetos, en particular.

1.1. Concepto de algoritmo

Un **algoritmo** es una secuencia finita de instrucciones, reglas o pasos que describen de modo preciso las operaciones que una computadora debe realizar para ejecutar una tarea determinada en un tiempo fijo [Knuth 68]¹. En la práctica, un algoritmo es un método para resolver problemas mediante los pasos o etapas siguientes:

1. *Diseño del algoritmo* que describe la secuencia ordenada de pasos —sin ambigüedades— conducentes a la solución de un problema dado (*Análisis del problema y desarrollo del algoritmo*).
2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación*).
3. *Ejecución y validación* del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo indicando *cómo* hace el algoritmo la tarea solicitada, y eso se traduce en la construcción de un algoritmo. 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.

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

- **Diagrama de flujo (flowchart).** Representación gráfica de un algoritmo.
- **Pseudocódigo.** Lenguaje de especificación de algoritmos, mediante palabras similares al inglés o español.

¹ Donald E. Knuth (1968): *The art of Computer Programming*, vol. 1, 1.^a ed., 1968; 2.^a ed. 1997, Addison Wesley. Knuth, es considerado uno de los padres de la algoritmia y de la programación. Su trilogía sobre “Programación de computadoras” es referencia obligada en todo el mundo docente e investigador de Informática y Computación.

El **algoritmo** es la especificación concisa del método para resolver un problema con indicación de las acciones a realizar. Un algoritmo es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un determinado problema. Es, por tanto, *un método para resolver un problema que tiene en general una entrada y una salida*. Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar bien *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada, Proceso y Salida*.

EJEMPLO 1.1. *Se desea diseñar un algoritmo para conocer si un número es primo o no.*

Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo: 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etc.

Entrada: dato n entero positivo

Salida: es o no primo.

Proceso:

1. Inicio.
2. Poner x igual a 2 ($x = 2$, x variable que representa a los divisores del número que se busca n).
3. Dividir n por x (n/x).
4. Si el resultado de n/x es entero, entonces n es un número primo y bifurcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a x ($x \leftarrow x + 1$).
6. Si x es igual a n, entonces n es un número primo; en caso contrario, bifurcar al punto 3.
7. Fin.

El algoritmo anterior escrito en pseudocódigo es:

```

algoritmo primo
1. inicio
    variables
        entero: n, x:
        lógico: primo;
2. leer(n);
    x←2;
    primo←verdadero;
3. mientras primo y (x < n) hacer
4.     si n mod x != 0 entonces
5.         x← x+1
6.     sino
7.         primo ←falso
8.     fin si
9. fin mientras
10. si (primo) entonces
11.     escribe('es primo')
12. sino
13.     escribe('no es primo')
14. fin si
7. fin

```

1.2. Programación estructurada

La programación estructurada consiste en escribir un programa de acuerdo con unas reglas y un conjunto de técnicas. Las reglas son: el programa tiene un diseño modular, los módulos son diseñados descentralmente, cada módulo de programa se codifica usando tres estructuras de control (*secuencia, selección e iteración*); es el conjunto de técnicas que han de incorporar: recursos abstractos; diseño descentralizado y estructuras básicas de control.

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.

El diseño *descendente* se encarga de resolver un problema realizando una descomposición en otros más sencillos mediante módulos jerárquicos. 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.

Las *estructuras básicas 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.

La programación estructurada significa:

- El programa completo tiene un diseño modular.
- Los módulos se diseñan con metodología descendente (puede hacerse también ascendente).
- Cada módulo se codifica utilizando las tres estructuras de control básicas: secuenciales, selectivas y repetitivas (ausencia total de sentencias **ir → a (goto)**).
- *Estructuración y modularidad* son conceptos complementarios (se solapan).

EJEMPLO 1.2. *Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.*

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, el algoritmo en forma descriptiva sería:

```

inicio
  1. Inicializar contador de números C y variable suma S a cero (S←0, C←1).
  2. Leer un número en la variable N (leer(N))
  3. Si el número leído es cero: (si (N =0) entonces)
    3.1. Si se ha leído algún número (Si C>0)
      • calcular la media; (media← S/C)
      • imprimir la media; (Escribe(media))
    3.2. si no se ha leído ningún número (Si C=0))
      • escribir no hay datos.
    3.3. fin del proceso.
  4. Si el numero leído no es cero : (Si (N <> 0) entonces
    • calcular la suma; (S← S+N)
    • incrementar en uno el contador de números; (C←C+1)
    • ir al paso 2.
Fin

```

algoritmo escrito en pseudocódigo:

```

algoritmo media
inicio
variables
  entero: n, c, s;
  real: media;
  C← 0;
  S←0;

```

```

repetir
    leer(N)
    Si N <> 0 Entonces
        S←S+N;
        C←C+1;
    fin si
    hasta N=0
    si C>0 entonces
        media ← S/C
        escribe(media)
    sino
        escribe('no datos')
    fin si
fin

```

Un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. Lenguajes de programación como C, Pascal, FORTRAN, y otros similares, se conocen como *lenguajes procedimentales* (por procedimientos). Es decir, cada sentencia o instrucción indica al compilador que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En el caso de pequeños programas, estos principios de organización (denominados *paradigma*) se demuestran eficientes. El programador sólo ha de crear esta lista de instrucciones en un lenguaje de programación com-pilar en la computadora y ésta, a su vez, ejecuta las instrucciones.

Cuando los programas se vuelven más grandes, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Para resolver este problema los programas se descomponen en unidades más pequeñas que adoptan el nombre de *funciones* (*procedimientos, subprogramas o subrutinas* en otros lenguajes de programación). De este modo un programa orientado a procedimientos se divide en funciones, cada una de las cuales tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper un programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas *módulos* (normalmente, en el caso de C++, denominadas **archivos o ficheros**); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son éstas. Primera, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedural proporcionan un modelo pobre del mundo real. La **programación orientada a objetos** se desarrolló para tratar de paliar diversas limitaciones que se encontraban en anteriores enfoques de programación.

1.3. El paradigma de orientación a objetos

La programación orientada a objetos aporta un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación procedural que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque procedural de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema. Los lenguajes orientados combinan en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama **objeto**. Si se desea modificar los datos de un objeto, hay que realizarlo mediante las funciones miembro del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa.

En el paradigma orientado a objetos, el programa se organiza como un conjunto formado por objetos que contienen datos y operaciones (*funciones miembro en C++*) que llaman a esos datos y que se comunican entre sí mediante mensajes.

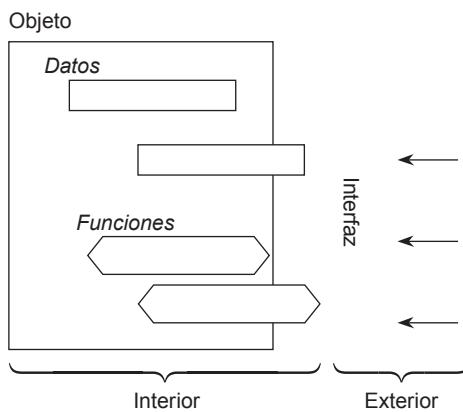


Figura 1.1. Diagrama de un objeto

Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

1.3.1. PROPIEDADES FUNDAMENTALES DE LA ORIENTACIÓN A OBJETOS

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar, no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Estas propiedades son:

- Abstracción (tipos abstractos de datos y clases).
- Encapsulado de datos.
- Ocultación de datos.
- Herencia.
- Polimorfismo.

C++ soporta todas las características anteriores que definen la orientación a objetos.

1.3.2. ABSTRACCIÓN

La **abstracción** es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo. Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos es de interés no sólo *cómo* están organizados sino también *qué* se puede hacer con ellos; es decir, las operaciones de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (**TAD**). Un **tipo abstracto de datos** describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

EJEMPLO 1.3. Diferentes modelos de abstracción del término coche (carro).

- Un coche (carro) es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etc.
- Un coche (carro) es un concepto común para diferentes tipos de coches. Pueden clasificarse, por el nombre del fabricante (Audi, BMW, SEAT, Toyota, Chrysler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido...).

La abstracción *coche* se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un carro (coche) se utilizará para transportar personas o ir de Carchelejo a Cazorla, o de Paradinas a Mazarambroz.

1.3.3. ENCAPSULACIÓN Y OCULTACIÓN DE DATOS

El *encapsulado* o *encapsulación de datos* es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de que los objetos posean las mismas características y comportamiento se agrupan en clases (unidades o módulos de programación que encapsulan datos y operaciones).

La ocultación de datos permite separar el aspecto de un componente, definido por su *interfaz* con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (*information hiding*) y encapsulación de datos (*data encapsulation*) se suelen utilizar como sinónimos, pero no siempre es así y muy al contrario son términos similares pero distintos. En C++ no es lo mismo, ya que los datos internos están protegidos del exterior y no se pueden acceder a ellos más que desde su propio interior y, por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

1.3.4. GENERALIZACIÓN Y ESPECIALIZACIÓN

La *generalización* es la propiedad que permite compartir información entre dos entidades evitando la redundancia. En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina *generalización*.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etc., son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es una **subclase** de la clase **Electrodoméstico** y a su vez **Electrodoméstico** es una **superclase** de todas las otras clases (máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas...). El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina *especialización*.

En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina **herencia**. La herencia permite definir nuevas clases a partir de otras clases ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales.

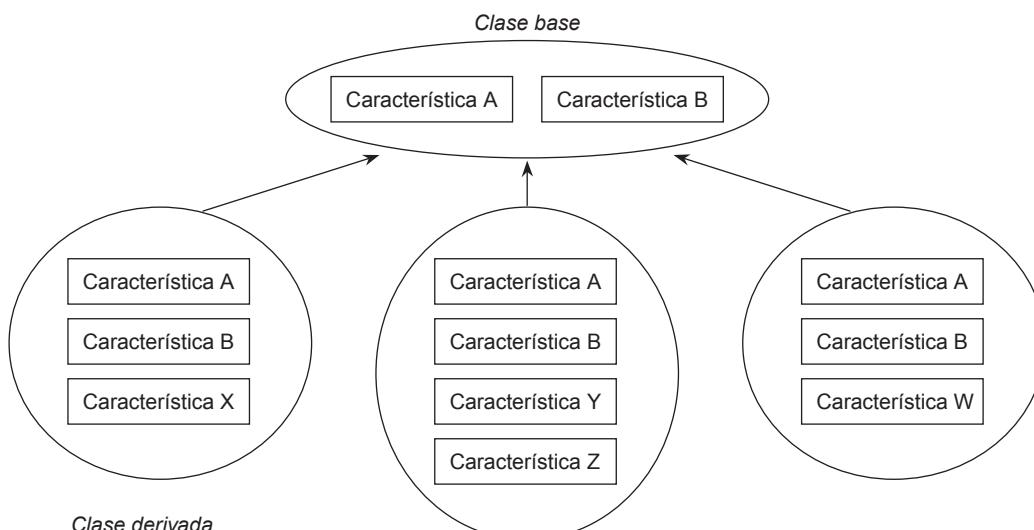


Figura 1.2. Jerarquía de clases.

Una clase *hereda* sus características (datos y funciones) de otra clase.

1.3.5. POLIMORFISMO

La propiedad de **polimorfismo** es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación *abrir* se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro, etc. En cada caso se ejecuta una operación diferente aunque tiene el mismo nombre en todos ellos “*abrir*”. El polimorfismo es la propiedad de una operación de ser interpretada sólo por el objeto al que pertenece.

1.4. C++ Como lenguaje de programación orientada a objetos

C++ es una extensión (ampliación) de C con características más potentes. Estrictamente hablando es un superconjunto de C. Los elementos más importantes añadidos a C por C++ son: clases, objetos y programación orientada a objetos (C++ fue llamado originalmente “*C con clases*”).

En C++, un **objeto** es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil... Una **clase** puede describir las propiedades genéricas de un ejecutivo de una empresa (nombre, título, salario, cargo...) mientras que un objeto representará a un ejecutivo específico (Luis Mackoy, Lucas Soblechero).

En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos. En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, estas variables se llaman *instancias*. Las instancias son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Una **clase** es una descripción general de un conjunto de objetos similares. Por definición todos los objetos de una clase comparten los mismos atributos (*datos*) y las mismas operaciones (*métodos*). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente los lenguajes de programación orientada a objetos facilitan la tarea ya que incorporan clases existentes en su propia programación. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa; por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etc. Las definiciones de clases, incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea. A continuación se prosigue el diseño de un programa utilizando objetos de las clases.

Los términos *objeto* e *instancia* se utilizan frecuentemente como sinónimos (especialmente en C++). Si una variable de tipo *Carro* se declara, se crea un objeto *Carro* (una instancia de la clase *Carro*). Las operaciones definidas en los objetos se llaman *métodos*. Cada operación llamada por un objeto se interpreta como un *mensaje* al objeto, que utiliza un método específico para procesar la operación.

Desde el punto de vista de implementación un **objeto** es una entidad que posee un conjunto de *datos* y un conjunto de *operaciones* (*funciones* o *métodos*). El *estado* de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los **datos** se denominan también *atributos* y componen la estructura del objeto y las **operaciones** —también llamadas *métodos*— representan los servicios que proporciona el objeto.

Nuestro objetivo es ayudarle a escribir programas orientado a objetos tan pronto como sea posible. Sin embargo, como ya se ha observado, muchas características de C++ se han heredado de C, de modo que, aunque la estructura global de un programa pueda ser orientado objetos, consideraremos que usted necesita conocimientos profundos del “viejo estilo *procedimental*”. Por ello, los capítulos de la primera parte del libro le van introduciendo lenta y pausadamente en las potentes propiedades orientadas a objetos de las últimas partes, al objeto de conseguir a la terminación del libro el dominio de la programación en C++.

NOTA: Compiladores y compilación de programas en C++

Existen numerosos compiladores de C++ en el mercado. Desde ediciones gratuitas y *descargables* a través de Internet hasta profesionales, con costes diferentes, comercializados por diferentes fabricantes. Es difícil dar una recomendación al lector porque casi todos ellos son buenos compiladores, muchos de ellos con Entornos Integrados de Desarrollo (EID). Si usted es estudiante, tal vez la mejor decisión sea utilizar el compilador que le haya propuesto su profesor y que utilice en su Universidad, Instituto Tecnológico o cualquier otro Centro de Formación, donde estudie. Si usted es un lector autodidacta y está aprendiendo por su cuenta existen varias versiones gratuitas que puede descargar desde Internet. Algunos de los más reconocidos son: **Dev-C++** de **Bloodshed** que cumple fielmente el estándar ANSI/ISO C++ (utilizado por los autores del libro para editar y compilar todos los programas incluidos en el mismo) y que corre bajo entornos Windows; **GCC** de GNU que corre bajo los entornos Linux y Unix. Existen muchos otros compiladores gratuitos por lo que tiene donde elegir. Tal vez un consejo más: procure que sea compatible con el estándar ANSI/ISO C++.

Bjarne Stroustrup (creador inventor de C++) en su página oficial² ha publicado el 9 de febrero de 2006, “una lista in-

² <http://public.research.att.com/~bs/compilers.html>. El artículo “An incomplete list of C++ compilers” lo suele modificar Stroustrup y en la cabecera indica la fecha de modificación. En nuestro caso, consultamos dicha página mientras escribímos el prólogo en la primera quincena de Marzo, y la fecha incluida es la de 9 de febrero de 2006.

completa de compiladores de C++", que le recomiendo lea y visite. Por su interés incluimos, a continuación, un breve extracto de su lista recomendada:

Compiladores gratuitos

Apple C++
Borland C++
Dev-C++ de Bloodshed
GNUIntel C++ para Linux
Microsoft Visual C++ Toolkit 2003
Sun Studio...

Compiladores comerciales

Borland C++
Compaq C++
HP C++
IBM C++
Intel C++ para Windows, Linux y algunos sistemas empotrados
Microsoft C++

Stroustrup recomienda un sitio de compiladores gratuitos de C y C++ (Compilers.net).

EJERCICIOS

- 1.1. Escribir un algoritmo que lea tres números y si el primero es positivo calcule el producto de los tres números, y en otro caso calcule la suma.
- 1.2. Escribir un algoritmo que lea el radio de un círculo, y calcule su perímetro y su área.
- 1.3. Escribir un algoritmo que lea cuatro números enteros del teclado y calcule su suma.
- 1.4. Escribir un algoritmo que lea un número entero positivo n , y sume los n primeros número naturales.
- 1.5. Escribir un algoritmo que lea un número $n > 0$ del teclado y sume la serie siguiente:

$$\sum_{i=1}^n a_i \quad \text{si} \quad a_i = \begin{cases} i*i & \text{si} & i \text{ es impar} \\ 2 & \text{si} & i \text{ es par} \end{cases} \quad \text{y} \quad n=5$$

- 1.6. Definir una jerarquía de clases para: animal, insecto, mamíferos, pájaros, persona hombre y mujer. Realizar un definición en pseudocódigo de las clases.

SOLUCIÓN DE LOS EJERCICIOS

- 1.1. 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 se calcula o bien el producto o bien la suma.

Entrada Numero1,Numero2 y Numero 3

Salida la suma o el producto

```

incio
1 leer los tres número Numero1, Numero2, Numero3
2 si el Numero1 es positivo
    calcular el producto de los tres numeros
    escribir el producto
3 si el Numero1 es no positivo
    calcular la suma de los tres número
    escribir la suma
fin

```

El algoritmo en pseudocódigo es:

```

algoritmo Producto_Suma
variables

```

```

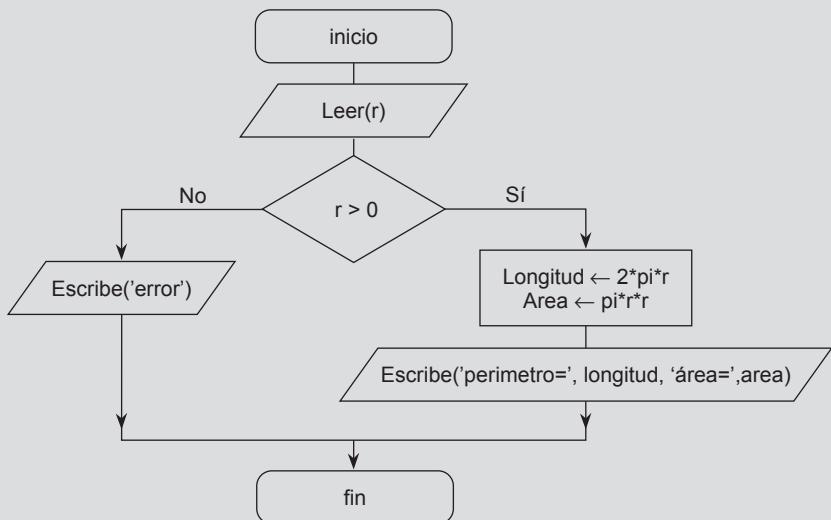
entero: Numero1, Numero2, Numero3, Producto, Suma
inicio
    leer(Numero1, Numero2, Numero3)
    si (Numero1 > 0) entonces
        Producto ← Numero1 * Numero2 * Numero3
        Escribe('El producto de los números es', Producto)
    sino
        Suma ← Numero1 + Numero2 + Numero3
        Escribe('La suma de los números es', Suma)
    fin si
fin

```

- 1.2.** Se declaran las variables reales r , longitud y área, así como la constante π

variables
 real r , longitud, área

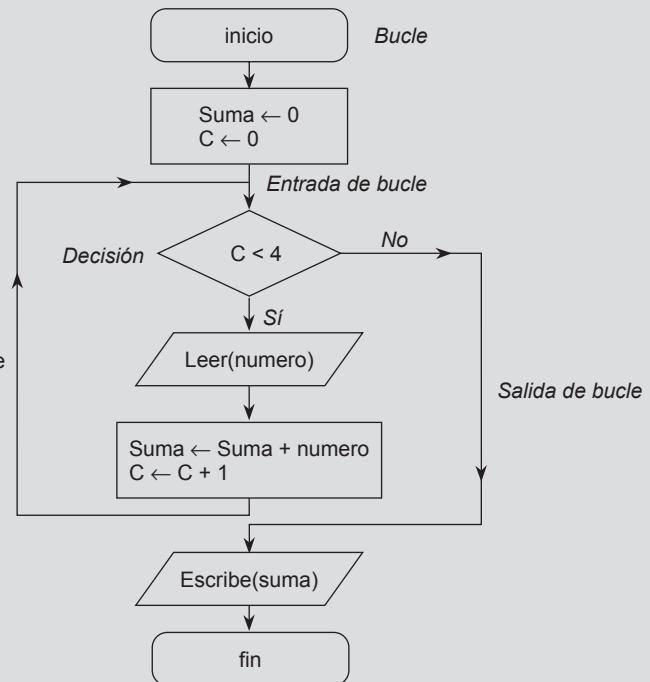
Diagrama de flujo



- 1.3.** Un bucle es un segmento de programa cuyas instrucciones se repiten un número determinado de veces hasta que se cumple una determinada condición. Tiene las siguientes partes: entrada, salida, cuerpo del bucle y decisión que forma parte del cuerpo del bucle o bien de la entrada/salida. El algoritmo se diseña con un bucle, un contador entero C , un acumulador entero $Suma$ y una variable numero para leer los datos.

variables
 entero Suma, C, numero

Cuerpo de bucle



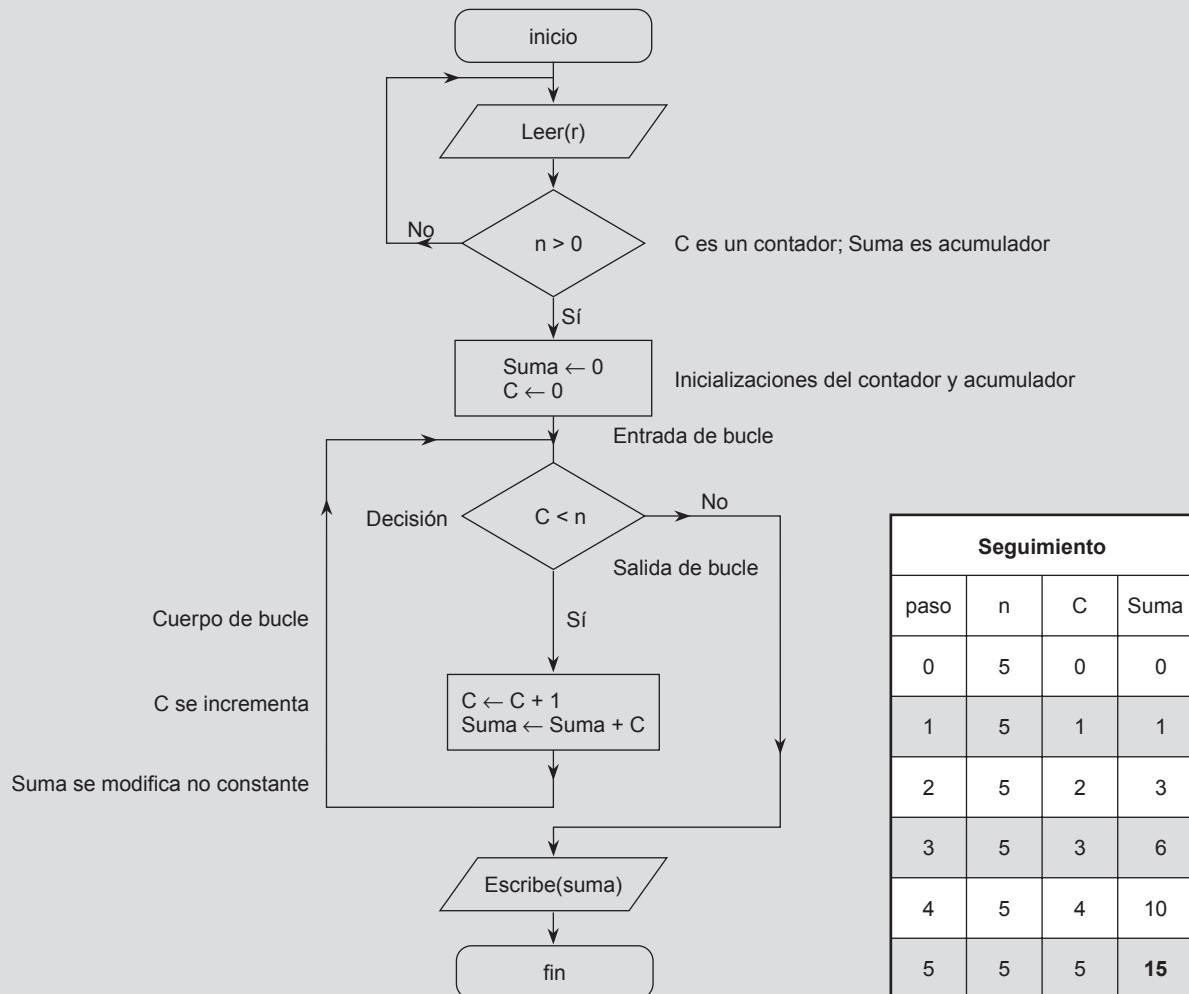
```

algoritmo suma_4
inicio
variables
    entero: Suma, C, numero;
    C← 0;
    suma← 0;
mientras C< 4 hacer
    Leer(Numero)
    Suma← Suma + numero;
    C←C+1;
fin mientras
escribe(suma)
fin

```

- 1.4.** Inicialmente se asegura la lectura del número natural n positivo. Mediante un contador C se cuentan los números naturales que se suman, y en el acumulador $Suma$ se van obteniendo las sumas parciales. Además del diagrama de flujo se realiza un seguimiento para el caso de la entrada $n = 5$

variables Entero: n, Suma, C



```

algoritmo suma_n_naturales
inicio
variables
entero: Suma, C, n;
repetir
    Leer(n)
    hasta n>0
    C ← 0;
    suma← 0;
    mientras C< n Hacer
        C←C + 1;
        suma ← Suma + C;
    fin mientras
    escribe(Suma)
fin

```

- 1.5.** La estructura del algoritmo es muy parecida a la del ejercicio anterior, salvo que ahora en lugar de sumar el valor de una constante se suma el valor de una variable . Se usa un interruptor sw que puede tomar los valores verdadero o falso y que se encarga de decidir si la suma es el valor de i^2 o el valor de 2. Observar que en este ejercicio el contador i se inicializa a 1 por lo que el incremento de i se realiza al final del cuerpo del bucle, y la salida del bucle se realiza cuando $i > n$.

Pseudocódigo

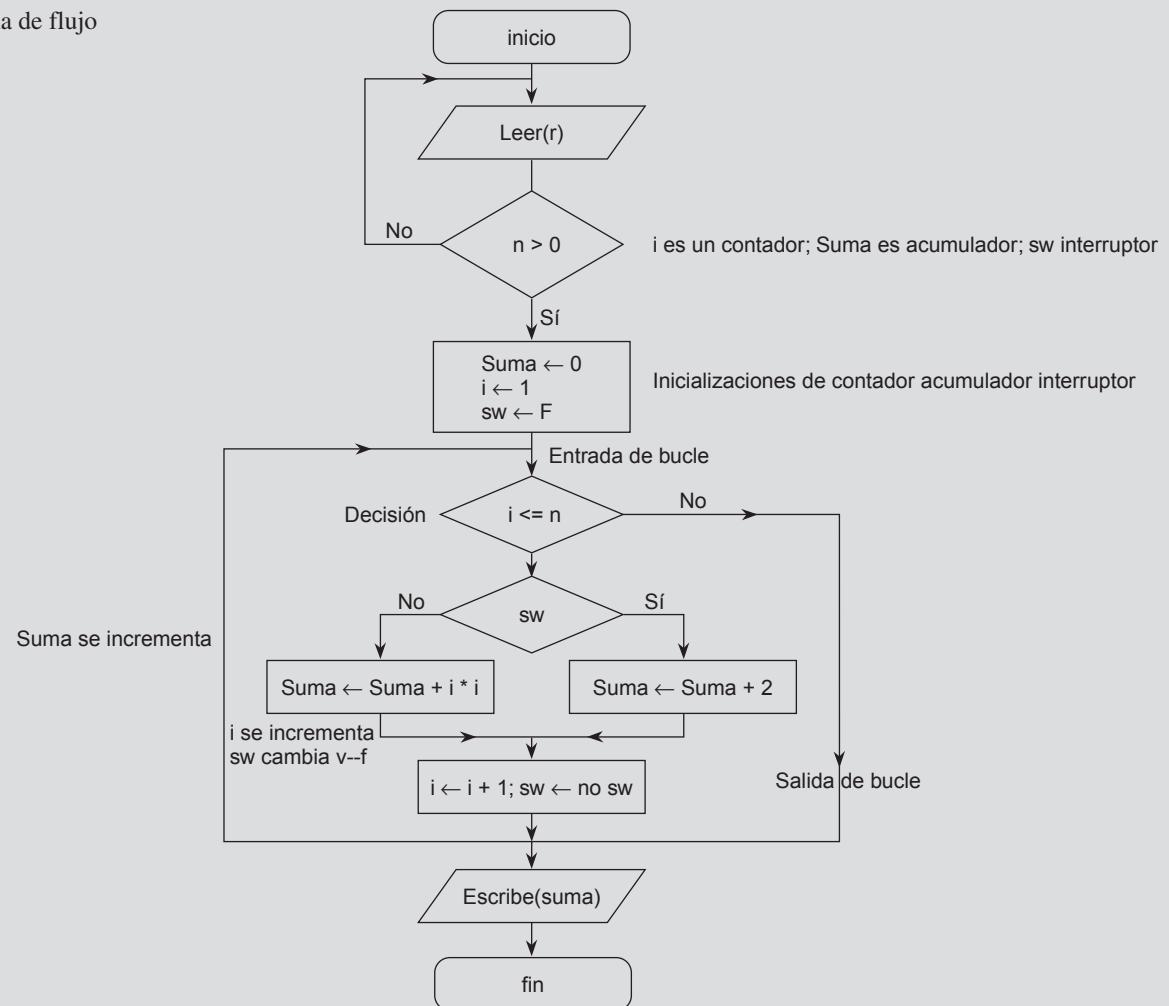
```

algoritmo suma_serie
inicio
variables
entero: Suma, i, n;
logico sw;
repetir
    Leer(n);
    hasta n>0
    i ← 1;
    suma← 0;
    Sw←falso
    mientras i <= n Hacer
        si (sw) Entonces
            Suma ← Suma + 2;
        sino
            Suma ← Suma + i*i
        fin si
        i←i+1
        sw ← no sw
    fin mientras
    escribe(Suma)
fin

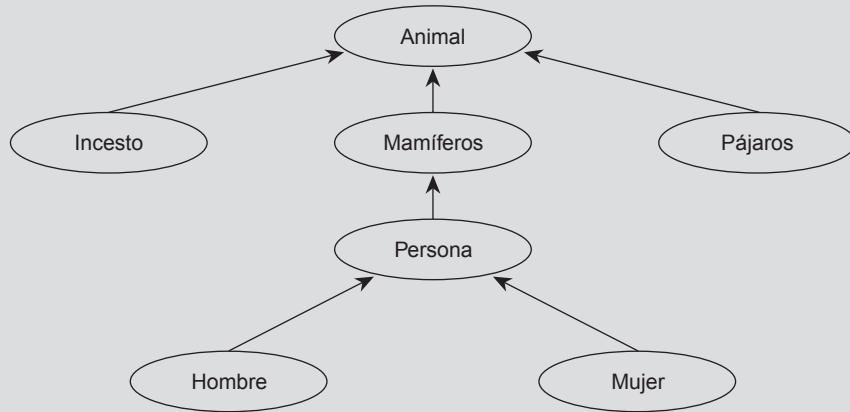
```

Seguimiento n=5				
paso	n	sw	i	suma
0	5	F	1	0
1	5	V	2	1
2	5	F	3	3
3	5	V	4	13
4	5	F	5	15
5	5	V	6	40

Diagrama de flujo



- 1.6. Las clases de objeto mamífero, pájaro e insecto se definen como subclases de animal; la clase de objeto persona, como una subclase de mamífero, y un hombre y una mujer son subclases de persona.



Las definiciones de clases para esta jerarquía puede tomar la siguiente estructura:

```
clase criatura
    atributos (propiedades)
        string: tipo;
        real: peso;
        (...algun tipo de habitat...):habitat;
    operaciones
        crear()→ criatura;
        predadores(criatura)→ fijar(criatura);
        esperanza_vida(criatura) → entero;
    ...
fin criatura.

clase mamifero hereda criatura;
    atributos (propiedades)
        real: periodo_gestacion;
    operaciones
    ...
fin mamifero.

clase persona hereda mamifero;
    atributos (propiedades)
        string: apellidos, nombre;
        date: fecha_nacimiento;
        pais: origen;
fin persona.

clase hombre hereda persona;
    atributos (propiedades)
        mujer: esposa;
    ...
operaciones
    ...
fin hombre.

clase mujer hereda persona;
    propiedades
        esposo: hombre;
        string: nombre;
    ...
fin mujer.
```

EJERCICIOS PROPUESTOS

- 1.1. 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.
- 1.2. Diseñar un algoritmo que imprima y sume la serie de números 4, 8, 12, 16,..., 400.
- 1.3. Escribir un algoritmo que lea cuatro números y a continuación imprima el mayor de los cuatro.
- 1.4. Diseñar un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1.500 metros. La entrada (tiempo del corredor) consistirá en parejas de números (minutos, segundos); por cada corredor , el algo-

- ritmo debe imprimir el tiempo en minutos y se gundos así como la velocidad media. Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.
- 1.5.** Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.
- 1.6.** Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, “*Mortimer*” contiene dos “*m*”, una “*o*”, dos “*r*”, una “*y*”, una “*t*” y una “*e*”.
- 1.7.** Dibujar un diagrama jerárquico de objetos que represente la estructura de un coche (carro).
- 1.8.** Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo Figura geométrica.
- 1.9.** Escribir un algoritmo para determinar el máximo común divisor de dos números enteros (MCD) por el algoritmo de Euclides:
- Dividir el mayor de los dos enteros por el más pequeño.
 - A continuación, dividir el divisor por el resto.
 - Continuar el proceso de dividir el último divisor por el último resto hasta que la división sea exacta.
 - El último divisor es el MCD.

NOTA PRÁCTICA SOBRE COMPILACIÓN DE PROGRAMAS FUENTE CON EL COMPILADOR DEV C++ Y OTROS COMPILADORES COMPATIBLES ANSI/ISO C++

Todos los programas del libro han sido compilados y ejecutados en el compilador de *freeware* (de libre distribución) BloodShed Dev C++³, versión 4.9.9.2. Dev C++ es un editor de múltiples ventanas integrado con un compilador de fácil uso que permite la compilación, enlace y ejecución de programas C o C++.

Las sentencias `system("PAUSE");` y `return EXIT_SUCCESS;` se incluyen por defecto por el entorno **Dev** en todos los programas escritos en C++. El libro incluye ambas sentencias en prácticamente todos los códigos de programa. Por comodidad para el lector que compila los programas se han dejado estas sentencias. Sin embargo, si usted no utiliza el compilador Dev C++ o no desea que en sus listados aparezcan estas sentencias, puede sustituirlas bien por otras equivalentes o bien quitar las dos y sustituirlas por `return 0;` como recomienda Stroustrup para terminar sus programas.

`system("PAUSE");` detiene la ejecución del programa hasta que se pulsa una tecla. Las dos sentencias siguientes de C++ producen el mismo efecto:

```
cout << "Presione ENTER para terminar";
cin.get();
```

`return EXIT_SUCCESS;` devuelve el estado de terminación correcta del programa al sistema operativo. La siguiente sentencia de C++ estándar, produce el mismo efecto:

```
return 0;
```

Tabla 1.1. Dev C++ versus otros compiladores de C++ (en compilación)

Sustitución de sentencias	
DEV C++	Otros compiladores de C++
<code>system("PAUSE");</code>	<code>cout << "Presione ENTER para terminar"; cin.get();</code>
<code>return EXIT_SUCCESS;</code>	<code>return 0;</code>

³ <http://www.bloodshed.net/>

CAPÍTULO 2

Conceptos básicos de los programas en C++

Introducción

Este capítulo comienza con un repaso de los conceptos teóricos y prácticos relativos a la estructura de un programa en C++ dada su gran importancia en el desarrollo de aplicaciones; se describen también los 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 y espacios de nombres.

2.1. Estructura general de un programa en C++

Un programa en C++ se compone de una o más funciones, de las cuales una debe ser obligatoriamente `main()`. Una función en C++ es un grupo de instrucciones que realizan una o más acciones. Un programa contiene una serie de directivas `#include` que permiten incluir en el mismo, archivos de cabecera que a su vez contienen funciones y datos predefinidos en ellos.

EJEMPLO 2.1. Estructura de un programa C++.

```
#include <iostream.h>
int main( )
{
...
}
int func1 ( )
{
...
}
int func2 ( )
{
...
}
```

archivo de cabecera `iostream.h` (o simplemente, `<iostream>`)

cabecera de función
nombre de la función
sentencias de la función

Un programa C++ puede incluir: directivas de preprocesador; declaraciones globales; la función `main()`; funciones definidas por el usuario; comentarios del programa.

EJEMPLO 2.2. Primer programa en C++.

```
#include <iostream>
using namespace std;
// Este programa visualiza Bienvenido a la programacion en C++
// Pronto se van a escribir programas en C++
int main()
{
    cout << "Bienvenido a la programacion en C++\n";
    cout << "Pronto se van a escribir programas en C++\n";
    system("PAUSE"); //si no compila con Dev C++ sustituya esta linea
    return EXIT_SUCCESS; //y la siguiente por la sentencia
} //cout << "Presione ENTER (INTRO) para terminar"; cin.get(); return 0
```

La directiva `#include <iostream>` se refiere a la inclusión del archivo externo denominado `iostream.h` en el que se declaran las funciones y datos que contiene `iostream`, entre otros, la información relativa al objeto `cout`. `using namespace std` sirve para usar las declaraciones estándar del espacio nombrado (consola). Las dobles barras inclinadas `(//)` sirven para indicar comentarios que son ignorados por el compilador. La sentencia `main()` es obligatoria en cada programa C++, e indica el comienzo del programa. Las llaves `{ ... }` encierran el cuerpo de la función `main()` y son necesarias en todos los programas C++.

```
cout << "Bienvenido a la programacion en C++\n";
cout << "Pronto se van a escribir programas en C++\n";
```

Las sentencias anteriores envían los mensajes a la salida estándar mediante el objeto `cout`. El símbolo `<<` se denomina *operador de salida* u *operador de inserción*. Inserta el mensaje en el flujo de salida. El símbolo `'\n'` es el símbolo de *nueva línea*. La sentencia `system("PAUSE")` obliga a que se visualice el resultado de la ejecución hasta que se pulse una tecla. La sentencia `return EXIT_SUCCESS`, retorna el control al sistema operativo, una vez terminada la ejecución.

2.1.1. DIRECTIVAS DEL PREPROCESADOR

El *preprocesador* en un programa C++ 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. Estos archivos se denominan *archivos de cabecera* o *archivos de inclusión*. Estas instrucciones son de la forma `#include<nombreach.h>` o bien `#include "nombreach.h"`. El archivo de cabecera más frecuente es `iostream.h`, ya que proporciona al compilador C++ la información necesaria sobre las funciones de biblioteca `cin` y `cout`, así como otras rutinas de biblioteca que realizan operaciones de entrada y salida.

La directiva `#define` indica al preprocesador que define un *item* 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.

2.1.2. 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. Un prototipo es una declaración de una función y consiste en una definición de la función sin cuerpo y terminado con un punto y coma.

EJEMPLO 2.3. Una estructura modelo que incluye declaraciones globales y prototipos de funciones.

```
#include <iostream.h>
//Definir macros
#define pi 3.14 15
//Declaraciones globales
```

```

float radio, altura;
//prototipo de función

float area_circulo( float radio);
float volumen_cilindor( float radio, float altura);

int main()
{
    // ...
}

```

2.1.3. FUNCIÓN main()

Cada programa C++ tiene una función `main()` que es el 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* (en C++ siempre son funciones). Todas las sentencias de C++ situadas en el cuerpo de la función `main()`, o de cualquier otra función, deben terminar en punto y coma.

EJEMPLO 2.4. Programa que visualiza el nombre y la dirección.

```

#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    cout << "Lucas Sánchez García\n";
    cout << "Calle Marquillos de Mazarambroz, \n";
    cout << "Mazarambroz, TOLEDO \n";
    cout << "Castilla la Mancha, ESPAÑA\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 2.5. Un ejemplo de algoritmo que incluye un prototipo de una función, su codificación y una llamada.

```

#include <cstdlib>           //cstdlib: contiene funciones de entrada y salida
#include <iostream>
using namespace std;

void prueba();

int main(int argc, char *argv[])
{
    prueba();
    system("PAUSE");
    return EXIT_SUCCESS;
}

void prueba()
{
    cout << "Mis primeros pasos en C++ \n";
}

```

La ejecución del programa anterior es:

Mis primeros pasos en C++

2.1.4. FUNCIONES DEFINIDAS POR EL USUARIO

C++ proporciona funciones predefinidas que se denominan *funciones de biblioteca* y otras *definidas por el usuario*. Las *funciones de biblioteca* requieren que se incluya el archivo donde está su declaración tales como `math.h` o `stdio.h`. Las *funciones definidas por el usuario* se invocan por su nombre y los parámetros actuales opcionales que tengan. Estos parámetros actuales deben coincidir en número orden y tipo con los parámetros formales de la declaración. 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 tipo de la función, el nombre por el que será invocada, así como el número y tipo de sus argumentos. Todos los programas C++ constan de un conjunto de funciones que están controladas por la función `main()`.

EJEMPLO 2.6. El algoritmo siguiente incluye un prototipo de una función `suma`, un programa `main` y con sentencias y una llamada a la función, así como la codificación de la función `suma`.

```
#include <cstdlib>
#include <iostream>
using namespace std;
float suma(float a, float b);

int main(int argc, char *argv[])
{
    float numero, numerol, sumadenumeros;

    numero = 2;
    numerol = 3;
    sumadenumeros = suma(numero, numerol);
    cout << " la suma de " << numero << " y " << numerol << endl;
    cout << " es : " << sumadenumeros << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
float suma(float a, float b)
{
    return a + b;
}
```

El resultado de ejecución del programa anterior es:

**la suma de 2 y 3
es : 5**

2.1.5. COMENTARIOS

Un *comentario* es cualquier información añadida a su archivo fuente e ignorada por el compilador. En C estándar los comentarios comienzan por `/*` y terminan con la secuencia `*/`. En C++ se define una línea de comentario comenzando con una doble barra inclinada (`//`). Todo lo que viene después de la doble barra inclinada es un comentario y el compilador lo ignora.

Los archivos de cabecera en C++ tienen normalmente una extensión <code>.h</code> y los archivos fuente, la extensión <code>.cpp</code> .
--

EJEMPLO 2.7. Programa con comentarios en C y C++.

```
#include <iostream.h>           // archivo cabecera donde se encuentra cout
using namespace std;           // usa la consola como dispositivo estándar

int main(int argc, char *argv[]) // número de argumentos y cuales
{
    // este programa sólo muestra en pantalla el mensaje
    //hola mundo cruel
    cout << " Hola mundo cruel";      // visualiza el mensaje
    system("PAUSE");                // archivo de ejecución retenido hasta pulsar tecla
    return EXIT_SUCCESS;            // retorna el control al sistema operativo.
}
```

2.2. Los elementos de un programa en C++

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

2.2.1. IDENTIFICADOR

Un *identificador* es una secuencia de caracteres, letras, dígitos y subrayados (_) que comienza siempre por un carácter . Las letras mayúsculas y minúsculas son diferentes. Pueden tener cualquier longitud, pero el compilador ignora a partir del 32. No pueden ser palabras reservadas. Son ejemplos de identificadores los siguientes:

nombre_Alumno	Letra_indice	Dia	Mayor	menor
Fecha_Compra	alfa	Habitacion24	i	j

2.2.2. PALABRAS RESERVADAS

C++ como la mayoría de los lenguajes tiene reservados algunos identificadores con un significado especial, que sólo pueden ser usados con ese cometido. Una *palabra reservada*, tal como void (ausencia de tipo, o tipo genérico), es una característica del lenguaje C++. Una palabra reservada no se puede utilizar como nombre de identificador, objeto o función. Son palabras reservadas de C++ las siguientes:

asm	delete	if	return	try
auto	do	inline	short	typedef
bool	double	int	signed	union
break	else	longmutable	sizeof	unsigned
case	enum	namespace	static	virtual
catch	explicit	new	struct	void
char	extern	operator	switch	volatile
class	float	private	template	wchar_t
const	for	protected	this	while
continue	friend	public	throw	
default	goto	register		

2.2.3. COMENTARIOS

Un comentario, estilo C++ comienza con // y termina al final de la línea en que se encuentra el símbolo. Un comentario, estilo C y soportados también por C++, encerrados entre /* y */ puede extenderse a lo largo de varias líneas.

2.2.4. 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:

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

2.2.5. 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 de cabecera se inserta con la directiva `#include`. El nuevo ANSI C++ ha cambiado el convenio (notación) original de los archivos de cabecera. Es posible utilizar sólo los nombres de las bibliotecas sin el sufijo .h; es decir, se puede usar `iostream`, `cmath`, `cassert` y `cstlib` en lugar de `iostream.h`, `math.h`, `assert.h` y `stdlib.h` respectivamente.

2.3. Tipos de datos en C++

C++ no soporta un gran número de tipos de datos predefinidos, pero tiene la capacidad para crear sus propios tipos de datos. Los tipos de datos simples o básicos de C++ son: *enteros*; *números de coma flotante (reales)* y *caracteres*, y se muestran en la Tabla 2.2. Existen tres tipos adicionales en C++ que se tratarán más adelante:

enum	constante de enumeración;
bool	constante falso-verdadero;
void	tipo especial que indica ausencia de tipo;

Tabla 2.1. Tipos de datos simples de C++

Tipo básico	Tipo	Ejemplo	Tamaño en bytes	Rango. Mínimo..Máximo
Entero	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)
Real	double	0.00045	8	2.7*(10 ⁻³⁰⁸) .. 2.7*(10 ³⁰⁸)
	long double	1e-8	8	igual que double

2.3.1. ENTEROS (INT)

Los tipos enteros se almacenan internamente en 2 bytes de memoria. La Tabla 2.2 resume los tres tipos enteros básicos, junto con el rango de valores y el tamaño en bytes usual (depende de cada compilador C++).

Tabla 2.2. Tipos de datos enteros en C++

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.
long	-2147483648 .. 2147483647	Entero largo. Números grandes.
unsigned long	0 .. +4294967295	Entero largo positivo. Números grandes.

A tener en cuenta: el tipo `char` sólo admite los modificadores `unsigned` y `signed`; el tipo `float` y `double` no admiten el modificador `unsigned`; los enteros y enteros largos admiten el modificador `unsined` y `signed`; el tipo `bool` no admite modificadores.

Declaración de variables

La forma más simple de una declaración de variable en C++ es poner primero el tipo de dato y, a continuación, el nombre o los nombres de las variables separados por comas. Si se desea dar un valor inicial a la variable, éste se pone a continuación del identificador de la variable precedido del signo igual:

`<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>`

EJEMPLO 2.8. Declaración e inicialización de variables enteras en C++.

```
int valor, valor1, valor2 = 100, valor3 = 25, x, y;
short int Numero_asociado = 345, numer, libre;
long numero_grande1, numero_grande2 = 345678;
```

C++ permite escribir constantes enteras en *octal* (base 8) o *hexadecimal* (base 16). Una constante octal es cualquier número que comienza con un 0 y contiene dígitos en el rango de 1 a 7. Una constante hexadecimal comienza con 0x y va seguida de los dígitos 0 a 9 o las letras A a F (o bien a a f). La Tabla 2.3 muestra ejemplos de constantes enteras representadas en sus notaciones decimal, hexadecimal y octal.

Tabla 2.3. 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

2.3.2. TIPOS EN COMA FLOTANTE (`float`/`double`)

Los tipos de datos “coma (punto) flotante” representan números reales que contienen una coma (un punto) decimal, tal como 2.123, o números muy grandes, tales como $2.43 \times 10^{18} = 2,43 \times 10^{18}$. 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 2.4 muestra los tipos de datos en coma flotante.

Tabla 2.4. Tipos de datos en coma flotante

Tipo C	Rango de valores	Precisión
<code>float</code>	$3.4 \times 10^{-38} \dots 3.4 \times 10^{38}$	7 dígitos
<code>double</code>	$2.7 \times 10^{-308} \dots 2.7 \times 10^{308}$	15 dígitos
<code>long double</code>	$3.4 \times 10^{-4932} \dots 2.1 \times 10^{4932}$	19 dígitos

2.3.3. CARACTERES (`char`)

Un carácter es cualquier elemento de un conjunto de caracteres predefinidos o alfabeto. La mayoría de las computadoras utilizan el conjunto de caracteres ASCII. C++ procesa datos carácter utilizando el tipo de dato `char`. Internamente, los caracteres se almacenan como números enteros en el rango -128 a +127 y se asocian con el código ASCII; por tanto se pueden rea-

lizar operaciones aritméticas con datos tipo `char`. El lenguaje C++ proporciona el tipo `unsigned char` para representar valores de 0 a 255 y así representar todos los caracteres ASCII. Se pueden realizar operaciones aritméticas con datos tipo `char`.

EJEMPLO 2.9. Define e inicializa una variable de tipo `char`, a continuación se pasa a mayúscula.

```
char car = 'c', car1 = 'D';
car = car - 32;
car1 = car1 + 32;
```

El ejemplo convierte 'c' (código ASCII 99) a 'C' (código ASCII 67), y 'D' (código ASCII 68) a 'd' (código ASCII 100).

EJEMPLO 2.10. Diferencia entre `char` y `unsigned char`.

El programa usa dos variables, `dato`, y `dato1` de tipo `char` y `unsigned char` respectivamente. Asigna a cada variable un número natural comprendido entre 115 y 129, visualizando ambas variables como enteras y como carácter. Cuando no está activado el bit de signo, aparecen números negativos que no se corresponden con los números ASCII asociados.

```
include <cstdlib>
#include <iostream>
using namespace std;

char dato;                                // carácter
unsigned char dato1;                      //declaración de tipo carácter sin signo

int main(int argc, char *argv[])
{
    cout << "    char      unsigned char " << endl;
    cout << " ASCII  CAR  ASCII   CAR " << endl;
    for (int i = 115; i < 130; i++)
    {
        dato = i; dato1 = i;
        cout << (int) dato << "      " << dato << "      ";
        cout << (int) dato1 << "      " << dato1<< endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución:

char	unsigned char	char	unsigned char
15	s	115	s
16	t	116	t
17	u	117	u
18	v	118	v
19	w	119	w
20	x	120	x
21	y	121	y
22	z	122	z
23	�	123	�
24	�	124	�
25	�	125	�
26	~	126	~
27	�	127	�
128	�	128	�
127	�	129	�
Presione una tecla para continuar .			

2.3.4. EL TIPO DE DATO `bool`

La mayoría de los compiladores de C++ incorporan el tipo de dato `bool` cuyos valores posibles son: “verdadero” (`true`) y “falso” (`false`). El tipo `bool` proporciona la capacidad de declarar variables lógicas, que pueden almacenar los valores verdadero y falso. Si en el compilador de C++ no está disponible el tipo `bool`, deberá utilizar el tipo de dato `int` para representar el tipo de dato `bool`. C++ utiliza el valor entero `0` para representar falso y cualquier valor entero distinto de cero (normalmente `1`) para representar verdadero. De esta forma, se pueden utilizar enteros para escribir expresiones lógicas de igual forma que se utiliza el tipo `bool`.

EJEMPLO 2.11. Uso del tipo `bool`.

```
#include <iostream.h>
void main()
{
    bool b1, b2 = false, b3 = true;
    int i1 = b3, i4 = 10;

    cout << " b2:" << b2 << endl; // muestra el valor de b2 que es 0
    ...
}
```

2.4. Constantes

Una constante es un objeto cuyo valor no puede cambiar a lo largo de la ejecución de un programa. En C++ existen cuatro tipos de constantes: *constantes literales*, *constantes definidas*, *constantes enumeradas*, *constantes declaradas*.

2.4.1. 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 siguientes: enteras, reales, de caracteres, de cadena, enumeradas, definidas y declaradas.

Constantes enteras. Son una sucesión de dígitos precedidos o no por el signo + o – dentro de un rango determinado.

EJEMPLO 2.12. Constantes numéricas.

234, -456	constantes enteras.
12334L, 43567L	constantes enteras largas; tienen el sufijo L
0777, 0123	constantes en octal; comienza por 0 y dígitos entre 0 y 7
0xFF3A, 0x2AB345	constantes hexadecimal; comienzan por 0X
3456UL	constante sin signo y larga; sufijo U sin signo

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 `62.e+4`.

Constantes carácter. Una constante carácter (`char`) es un carácter del código ASCII encerrado entre apóstrofes. Por ejemplo, `'A'`, `'a'`, `'b'`, `'c'`. Además de los caracteres ASCII estándar, una constante carácter soporta caracteres especiales que no se pueden representar utilizando su teclado, como por ejemplo los códigos ASCII altos y las secuencias de escape.

Tabla 2.5. Caracteres secuencias (códigos) de escape

Código de Escape	Significado	Códigos ASCII		
		Dec	Hex	
'\n'	nueva línea	13	10	0D
'\r'	retorno de carro	13		0D
'\t'	tabulación	9		09
'\v'	tabulación vertical	11		0B

Tabla 2.5. Caracteres secuencias (códigos) de escape (*continuación*)

Código de Escape	Significado	Códigos ASCII	
		Dec	Hex
'\a'	alerta (pitido sonoro)	7	07
'\b'	retroceso de espacio	8	08
'\f'	avance de página	12	0C
'\\'	barra inclinada inversa	92	5C
'\''	comilla simple	39	27
'\"'	doble comilla	34	22
'\?'	signo de interrogación	34	22
'\000'	número octal	todos	todos
'\xhh'	número hexadecimal	todos	todos

Constantes cadena. Una *constante cadena* (también llamada *literal cadena* o simplemente *cadena*) es una secuencia de caracteres encerrados entre dobles comillas. Por ejemplo, "123", "14 de Julio de 2005", "esto es un ejemplo de cadena". En memoria, las cadenas se representan por una serie de caracteres ASCII más un 0 o nulo '\0' que es definido en C++ mediante la constante NULL en diversos archivos de cabecera (normalmente STDEF.H, STDIO.H, STDLIB.H y STRING.H).

Constantes definidas (simbólicas). Las constantes pueden recibir nombres simbólicos mediante la directiva `#define`. Estas constantes son sustituidas por su valor por el preprocesador (antes de comenzar la compilación del programa).

EJEMPLO 2.13. Se ponen nombres simbólicos a constantes de interés.

```
#define NUEVALINEA '\n'
#define e 2.81
#define pi 3.1415929      // valor de la constante Pi
#define cabecera "esto es un ejemplo de cabecera de programa"
```

C++ sustituye '\n', 2.81, 3.1415929, "esto es un ejemplo de cabecera de programa", cuando se encuentra las constantes simbólicas NUEVALINEA, e, pi, cabecera.

Constantes enumeradas. Las constantes enumeradas permiten crear listas de elementos afines. Cuando se procesa esta sentencia el compilador *enumera* los identificadores comenzando por 0. Despues de declarar un tipo de dato enumerado, se pueden crear variables de ese tipo, como con cualquier otro tipo de datos. En el ejemplo 2.13 Miercoles toma el valor 2.

EJEMPLO 2.14. Declaración de tipos enumerados e inicialización de variables de tipo enumerado.

```
enum laborable { Lunes, Martes, Miercoles, Jueves, Viernes}
enum festivo {Sabado, Domingo};
laborable diaL =lunes;
festivo diaF = Domingo;
```

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`, modifica el tipo de acceso de una variable, permitiendo cambiar el valor de la variable por medios no explícitamente especificados por el programa. Por ejemplo la dirección de una variable global que apunta a un puerto externo. El valor de una variable `volatile` puede ser modificado no sólo por el propio programa, sino también por el *hardware* o por el *software* del sistema. Su formato de declaración es:

```
volatile tipo nombre = valor;
```

Si se usa `const` y `volatile` juntos, se asegura que una variable no puede cambiar por programa, pero sí por medios externos al programa C++.

Ejemplo: `const volatile unsigned char *puerto=0x30;.`

2.5. Variables

En C++ una *variable* es una posición de memoria a que se le asocia un nombre (identificador) en el que se almacena un valor del tipo de dato del que se ha definido. El valor de una variable puede cambiar a lo largo de la ejecución del programa, siendo manipulada por los operadores aplicables al tipo del que ha sido definida la variable.

Declaración. 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. La sintaxis de declaración es:

tipo lista de variables;

Siendo *tipo* el nombre de un tipo de dato conocido por C++ y *lista de variables* una sucesión de nombres separadas por comas, y cada nombre de la lista un identificador de C++.

Inicialización. Las variables, pueden ser inicializadas al tiempo que se declaran. El formato general de una declaración de inicialización es:

tipo lista de inicialización;

Siendo *lista de inicialización* una sucesión *nombre_variable = expresión*. Además *expresión* es cualquier expresión válida cuyo valor es del mismo tipo que *tipo*.

Hay que tener en cuenta que los dos formatos de declaración pueden combinarse entre sí.

EJEMPLO 2.15. Declaración de variables e inicialización de alguna de ellas.

```
int x, z, t = 4;
float xx = 2.0, yy = 8, zz;
char Si = 'S', No = 'N', ch;
```

Las variables *x*, *z* y *t* son enteras. La variable *t* se inicializa a 4.

Las variables *xx*, *yy*, *zz* son reales. Se inicializan *xx* y *zz* a 2.0, y 8 respectivamente.

Las variables *Si*, *No* y *ch* son caracteres. Se inicializan las dos primeras a los caracteres 'S' y 'N' respectivamente.

2.6. Duración de una variable

Dependiendo del lugar de definición, las variables de C++, pueden ser utilizadas en la totalidad del programa, dentro de una función o sólo dentro de un bloque de una función. La zona de un programa en la que una variable está activa se denomina, normalmente, *ámbito* o *alcance* ("scope").

2.6.1. VARIABLES LOCALES

Las variables locales son aquellas que se definen dentro de una función. Las reglas por las que se rigen las variables locales son:

1. Las variables locales no pueden modificarse fuera de la función.
2. Los nombres de las variables locales no son conocidas fuera de donde se declaran.
3. Las variables locales no existen en memoria hasta que se ejecuta la función.

2.6.2. VARIABLES GLOBALES

Las *variables globales* son variables que se declaran fuera de las funciones y por defecto (omisión) son visibles a cualquier función incluyendo `main()`. Una variable global puede ser accedida o modificada desde cualquier función definida del archivo fuente.

EJEMPLO 2.16. *Ámbito de programa, de función y de bloque de variables. Variables locales y globales.*

```
#include <iostream>
using namespace std;
int suma=0;//puede usarse en todas las funciones. Ámbito de programa.
            // suma es una variable global.

int f( )
{
    // puede usar suma por ser global, pero no i ni j por ser locales.
    suma= suma + 30; // efecto lateral sobre suma. Debe evitarse.
}

int main(int argc, char *argv)
{
    int j = 3 ;           // solo puede usarse en la función main
                          // variable local de main

    for (int i = 1; i <= ;i++)
        suma = suma + i;
                    //la variable i solo puede usarse dentro del bucle for.
                    // i tiene ámbito de bloque
                    //i es variable local del bucle for

    suma = suma + j;//puede usar suma por ser global y j por ser local
    f( );
    cout << suma ;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.6.4. VARIABLES DINÁMICAS Y DE OBJETOS

Las *variables dinámicas* similares tanto a variables locales como a globales. Al igual que una variable local, una variable dinámica se crea y libera durante la ejecución de una función. La diferencia entre una variable local y una variable dinámica es que la variable dinámica se crea tras su petición (en vez de automáticamente, como las variables locales), y se libera cuando ya no se necesita. Al igual que una variable global, se pueden crear variables dinámicas que son accesibles desde múltiples funciones. Las variables dinámicas se examinan en detalle en el Capítulo 10 (*Punteros*).

Los objetos son tipos agragados de dato que pueden contener múltiples funciones y variables juntas en el mismo tipo de dato. En el Capítulo 12 se introducen los objetos, tipo de dato clave en C++ y sobre todo en programación orientada a objetos.

2.7. Entradas y salidas

En C++ la entrada y salida se lee y escribe en *fluxos (streams)*. Cuando `iostream.h` se incluye en un programa, diferentes fluxos estándar son definidos automáticamente. El flujo `cin` se utiliza para entrada, que normalmente se lee de teclado. El flujo `cout` se utiliza para salida y, normalmente, se envía a la pantalla del usuario.

2.7.1. SALIDA (cout)

El *operador de inserción*, `<<`, inserta datos en el flujo `cout` que los visualiza en la pantalla de su equipo. Es posible utilizar una serie de operadores `<<` en cascada

```
cout << "Esto es una cadena";           //visualiza: Esto es una cadena
cout << 500 << 600 << 700;           //visualiza 500 600 700
cout << 500 << "," << 600;           //visualiza 500, 600
```

C++ utiliza *secuencias de escape* para visualizar caracteres que no están representados por símbolos tradicionales. Entre las más usadas están: *línea.nueva línea* (`\n`), *tabulación* (`\t`) y *alarma* (`\a`). Una lista de las secuencias completa de escape se recoge en la Tabla 2.6.

Ejemplo:

```
cout << "\n Error \n Pulsar una tecla para continuar \n";
```

La instrucción anterior salta de línea, escribe en otra línea *Error*, salta de línea escribe *Pulsar una tecla para continuar* y salta de línea. Es decir produce la siguiente salida:

```
Error
Pulsar una tecla para continuar
```

Tabla 2.6. Caracteres secuencias de escape

Secuencia de escape	Significado
<code>\a</code>	Alarma
<code>\b</code>	Retroceso de espacio
<code>\f</code>	Avance de página
<code>\n</code>	Retorno de carro y avance de línea
<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulación
<code>\v</code>	Tabulación vertical
<code>\\\</code>	Barra inclinada
<code>\?</code>	Signo de interrogación
<code>\"</code>	Dobles comillas
<code>\000</code>	Número octal
<code>\xhh</code>	Número hexadecimal
<code>\0</code>	Cero, nulo (ASCII 0)

2.7.2. ENTRADA (cin)

El archivo de cabecera `iostream.h` de la biblioteca C++ proporciona un flujo de entrada estándar `cin` y un *operador de extracción*, `>>`, para extraer valores del flujo y almacenarlos en variables. Si no se redirige explícitamente `cin`, la entrada procede del teclado.

```
int numero;                  double real;
cin >> numero;              cin >> real;
```

Las órdenes `cin` leen dos datos del teclado y los almacenan en las variables `numero` y `real`.

EJEMPLO 2.17. Programa que lee las iniciales del nombre y primer apellido y las escribe en pantalla separadas de un punto.

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

```

int main()
{
    char Nombre, Apellido;

    cout << "Introduzca la inicial de su nombre y primer apellido: ";
    cin >> Nombre >> Apellido;
    cout << "Hola," << Nombre << "." << Apellido << ".\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

2.8. Espacios de nombres (NAMESPACES)

Un espacio de nombres es un mecanismo para agrupar lógicamente un conjunto de identificadores (nombres de tipos, funciones, etc.). Existe un espacio de nombres global y sobre él se define el resto de los espacios de nombres. A un identificador que se encuentra en un espacio de nombres se puede acceder de distintas formas.

EJEMPLO 2.18. *Espacio de nombre estándar.*

En el programa estándar, la cláusula `using` carga espacio de nombres estándar (`std`) en el espacio de nombres actual, que es global permitiendo así el empleo de identificadores como `endl` (declarado en `<iostream>` sin necesidad de cuatificar con el operador de alcance).

```

// Programa inicial
#include <iostream>
.....
using namespace std;
int main ( )
{
    cout << "¡Hola mundo cruel!" << endl;
}

```

Un espacio de nombres `namespace` es una región declarativa con nombre opcional. El nombre de un espacio de nombres se puede utilizar para acceder a entidades declaradas en ese espacio de nombre; es decir los miembros del espacio de nombres. En esencia, un conjunto de variables, de funciones, de clases y de subespacios de nombres, miembros que siguen unas reglas de visibilidad. El espacio de nombres es una característica de C++ introducida en las últimas versiones, diseñada para simplificar la escritura de programas.

Definición de un espacio de nombres:

`namespace identificador{cuerpo_del_espacio_de_nombres }` ← No hay punto y coma

<code>cuerpo_del_espacio_de_nombres</code>	
<code>sección_de_declaraciones.</code>	//miembros del espacio de nombre

Para acceder al espacio de nombres se debe invocar al nombre del mismo cuando se refiera a ellos. Existen dos procedimientos para hacer esto.

Método 1. Preceder a cada nombre del elemento en el nombre del espacio de nombre y el operador de resolución del ámbito de alcance (`::`).

Método 2. Utilizar la directiva `using`, lo que permite poder utilizar el espacio de nombres a partir de donde se declare.

EJEMPLO 2.19. *Espacio de nombres geo.*

```
#include <iostream>
namespace geo
{
    const double PI = 3.141592;
    double longcircun (double radio)
    {
        return 2*PI*radio;
    }
}

using namespace std;
using namespace geo;
int main(int argc, char *argv[])
{
    cout << "¡Hola mundo cruel!" << endl;
    cout << longcircun (16); //no funciona al omitir using namespace geo;
    cout << geo::longcircun(20);
}
```

EJERCICIOS

2.1. *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    // cout << "Hola mundo!\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.2. *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
using namespace std;
#define prueba "esto es una prueba"
int main()
{
    char cadena[21]="sale la cadena.";
    cout << prueba << endl;
    cout << "Escribimos de nuevo.\n";
    cout << cadena << endl;
    cout << &cadena[8] << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- 2.3.** *Escribir un programa que visualice la letra B con asteriscos.*

```
*****
*      *
*      *
*      *
*****  
*      *
*      *
*      *
*****
```

- 2.4.** *Codificar un programa en C++ que escriba en dos líneas distintas las frases: Bienvenido al C++; Pronto comenzamos a programar en C.*

- 2.5.** *Diseñar 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.*

- 2.6.** *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
#define Constante "de declaracion de constante."
using namespace std;

int main( )
{
    char Salida[21] = "Esto es un ejemplo";

    cout << Salida << endl;
    cout << Constante << endl;
    cout << "Salta dos lineas\n \n";
    cout << "y tambien un\n";
    cout << &Salida[11];
    cout << " cadenas\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- 2.7.** *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main( )
{
    char pax[] = "Juan Sin Miedo";

    cout << pax << "----> " << &pax[4] << endl;
    cout << pax << endl;
    cout << &pax[9] << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.8. Escribir un programa que lea una variable entera y dos reales y lo visualice.

2.9. Escribir un programa que lea el largo y el ancho de un rectángulo.

2.10. ¿Cuál de los siguientes identificadores son válidos?

N	85_Nombre
MiProblema	AAAAAAA
Mi_Juego	Nombre_Apellidos
MiJuego	Saldo_Actual
write	92
m&m	Universidad_Pontificia
registro	Set_15
* 143Edad	

SOLUCIÓN DE LOS EJERCICIOS

2.1. El programa produce en la salida el mensaje , pulse una tecla para continuar correspondiente a la sentencia `system ("PAUSE")`, ya que la sentencia `cout` está como comentario, y por tanto no se ejecuta. Si se eliminar a el comentario aparecería además el mensaje: Hola Mundo!

2.2. Un programa que resuelve el ejercicio es:

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    cout << "*****\n";
...cout << "*      *\n";
...cout << "*      *\n";
...cout << "*      *\n";
...cout << "*****\n";
...cout << "*      *\n";
...cout << "*      *\n";
...cout << "*      *\n";
...cout << "*****\n";
...system("PAUSE");
    return EXIT_SUCCESS;
}
```



- 2.3. La salida se refleja en el resultado de ejecución que se muestra a continuación.

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

- 2.4. Una codificación es:

```
#include <cstdlib>  
#include <iostream>  
using namespace std;  
int main()  
{  
    cout << " Bienvenido al C++ \n";  
    cout << " Pronto comenzaremos a programar en C++ \n";  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

- 2.5. Un programa que resuelve el problema planteado es:

```
#include <cstdlib>  
#include <iostream>  
using namespace std;  
int main()  
{  
    char ejemplo[50];  
  
    strcpy (ejemplo, " Es un nuevo ejemplo de programa en C\n");  
    cout << ejemplo << endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

- 2.6. El resultado de ejecución del programa es:

```
Esto es un ejemplo  
de declaracion de constante.  
Salta dos lineas  
  
y tambien un  
ejemplo cadenas
```

- 2.7. Al ejecutar el programa se obtiene:

```
Juan Sin Miedo---->  Sin Miedo  
Juan Sin Miedo  
Miedo
```

- 2.8. La codificación pedida es:

```
#include <cstdlib>  
#include <iostream>
```

```

using namespace std;
int main()
{
    int v1 ;
    float v2,precio;

    cout << "Introduzca v1 " ;
    cin >> v1 ;                                //lectura valor de v1
    cout << "valor leidos: " << v1 << "\n";
    cout << "introduzca dos valores\n";
    cin >> v2 >> precio;                      // lectura de v2, precio
    cout << v2 << " " << precio << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

2.9. Una codificación es la siguiente:

```

#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{

    float largo, ancho;

    cout << "Introduzca largo: ";
    cin >> largo;
    cout << " introduzca ancho: ";
    cout << "introduzca dos valores\n";
    cin >> ancho;
    cout << " largo = " << largo << "\n" ;
    cout << " ancho = " << ancho << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

2.10. N

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

EJERCICIOS PROPUESTOS

- 2.1. Depurar y escribir un programa que visualice la letra A mediante asteriscos.
- 2.2. Escribir un programa que lea un texto de cinco líneas y lo presente en pantalla.
- 2.3. Escribir un programa que lea 5 números enteros y tres números reales y los visualice.
- 2.4. Escribir y ejecutar un programa que lea su nombre y dirección y visualice la misma.
- 2.5. Escribir un programa que lea la base y la altura de un trapecio.
- 2.6. Escribir y ejecutar un programa que imprima una página de texto con no más de 40 caracteres por línea.
- 2.7. Escribir un programa que lea el radio de una circunferencia y calcule su longitud.
- 2.8. Escribir un programa que lea tres números reales, y los visualice.
- 2.9. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    char p[] = "Esto es una prueba";

    cout << p << " " << &p[2] << endl;
    cout << p << "\n";
    cout << &p[2] << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

CAPÍTULO 3

Operadores y expresiones

Introducción

Este capítulo muestra cómo 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 y especiales*. Además, se analizan las versiones de tipos de datos y las reglas que sigue el compilador cuando concurren en una misma expresión diferentes tipos de operadores. Estas reglas se conocen como reglas de *prioridad y asociatividad*.

3.1. 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 que especifican un cálculo. C++ soporta un conjunto potente de operadores unitarios, binarios y de otros tipos.

3.2. Operador de asignación

El operador de asignación es un operador cuya sintaxis es la siguiente:

```
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 = D = 12;` equivale a `A = (B = (C = (D = 12)))` que asigna a las variables A, B, C y D el valor 12. 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 3.1.

EJEMPLO 3.1. Programa que declara variables y hace uso de operadores de asignación.

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

```

int main(int argc, char *argv[])
{
    int codigo, CoordX, CoordY;
    float fahrenheit, valor;

    codigo = 3467;
    valor = 10;
    valor *= 3;
    CoordX = 525;
    CoordY = 725;
    CoordY -= 10;
    fahrenheit = 123.456;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Tabla 3.1. Operadores y equivalencias de asignación de C++

Símbolo	Uso	Descripción	Sentencia abreviada	Sentencia no abreviada
=	a = b	Asigna el valor de <i>b</i> a <i>a</i> .	m = n	m = n
*=	a *= b	Multiplica <i>a</i> por <i>b</i> y asigna el resultado a la variable <i>a</i> .	m *= n;	m = m * n;
/=	a /= b	Divide <i>a</i> entre <i>b</i> y asigna el resultado a la variable <i>a</i> .	m /= n;	m = m / n;
%=	a %= b	Fija <i>a</i> al resto de <i>a/b</i> .	m %= n;	m = m % n;
+=	a += b	Suma <i>b</i> y <i>a</i> y lo asigna a la variable <i>a</i> .	m += n;	m = m + n;
-=	a -= b	Resta <i>b</i> de <i>a</i> y asigna el resultado a la variable <i>a</i> .	m -= n;	m = m - n;

3.3. Operadores aritméticos

Los operadores aritméticos de C++ sirven para realizar operaciones aritméticas básicas. Siguen las reglas algebraicas típicas, de jerarquía o prioridad, clásicas de matemáticas. Estos operadores vienen recogidos en la Tabla 3.2.

Tabla 3.2. 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*. La prioridad de los operadores y su asociatividad se recogen en la Tabla 3.3.

Tabla 3.3. Prioridad y asociatividad

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

EJEMPLO 3.2. *Evaluación de expresiones.*

a) ¿Cuál es el resultado de evaluación de la expresión: $50 - 4 * 3 + 2$?

$50 - 4 * 3 + 2$
 $50 - 12 + 2$
 $38 + 2$
 32

b) ¿Cuál es el resultado de evaluación de la expresión: $5 * (10 - 2 * 4 + 2) - 2$?

$5 * (10 - 2 * 4 + 2) - 2$
 $5 * (10 - 8 + 2) - 2$
 $5 * (2 + 2) - 2$
 $5 * 4 - 2$
 $20 - 2$
 18

c) ¿Cuál es el resultado de la expresión: $7 * 5 - 6 \% 4 * 4 + 9$?

$7 * 5 - 6 \% 4 * 4 + 9$
 $35 - 6 \% 4 * 4 + 9$
 $35 - 2 * 4 + 9$
 $35 - 8 + 9$
 $27 + 9$
 36

d) ¿Cuál es el resultado de la expresión: $15 * 14 - 3 * 7$?

$15 * 14 - 3 * 7$
 $210 - 3 * 7$
 $210 - 21$
 189

e) ¿Cuál es el resultado de la expresión: $3 + 4 * (8 * (4 - (9 + 3) / 6))$?

$3 + 4 * (8 * (4 - (9 + 2) / 6))$
 $3 + 4 * (8 * (4 - 11 / 6))$
 $3 + 4 * (8 * (4 - 1))$
 $3 + 4 * (8 * 3)$
 $3 + 4 * 24$
 $3 + 96$
 99

EJEMPLO 3.3. *Programa que lee el radio, calcula y visualiza la longitud de la circunferencia de ese radio, y el área del círculo del mismo radio.*

```
#include <cstdlib>
#include <iostream>
#define pi 3.141592
using namespace std;

int main(int argc, char *argv[])
{
    float radio, longitud, area;
```

```

    cin >> radio;                      //lectura del radio
    longitud = 2 * pi * radio;
    area = pi * radio * radio;
    cout << " radio = " << radio << endl;
    cout << " longitud = " << longitud << endl;
    cout << " area = " << area << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 3.4. Desglosar cierta cantidad de segundos introducida por teclado en su equivalente en semanas, días, horas, minutos y segundos.

El programa lee el número de segundos y realiza las conversiones, teniendo en cuenta que una semana tiene 7 días, un día tiene 24 horas, una hora 60 minutos, y un minuto 60 segundos.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int semanas, dias, horas, minutos, segundos, acu;

    cout << "Introduzca segundos ";
    cin >> 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;
    cout << "segundos en semanas dias horas minutos y segundos " << endl;
    cout << " numero de semanas " << semanas << endl;
    cout << " numero de dias " << dias << endl;
    cout << " numero de horas " << horas << endl;
    cout << " numero de minutos " << minutos << endl;
    cout << " numero de segundos " << segundos << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

En resultado de ejecución es el siguiente:

```

Introduzca segundos 4567456
segundos en semanas dias horas minutos y segundos
numero de semanas 7
numero de dias 3
numero de horas 20
numero de minutos 44
numero de segundos 16
Presione una tecla para continuar . . .

```

3.4. Operadores de incrementación y decrementación

De las características que incorpora C++ una de las más útiles son los operadores de *incremento ++* y *decremento --*. Estos operadores unitarios suman o restan 1 respectivamente a la variable y tienen la propiedad de que pueden utilizarse como sufijo o prefijo. El resultado de la expresión puede ser distinto, dependiendo del contexto. En la Tabla 3.4 se recogen los operadores de incrementación y decrementación.

Tabla 3.4. Operadores de incrementación (++) y decrementación (--)

Incrementación	Decrementación
<code>++n, n++</code>	<code>--n, n--</code>
<code>n += 1</code>	<code>n -= 1</code>
<code>n = n + 1</code>	<code>n = n - 1</code>

Si los operadores ++ y -- están de prefijos, la operación de incremento se efectúa antes que la operación de asignación; si los operadores ++ y -- están de sufijos, la asignación se efectúa en primer lugar y la incrementación o decrementación a continuación.

EJEMPLO 3.5. Diferencias entre operadores de preincrementación y postincrementación.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int m = 10, n;

    n = ++m;           // primero se incrementa m y luego se asigna a n
    cout << " m = " << m << " n = " << n << endl ;
    n = m++;          // primero se asigna a n y luego se incrementa m
    cout << " m = " << m << " n = " << n << endl ;
    cout << " m = " << m++ << endl;
    cout << " m = " << ++m << endl;
    n = 5;
    m = ++n * --n;      // ++n pone n a 6, luego --n pone n a 5, luego m = 25
    cout << " n = " << n << " m = " << m << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución:

```
m = 11 n = 11
m = 12 n = 11
m = 12
m = 14
n = 5 m = 25
```

3.5. Operadores relacionales

C++ soporta el tipo `bool` que tiene dos literales `false` y `true`. Una expresión booleana es, por consiguiente, una secuencia de operandos y operadores que se combinan para producir uno de los valores `true` y `false`. C++ utiliza el tipo `int` para representar los valores verdadero (`true`) y falso (`false`). El valor entero 0 representa a falso y cualquier valor distinto de cero a ver-

dadero. 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 3.5

La Tabla 3.5 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: `char`, `int`, `float`, `double`, etc.

Tabla 3.5. Operadores relacionales de C

Operador	Significado	forma	Ejemplo
<code>==</code>	<i>Igual a</i>	<code>a == b</code>	<code>'A' == 'C'</code> falso
<code>!=</code>	<i>No igual a</i>	<code>a != b</code>	<code>2 != 4</code> verdadero
<code>></code>	<i>Mayor que</i>	<code>a > b</code>	<code>5 > 6</code> falso
<code><</code>	<i>Menor que</i>	<code>a < b</code>	<code>'a' < 'c'</code> verdadero
<code>>=</code>	<i>Mayor o igual que</i>	<code>a >= b</code>	<code>'B' >= 'b'</code> falso
<code><=</code>	<i>Menor o igual que</i>	<code>a <= b</code>	<code>2 <= 1</code> falso

3.6. 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 unitario lógico `!` (`not`, `no`) produce *falso* (cero) si su operando es *verdaderos* (distinto de cero) y vice versa. El operador binario lógico `&&` (`and`, `y`) produce *verdadero* sólo si ambos operandos son *verdaderos* (no cero); si cualquiera de los operandos es *falso* produce *falso*. El operador binario lógico `||` (`or`, `o`) produce *verdadero* si cualquiera de los operandos es *verdadero* (distinto de cero) y produce *falso* sólo si ambos operandos son *falsos*.

La precedencia de los operadores es: los operadores matemáticos tienen precedencia sobre los operadores relacionales, y los operadores relacionales tienen precedencia sobre los operadores lógicos.

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*.

Las sentencias de asignación se pueden escribir de modo que se puede dar un valor de tipo `bool` o una variable `bool`.

EJEMPLO 3.6. Sentencias de asignación a los tipos de variables `bool`.

```
int n;
float x ;
char car;
bool r, esletra, espositiva;

r = (n > -100) && (n < 100);           // true si n están entre -110 y 100
esletra = ((`A`<= car) && (car <= `Z`))
    (((`a`<= car) && (car <= `Z`));
     //si car es una letra la variable es letra true, y toma false en otro caso
espositivo = x >= 0;
```

3.7. 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. Estas operaciones son comparables en eficiencia y en velocidad a sus equivalentes en lenguaje ensamblador. 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 3.6 recoge los operadores lógicos bit a bit.

Tabla 3.6. Operadores lógicos bit a bit

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

3.7.1. OPERADORES DE DESPLAZAMIENTO DE BITS (>>, <<)

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

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

EJEMPLO 3.7. *Sentencias de manipulación de bits.*

variable	V. entero	V binario	
X	9	00001001	1001
Y	10	00001010	1010
x & y	8	00001000	1001 & 1010 and bit a bit
x y	11	00001011	1001 1010 or bit a bit
x ^ y	3	00000011	1001 ^ 1010 xor bit a bit
x << 2	36	01000100	1000100 aparecen dos ceros a la derecha
y >> 2	2	00000010	10

El siguiente programa comprueba los valores indicados en la tabla anterior.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{ int x = 9, y = 10, x_y_y, x_or_y, x_Xor_y;

    x_y_y = x & y;
    x_or_y = x | y;
    x_Xor_y = x ^ y;
    cout << "x = " << x << "\n";
    cout << "y = " << y << "\n";
    cout << "x_y_y = " << x_y_y << "\n";
```

```

cout << "x_or_y = " << x_or_y << "\n";
cout << "x_Xor_y = " << x_Xor_y << "\n";
x = x << 2;
y = y >> 2;
cout << "x << 2 = " << x << "\n";
cout << "y >> 2 = " << y << "\n";
system("PAUSE");
return EXIT_SUCCESS;
}

```

El resultado de ejecución del programa anterior es:

```

x = 9
y = 10
x_y_y = 8
x_or_y = 11
x_Xor_y = 3
x << 2 = 36
y >> 2 = 2

```

3.7.2. OPERADORES DE ASIGNACIÓN ADICIONALES

Los operadores de asignación abreviados están disponibles también para operadores de manipulación de bits. Estos operadores se muestran en la Tabla 3.7.

Tabla 3.7. Operadores de asignación adicionales

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

3.7.3. OPERADORES DE DIRECCIONES

Son operadores que permiten manipular las direcciones de los objetos. Se recogen en la Tabla 3.8.

Tabla 3.8. Operadores de direcciones

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

3.8. Operador condicional ?

El **operador condicional ?**, es un operador ternario que devuelve un resultado cuyo valor depende de la condición cumplida. El formato del operador condicional es:

`expresion_L ? expresion_v : expresion_f;`

Se evalúa la expresión lógica `expresion_L` y su valor 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 3.8. *Uso del operador condicional ?.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x = 10, y = 12, mayor, menor;
    bool z;

    z = x >= y ? true: false;           // z toma el valor de false.
    mayor = x >= y ? x : y;            // calcula y almacena el mayor
    menor = x >= y ? y : x;            // calcula y almacena el menor
    cout << "x = " << x << "\n";
    cout << "y = " << y << "\n";
    cout << "el mayor es = " << mayor << "\n";
    cout << "el menor es = " << menor << "\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultados de ejecución:

```
x = 10
y = 12
el mayor es = 12
el menor es = 10
```

3.9. 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 de la expresión. El uso del operador coma es:

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

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

EJEMPLO 3.9. *Uso del operador condicional ,.*

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int i, j, r, k;
```

```

r = j = 10, i = j, k = (i++, i+1) ;
cout << i= "<< << " j= " << j << " r= " << r << " k= " << k <<"\n";
system("PAUSE");
return EXIT_SUCCESS;
}

```

El resultado de ejecución es:

```
i= 11 j= 10 r= 10 k= 12
```

j toma el valor de 10, r toma el valor de j que es 10, i toma el valor de 10. Posteriormente i toma el valor 11; se ejecuta la expresión i+1 que es 12 y se asigna a k.

3.10. Operadores especiales (), [] y ::

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

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 y resolver los conflictos de prioridad entre operadores.

El operador [] sirve para designar un elemento de un array. También se puede utilizar en unión con el operador delete; en este caso, indica el tamaño del array a destruir y su sintaxis es: `delete [tamaño_array] puntero_array;`

El operador :: es específico de C++ y se denomina *operador de ámbito de resolución*, y permite especificar el alcance o ámbito de un objeto. Su sintaxis es:

```
class::miembro    o bien    ::miembro
```

3.11. 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

```

EJEMPLO 3.10. El siguiente programa escribe el tamaño de los tipos de datos en su ordenador.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    cout << " el tamaño de variables de esta computadora son:\n";
    cout << " entero: " << sizeof(int) << '\n';
    cout << " entero largo: " << sizeof(long int) << '\n';
    cout << " rael: " << sizeof(float) << '\n';
    cout << " doble: " << sizeof(double) << '\n';
    cout << " long doble: " << sizeof(long double) << '\n';
    cout << " long doble: " << sizeof(20) << '\n';
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

3.12. Conversiones de tipos

Las conversiones de tipos pueden ser *implícitas* (ejecutadas automáticamente) o *explícitas* (solicitadas específicamente por el programador). C++ hace muchas conversiones de tipos automáticamente: convierte valores cuando se asigna un valor de un tipo a una variable de otro tipo; convierte valores cuando se combinan tipos mixtos en expresiones; convierte valores cuando se pasan argumentos a funciones.

Conversión implícita. 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 siguiente lista determina a qué operación se 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, etc.

Conversiones explícitas. C++ fuerza la conversión explícita de tipos mediante el operador de *moldé* (*cast*). El operador *moldé* tiene el formato `(tiponombre)valor`. Convierte `valor a tiponombre`.

Por ejemplo dada la declaración: `float x;`

```
x = 3/2           asigna a x el valor de 1,  
x = (float)3/2    asigna a x el valor de 1.5
```

EJEMPLO 3.10. *El siguiente programa muestra conversiones implícitas y explícitas de enteros y caracteres.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char c = 'Z' + 1; // asigna a c el siguiente carácter de 'Z'

    cout << 'A' << " " << (int)'A' << endl; // carácter y número ASCII
    cout << '0' << " " << (int)'0' << endl; // carácter y número ASCII
    cout << 'a' << " " << (int)'a' << endl; // carácter y número ASCII
    cout << c << " " << (int)c << endl; // carácter y número ASCII
    cout << 'Z'+1 << " " << (char)('Z'+1) << endl; // número ASCII y carácter
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

La ejecución del programa anterior es:

```
A 65
0 48
a 97
[ 91
91 [
```

3.13. Prioridad y asociatividad

La **prioridad** o **precedencia** de operadores determina el orden en el que se aplican los operadores a un valor. A la hora de evaluar una expresión hay que tener en cuenta las siguientes reglas y la Tabla 3.9.

- 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.

Tabla 3.9. Prioridad y asociatividad de los operadores

Prioridad	Operadores	Asociatividad
1	<code>:: x -> [] ()</code>	I - D
2	<code>++ -- ~ ! - + & * sizeof</code>	D - I
3	<code>.* ->*</code>	I - D
4	<code>* / %</code>	I - D
5	<code>+ -</code>	I - D
6	<code><< >></code>	I - D
7	<code>< <= > >=</code>	I - D
8	<code>== !=</code>	I - D
9	<code>&</code>	I - D
10	<code>^</code>	I - D
11	<code> </code>	I - D
12	<code>&&</code>	I - D
13	<code> </code>	I - D
14	<code>? : (expresión condicional)</code>	D - I
15	<code>= *= /= %= += -= <<= >>= &= = ^= , (operador coma)</code>	D - I
16		I
<i>I - D : Izquierda - Derecha</i>		<i>D - I : Derecha - Izquierda.</i>

EJERCICIOS

3.1. Determinar el valor de las siguientes expresiones aritméticas:

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

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

$$\begin{array}{ll} a) 10 * 14 - 3 * 2 & d) (4 - 40 / 5) \% 3 \\ b) -4 + 5 * 2 & e) 4 * (3 + 5) - 8 * 4 \% 2 - 5 \\ c) 13 - (24 + 2 * 5) / 4 \% 3 & f) -3 * 10 + 4 * (8 + 4 * 7 - 10 * 3) / 6 \end{array}$$

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

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

- 3.4.** Escribir las sentencias de asignación que permitan intercambiar los contenidos (valores) de dos variables x , e y de un cierto tipo de datos.
- 3.5.** Escribir un programa que lea dos enteros en las variables x e y , a continuación, obtenga los valores de: a) x / y ; b); $x \% y$. Ejecute el programa varias veces con diferentes pares de enteros como entrada.
- 3.6.** Una temperatura dada en grados Celsius (centígrados) puede ser convertida a una temperatura equivalente Fahrenheit de acuerdo a la siguiente fórmula: $f = \frac{9}{5}c + 32$. Escribir un programa que lea la temperatura en grados centígrados y la convierta a grados Fahrenheit.

PROBLEMAS

- 3.1.** La relación entre los lados (a,b) de un triángulo y la hipotenusa (h) viene dada por la fórmula: $a^2 + b^2 = h^2$. Escribir un programa que lea la longitud de los lados y calcule la hipotenusa .

- 3.2.** Escribir un programa que lea un entero y , a continuación, visualice su doble y su triple .

- 3.3.** Escriba un programa que lea los coeficientes a , b , c , d , e , f de un sistema lineal de dos ecuaciones con dos incógnitas y muestre la solución.

$$\begin{cases} ax + by = c \\ cx + dy = f \end{cases}$$

- 3.4.** 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, $G = 6.673 \times 10^{-8} \text{ cm}^3/\text{g. seg}^2$

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 $\text{gr. cm}/\text{seg}^2$

- 3.5.** La famosa ecuación de Einstein para conversión de una masa m en energía viene dada por la fórmula: $E = cm^3$, donde 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.

- 3.6.** Escribir un programa para convertir una medida dada en pies a sus equivalentes en: a) yardas; b) pulgadas; donde 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.

- 3.7.** Escribir un programa en el que se introduzca como datos de entrada la longitud del perímetro de un terreno, expresada con tres números enteros que representen hectómetros, decámetros y metros respectivamente, y visualice el perímetro en decímetros.

- 3.8.** Escribir un programa que solicite al usuario una cantidad de euros y transforme la cantidad en euros en billetes y monedas de curso legal (cambio óptimo).

SOLUCIÓN DE LOS EJERCICIOS

- 3.1.** Hay que tener en cuenta que / en el caso de números enteros calcula el cociente de la división entera, y % calcula el resto de la división entera. Por tanto los resultados son:

$$15 / 12 = 1$$

$$24 / 12 = 2$$

$$123 / 100 = 1$$

$$200 / 100 = 2$$

$$15 \% 12 = 3$$

$$24 \% 12 = 0$$

$$123 \% 100 = 23$$

$$200 \% 100 = 0$$

- 3.2.** La solución por pasos de cada uno de ellos por pasos es:

<p>a) $10 * 14 - 3 * 2$ $140 - 3 * 2$ $140 - 6$ 134</p>	<p>d) $(4 - 40 / 5) \% 3$ $(4 - 8) \% 3$ $-4 \% 3$ -1</p>
<p>b) $4 + 5 * 2$ $4 + 10$ 14</p>	<p>e) $4 * (3 + 5) - 8 * 4 \% 2 - 5$ $4 * 8 - 8 * 4 \% 2 - 5$ $32 - 8 * 4 \% 2 - 5$ $32 - 32 \% 2 - 5$ $32 - 0 - 5$ $32 - 5$ 27</p>
<p>c) $13 - (24 + 2 * 5) / 4 \% 3$ $13 - (24 + 10) / 4 \% 3$ $13 - 34 / 4 \% 3$ $13 - 8 \% 3$ $13 - 2$ 11</p>	<p>f) $-3 * 10 + 4 * (8 + 4 * 7 - 10 * 3) / 6$ $-30 + 4 * (8 + 4 * 7 - 10 * 3) / 6$ $-30 + 4 * (8 + 28 - 10 * 3) / 6$ $-30 + 4 * (8 + 28 - 30) / 6$ $-30 + 4 * (36 - 30) / 6$ $-30 + 4 * 6 / 6$ $-30 + 24 / 6$ $-30 + 4$ -26</p>

- 3.3.** La potencia puede hacerse con la función pow(), por ejemplo $(x + y)^2 == \text{pow}(x+y, 2)$

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

- 3.4.** Si Aux es una variable del mismo tipo que x e y las sentencias pedidas son:

```
Aux = x;
x = y;
y = Aux;
```

- 3.5.** Una codificación del programa solicitado es:

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

```

int main(int argc, char *argv[])
{
    int x , y ;

    cin >> x >> y ;
    cout << x / y << " << x % y<< endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Al ejecutar varias veces el programa se observa que / obtiene el cociente entero, y % obtiene el resto de la división entera.

3.6. Una codificación del programa solicitado es:

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float c, f;

    cout << "Introduce grados ";
    cin >> c;
    f = c * 9 / 5 + 32;
    cout << " grados fahrenheit = " << f << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

SOLUCIÓN DE LOS PROBLEMAS

3.1. El código solicita los lados, y visualiza los lados y la hipotenusa:

```

#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char *argv[])
{
    float a, b, h;

    cout << "Introduce los lados ";
    cin >> a >> b;
    h = sqrt( a * a + b * b);
    cout << " lado 1 = " << a << endl;
    cout << " lado 2 = " << b << endl;
    cout << " hipotenusa = " << h << endl;
}

```

```

    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Un resultado de ejecución es:

```

Introduce los lados 3 4
lado 1 = 3
lado 2 = 4
hipotenusa = 5

```

- 3.2.** La codificación pedida es:

```

#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int x;

    cout << " dame un numero entero ";
    cin >> x;
    x = 2 * x;
    cout << "su doble es " << x << " su triple es " << 3 * x << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Un resultado de ejecución es:

```

dame un numero entero 6
su doble es 12 su triple es 36
Presione una tecla para continuar . . .

```

- 3.3.** Un sistema lineal de dos ecuaciones con dos incógnitas $\begin{cases} ax + by = c \\ cx + dy = f \end{cases}$ tiene solución única si y solamente si $a * e - b * d$ es distinto de cero, y además la solución viene dada por las expresiones siguientes:

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

El programa codificado solicita al usuario los valores a, b, c, d, e, f , y en caso de que exista solución la muestra.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float a, b, c, d, e, f, denominador, x, y;

    cout << " Introduzca el valor de a de b y de c ";
    cin >> a >> b >> c;

```

```
cout << " Introduzca el valor de d de e y de f " ;
cin >> d >> e >> f;
denominador = a * e - b * d;
if (denominador == 0)
    cout << " no solucion\n";
else
{
    x = (c * e - b * f) / denominador;
    y = (a * f - c * d) / denominador;
    cout << " la solucion del sistema es\n";
    cout << " x = " << x << " y = " << y << endl;
}
system("PAUSE");
return EXIT_SUCCESS;
}
```

Un resultado de ejecución del programa anterior es:

```
Introduzca el valor de a de b y de c 2 3 5
Introduzca el valor de d de e y de f 1 6 7
la solucion del sistema es
x = 1 y = 1
Presione una tecla para continuar . . . -
```

- 3.4. Se declara la constante de gravedad universal $G = 6.673e-8$, así como las variables *masa1*, *masa2*, *distancia*, *fuerza* para posteriormente aplicar la fórmula y presentar el resultado:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    const float G = 6.673e-8;

    float masa1, masa2, distancia, fuerza;
    cout << " Introduzca la masa de los dos cuerpos en gramos:\n ";
    cin >> masa1 >> masa2;
    cout << " Introduzca la distancia entre ellos en centimetros:\n ";
    cin >> distancia;
    if ((masa1 <= 0) || (masa2 <= 0) || (distancia <= 0))
        cout << " no solucion\n";
    else
    {
        fuerza = G * masa1 * masa2 / (distancia * distancia);
        cout << " la solucion es: \n";
        cout << " Fuerza en dinas = " << fuerza << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Una ejecución es la siguiente:

```
Introduzca la masa de los dos cuerpos en gramos:  
4567 5267  
Introduzca la distancia entre ellos en centimetros:  
3  
la solucion es:  
Fuerza en dinas = 0.17835  
Presione una tecla para continuar . . .
```

3.5. Una codificación es la siguiente:

```
#include <cstdlib>  
#include <iostream>  
using namespace std;  
  
int main(int argc, char *argv[]){  
    float m, energia;  
  
    const float c = 2.997925e+10;  
    cout << " introduzca masa\n ";  
    cin >> m;  
    energia = c * m * m * m;  
    cout << " energia en ergios : " << energia;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

3.6. El programa lee el número de pies y realiza las transformaciones correspondientes de acuerdo con lo indicado.

```
#include <cstdlib>  
#include <iostream>  
using namespace std;  
  
int main(int argc, char *argv[]){  
    float pies, pulgadas, yardas, metros, centimetros;  
  
    cout << " Introduzca pies: \n ";  
    cin >> pies;  
    pulgadas = pies * 12;  
    yardas = pies / 3;  
    centimetros = pulgadas * 2.54;  
    metros = centimetros / 100;  
    cout << " pies " << pies << endl;  
    cout << " pulgadas " << pulgadas << endl;  
    cout << " yardas " << yardas << endl;  
    cout << " centimetros " << centimetros << endl;  
    cout << " metros " << metros << endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

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

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int hectometros, decametros, metros, decimetros;

    cout << " Introduzca hectometros, decametros y metros ";
    cin >> hectometros >> decametros >> metros;
    decimetros = ((hectometros * 10 + decametros) * 10 + metros) * 10;
    cout << " numero de decimetros es " << decimetros << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Un resultado de ejecución del programa anterior es:

```
Introduzca hectometros, decametros y metros 4 35 5
numero de decimetros es 7550
Presione una tecla para continuar . . . -
```

- 3.8.** Para obtener el menor número de billetes y monedas de euro basta con comenzar por el billete o moneda de más alto valor. Para obtener el número o cantidad que hay que tomar de esa moneda o billete, se calcula el cociente de la cantidad dividido por el valor. El resto de la división entera de la cantidad dividido por el valor es la nueva cantidad con la que hay que volver a hacer lo mismo, pero con el siguiente billete o moneda en valor. Si se considera el sistema monetario del euro, los billetes y monedas son: billetes (quinientos, doscientos, cien, cincuenta, veinte, diez y cinco euros); monedas (dos y un euro; cincuenta, veinte, diez, cinco, dos y un céntimo de euro). El programa lee la cantidad en euros en una variable real CantidadOriginal. La transforma a céntimos de euros y se realizan los cálculos. Posteriormente se visualizan los resultados.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int quinientos, doscientos, cien, cincuenta, veinte, diez;
    int cinco, dos, uno, cincuentac, veintec, diezc, cincoc, dosc, unc;
    float CantidadOriginal;
    long int cantidad;

    cout << "Introduzca cantidad en euros ";
    cin >> CantidadOriginal;
    CantidadOriginal*=100; // se pasa de euros con decimales a centimos
    cantidad = (int) CantidadOriginal; // se trunca a centimos de euro
    quinientos = cantidad / 50000; cantidad = cantidad % 50000;
    doscientos = cantidad / 20000; cantidad = cantidad % 20000;
    cien = cantidad / 10000; cantidad = cantidad % 10000;
    cincuenta = cantidad / 5000; cantidad = cantidad % 5000;
    veinte = cantidad / 2000; cantidad = cantidad % 2000;
```

```
diez = cantidad / 1000; cantidad = cantidad % 1000;
cinco = cantidad / 500; cantidad = cantidad % 500;
dos = cantidad / 200; cantidad = cantidad % 200;
uno = cantidad / 100; cantidad = cantidad % 100;
cincuentac = cantidad / 50; cantidad = cantidad % 50;
veintec = cantidad / 20; cantidad = cantidad % 20;
diezc = cantidad / 10; cantidad = cantidad % 10;
cincoc = cantidad / 5; cantidad = cantidad % 5;
dosc = cantidad / 2; cantidad = cantidad % 2;
unc = cantidad;
cout << " cambio en moneda con el menor numero " << endl;
cout << " cantidad original en centimos: " << CantidadOriginal << endl;
cout << " billetes de quinientos euros: " << quinientos << endl;
cout << " billetes de doscientos euros: " << doscientos << endl;
cout << " billetes de cien euros : " << cien << endl;
cout << " billetes de cincuenta euros: " << cincuenta << endl;
cout << " billetes de veinte euros: " << veinte << endl;
cout << " billetes de diez euros : " << diez << endl;
cout << " billetes de cinco euros: " << cinco << endl;
cout << " monedad de dos euros: " << dos << endl;
cout << " monedad de un euro: " << uno << endl;
cout << " monedas de cincuenta centimos de euros: " << cincuentac << endl;
cout << " monedad de veinte centimos de eruro: " << veintec << endl;
cout << " monedad de diez centimos de euro: " << diezc << endl;
cout << " monedas de cinco centimos de euros: " << cincoc << endl;
cout << " monedad de dos centimos de eruro: " << dosc << endl;
cout << " monedad de un centimo de euro: " << unc << endl;
system("PAUSE");
return EXIT_SUCCESS;
}
```

Una ejecución del programa anterior es:

```
Introduzca cantidad en euros 9988.88
cambio en monedad con el menor numero
cantidad original en centimos: 998888
billetes de quinientos euros: 19
billetes de doscientos euros: 2
billetes de cien euros : 0
billetes de cincuenta euros: 1
billetes de veinte euros: 1
billetes de diez euros : 1
billetes de cinco euros: 1
monedad de dos euros: 1
monedad de un euro: 1
monedas de cincuenta centimos de euros: 1
monedad de veinte centimos de eruro: 1
monedad de diez centimos de euro: 1
monedas de cinco centimos de euros: 1
monedad de dos centimos de eruro: 1
monedad de un centimo de euro: 1
Presione una tecla para continuar . . .
```

EJERCICIOS PROPUESTOS

- 3.1. Escribir un programa que acepte un año escrito en cifras arábigas y visualice el año escrito en números romanos, dentro del rango 1000 a 2100.

Nota: Recuerde que V = 5, X = 10, L = 50, C = 100, D = 500 y M = 1.000.

$$\begin{array}{lll} IV = 4 & XL = 40 & CM = 900 \\ MCM = 1900 & MCML = 1950 & MCMXL = 196 \\ MCMXL = 1940 & MCMLXXXIX = 1989 \end{array}$$

- 3.2. 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

- 3.3. 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 múltiples de 100 sólo son bisiestos cuando a la vez son múltiples de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo es).

- 3.4. 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 ?.

- 3.5. 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.

- 3.6. Escribir un programa que lea tres números y nos escriba el mayor y el menor.
- 3.7. Escribir un programa que solicite al usuario la longitud y anchura de una habitación y, a continuación, visualice su superficie y perímetro.
- 3.8. Escribir un programa que lea cuatro números y calcule la media aritmética.
- 3.9. Escribir un programa que lea el radio de un círculo y calcule su área, así como la longitud de la circunferencia de ese radio.
- 3.10. Escribir un programa que lea el radio y la altura de un cono y calcule su volumen y área total.
- 3.11. Escribir un programa que lea tres enteros de tres dígitos y calcule y visualice su suma y su producto. La salida será justificada a derecha.
- 3.12. Escribir un programa que lea tres 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.
- 3.13. 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 600 euros.
Impuestos 10 por 100, si el salario bruto es mayor de 600 euros.

CAPÍTULO 4

Estructuras de control selectivas (if, if-else, switch)

Introducción

Los programas definidos hasta este punto se ejecutan de modo secuencial. 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. 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*.

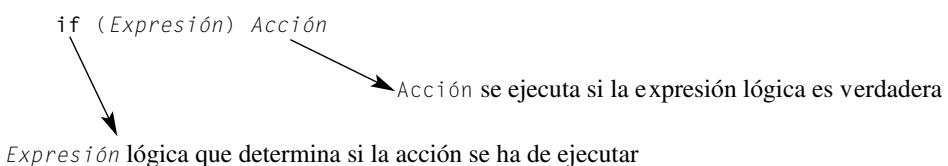
Este capítulo considera las *estructuras selectivas* o *condicionales* —sentencias `if` y `switch`— que controlan si una sentencia o lista de sentencias se ejecutan en función del cumplimiento o no de una condición. Para soportar estas construcciones, C++ tiene el tipo lógico `bool`.

4.1. 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*. Una **sentencia compuesta** es un conjunto de sentencias encerradas entre llaves (`{` y `}`) que se utiliza para especificar un flujo secuencial.

4.2. La sentencia if

En C++, la estructura de control principal de selección es una sentencia `if`. La sentencia `if` tiene dos alternativas o formatos posibles. El formato más sencillo tiene la sintaxis siguiente:



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

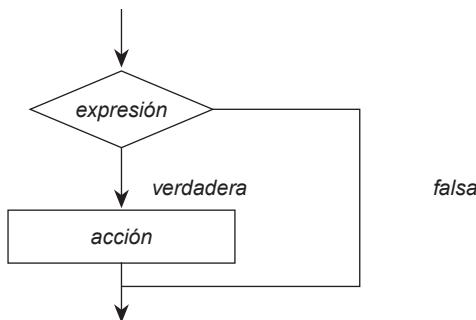


Figura 4.1. Diagrama de flujo de una sentencia básica *if*

EJEMPLO 4.1. Prueba de divisibilidad de dos números enteros.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int numero1, numero2;

    cout << "Introduzca dos enteros:" ;
    cin >> numero1 >> numero2;
    if (numero1 % numero2 == 0)
        cout << numero1 << " es divisible por " << numero2 << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
  
```

Resultados de ejecución del programa anterior

```

Introduzca dos enteros: 25 5
25 es divisible por 5.
  
```

EJEMPLO 4.2. Decidir si un número es mayor que 10.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float numero;

    cout << "Introduzca un número :" ;
    cin >> numero;
    // comparar número con diez
    if (numero > 10)
        cout << numero << "es mayor que 10" << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
  
```

EJEMPLO 4.3. Leer tres números enteros y visualizar el mayor.

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 teníamos, o bien es el nuevo.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int n1, n2, n3, mayor;

    cout << " introduzca tres numeros ";
    cin >> n1 >> n2 >> n3;
    mayor = n1;                                // candidato a mayor
    if (mayor < n2)
        mayor = n2;                            // nuevo mayor
    if (mayor < n3)
        mayor = n3;                            // nuevo mayor
    cout << " el mayor es :" << mayor << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultados de ejecución del programa anterior:

```
introduzca tres numeros 2 7 4
el mayor es :7
Presione una tecla para continuar . . .
```

EJEMPLO 4.4. Lee un dato real y visualiza su valor absoluto.

```
include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float Dato;

    cout << "Introduzca un numero: ";
    cin >> Dato;
    if (Dato < 0)
        Dato = - Dato;                      //Cambio de signo
    cout << " Valor absoluto siempre positivo " << Dato << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

4.3. Sentencia if de dos alternativas: if-else

El formato de la sentencia if-else 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*.

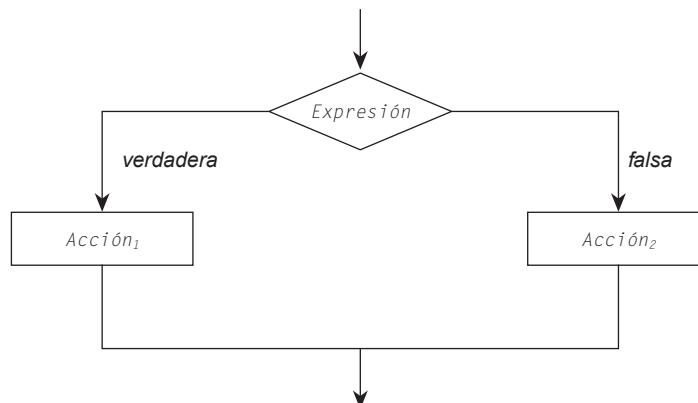


Figura 4.2. Diagrama de flujo de la representación de una sentencia if-else

EJEMPLO 4.5. Leer una nota, y visualizar baja si es menor que 100 y alta en otro caso.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int nota;

    cout << " dame nota: ";
    cin >> nota;
    if (nota < 100)
        cout << " Baja ";
    else
        cout << "Alta";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJEMPLO 4.6. Leer el salario y los impuestos. Visualizar el salario neto.

```

        Salario_neto = Salario_bruto;
        cout << Salario_neto << Salario_bruto;           // visualización
        system("PAUSE");
        return EXIT_SUCCESS;
    }
}

```

4.4. Sentencias if-else anidadas

Una sentencia `if` es anidada cuando la sentencia de la rama *verdadera* o la rama *falsa*, es a su vez es 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)
    sentencian;
else
    sentencia;
}

```

EJEMPLO 4.7. Leer la calificación (*nota*) en una variable real, y mediante `if` anidados escribir el resultado:

<i>Menor que 0 o mayor que 10</i>	Error en nota
<i>0 ≤ nota < 5.0</i>	Suspensos
<i>5.0 ≤ nota < 6.5</i>	Aprobado
<i>6.5 ≤ nota < 8.5</i>	Notable
<i>8.5 ≤ nota < 10</i>	Sobresaliente
<i>10</i>	Matrícula de honor
<i>< 0 o > 10</i>	Error en nota

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float nota;
    cout << "dame nota: ";
    cin >> nota;
    if( ( nota < 0.0 ) || ( nota > 10 ) )
        cout << "Error en nota ";
    else if ( nota < 5.0 )
        cout << "Suspensos";
    else if( nota < 6.5 )
        cout << "Aprobado";
    else if ( nota < 8.5)
        cout << "Notable";
}

```

```

    else if ( nota < 10)
        cout <<"Sobresaliente";
    else
        cout <<"Matricula de Honor";
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

4.5. Sentencia de control switch

La sentencia `switch` es una sentencia C++ que se utiliza para hacer una selección entre múltiples alternativas.

Sintaxis

```

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

```

La expresión `selector` debe ser un tipo ordinal (`int`, `char`, `bool` pero no `float` o `string`). 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, `etiquetai`— entonces la ejecución comenzará con la primera sentencia de la secuencia `secuenciai` y continuará hasta que se encuentra el final de la sentencia de control `switch`, o hasta encontrar la sentencia `break`.

EJEMPLO 4.8. Sentencia switch para informar sobre la lectura de una opción dentro de un rango.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int opcion;

    cout << "introduzca opcion entre 0 y 3:";
    cin >> opcion;
    switch (opcion)
    {
        case 0:
            cout << "Cero!" << endl;
            break;
        case 1:
            cout << "Uno!" << endl;
            break;
    }
}

```

```

    case 2:
        cout << "Dos!" << endl;
        break;
    case 3:
        cout << "Tres!" << endl;
        break;
    default:
        cout << "Fuera de rango" << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 4.9. *Sentencia switch con caracteres.*

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char nota;

    cout << "Introduzca calificación (S, A, B, N, E) :";
    cin >> nota;
    switch (nota)
    {
        case 'E': cout << "Sobresaliente.";
                    break;
        case 'N': cout << "Notable.";
                    break;
        case 'B': cout << "Bien.";
                    break;
        case 'A': cout << "Aprobado.";
                    break;
        case 'S': cout << "Suspens.";
                    break;
        default:
            cout << "no es posible esta nota";
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

4.6. Expresiones condicionales: el operador ?:

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

Sintaxis

<code>condición ? expresión₁: expresión₂</code>

`condición`
`expresión1 | expresión2`

es una expresión lógica
son expresiones compatibles de tipos

Se 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 4.10. *Sentencia ?: para decidir el orden de dos números.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int n1, n2;

    cout << " introduzca dos numeros ";
    cin >> n1 >> n2 ;
    n1 > n2 ? cout << n1 << " > " << n2
              : cout << n1 << " <= " << n2;
    system("PAUSE");
    return EXIT_SUCCESS;
}

#include <cstdlib>
#include <iostream>
using namespace std;
```

EJEMPLO 4.11. *Escribe el mayor de dos números usando ?:*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int n1, n2, mayor;

    cout << " introduzca dos numeros ";
    cin >> n1 >> n2 ;
    mayor = n1 > n2 ? n1 : n2;
    cout << " el mayor es: " << mayor << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución

```
introduzca dos numeros 5 ?
el mayor es: 5
Presione una tecla para continuar . . .
```

4.7. 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 la 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. Esta característica permite, en general, disminuir el tiempo de ejecución.

EJERCICIOS

- 4.1. ¿Qué errores de sintaxis tiene la siguiente sentencia?

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

- 4.2. ¿Qué valor se 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;
```

- 4.3. ¿Qué salida producirá el código siguiente, cuando se inserta en un programa completo?

```
int primera_opcion = 1;
switch (primera_opcion + 1)
{
    case 1:
        cout << "Cordero asado\n";
        break;
    case 2:
        cout << "Chuleta lechal\n";
        break;
    case 3:
        cout << "Chuletón\n";
    case 4:
        cout << "Postre de pastel\n";
        break;
    default:
        cout << "Buen apetito\n";
}
```

- 4.4. ¿Qué salida producirá el siguiente código, cuando se inserta en un programa completo?

```
int x = 2;

cout << "Arranque\n";
if (x <= 3)
    if (x != 0)
        cout << "Hola desde el segundo if.\n";
    else
        cout << "Hola desde el else.\n";
```

```

cout << "Fin\n";
cout << "Arranque de nuevo\n";
if (x > 3)
    if (x != 0)
        cout << "Hola desde el segundo if.\n";
    else
        cout << "Hola desde el else.\n";
cout << "De nuevo fin\n";

```

- 4.5.** 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.

- 4.6.** ¿Cuál es la salida de este segmento de programa?

```

int x = 1;

cout << x << endl;
{
    cout << x << endl;
    int x = 2;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 3;
        cout << x << endl;
    }
    cout << x << endl;
}

```

- 4.7.** 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

- 4.8.** 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 múltiples 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).

PROBLEMAS

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

- 4.2.** 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:

Media	Puntuación
[90-100]	A
[80-90)	B
[70-80)	C
[60-70)	D
[0-60)	E

- 4.3.** 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 300 euros.
 Impuestos 10 por 100, si el salario bruto es mayor de 300 euros.

- 4.4.** Escribir un programa que lea dos números enteros y visualice el menor de los dos.
- 4.5.** Escribir y comprobar un programa que resuelva la ecuación cuadrática ($ax^2 + bx + c = 0$).
- 4.6.** Escribir un programa que lea tres enteros y emita un mensaje que indique si están o no en orden numérico.
- 4.7.** Escribir un programa que lea los valores de tres lados posibles de un triángulo a , b y c , y calcule en el caso de que formen un triángulo su área y su perímetro, sabiendo que su área viene dada por la siguiente expresión:

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

donde p es el semiperímetro del triángulo $p = (a + b + c)/2$

- 4.8.** Escribir y ejecutar un programa que simule un calculador simple. Lee dos enteros y un carácter. Si el carácter es un $+$, se visualiza la suma; si es un $-$, se visualiza la diferencia; si es un $*$, se visualiza el producto; si es un $/$, se visualiza el cociente; y si es un $\%$ se imprime el resto.
- 4.9.** Escribir un programa que calcule los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.

SOLUCIÓN DE LOS EJERCICIOS

- 4.1.** 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 $y = x$.

- 4.2.** Si velocidad toma el valor de 120 entonces necesariamente consumo debe tomar el valor de 10.00, ya que se evalúa la primera condición y es cierta por lo que se ejecuta la sentencia $consumo = 10.00$;
- 4.3.** Aparece escrito Chuleta lechal. Si a primera opción se le asignara el valor de 2 entonces aparece escrito Chuletón y en la siguiente línea Postre de pastel, ya que case 3: no lleva la orden break.
- 4.4.** La salida del programa es:

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

4.5. Una codificación es la siguiente:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int nota;

    cout << " dame nota: ";
    cin >> nota;
    if (nota < 100)
        cout << " Baja ";
    else if (nota > 100)
        cout << "Alta";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

4.6. La salida del programa es:

1
1
2
2
3
2

4.7. Una codificación es la siguiente:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x;

    cout << " dato ";
    cin >> x;
    if (x < 0)
        cout << "es negativo\n";
    else if (x <= 100)
        cout << "0 <= x = %d <= 100";
    else
        cout << " > 100";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- 4.8.** La variable booleana `bisiesto` se pone a `true` si el año es bisiesto. Esto ocurre cuando es divisible el año por 400, o es divisible por 4 y no por 100.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x;
    bool bisiesto;

    cout << " introduzca año entero ";
    cin >> x;
    if (x % 400 == 0)
        bisiesto = true;
    else if (x % 100 == 0)
        bisiesto = false;
    else
        bisiesto =(x % 4 == 0);
    if (bisiesto)
        cout << x << " es bisiesto\n";
    else
        cout << x << " no es un año bisiesto\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

SOLUCIÓN DE LOS PROBLEMAS

- 4.1.** Para resolver el problema, se ha de 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 se decide si el año es bisiesto de acuerdo con lo indicado en el Ejercicio resuelto 4.8 para saber si el mes de febrero tiene 28 o 29 días. El resto de los meses tiene 31 días excepto abril, junio, septiembre y noviembre que corresponden a los meses 4, 6, 9 y 11.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int mes, ano;

    bool bisiesto;
    cout << " introduzca mes entre 1 y 12 ";
    cin >> mes;
    cout << " introduzca año entero ";
    cin >> ano;
```

```

if (x % 400 == 0)
    bisiesto = true;
else if (ano % 100 == 0)
    bisiesto = false;
else
    bisiesto = (ano % 4 == 0);
if (mes == 2)
    if(bisiesto)
        cout << " tiene 29 dias\n";
    else
        cout << " tiene 28 dias\n";
    else
        if((mes == 4) || (mes == 6) || (mes == 9) || (mes == 11))
            cout << " tiene 30 dias \n";
        else
            cout <<" tiene 31 dias \n";
system("PAUSE");
return EXIT_SUCCESS;
}

```

- 4.2.** 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 indicada usando if anidados.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int nota1, nota2, nota3, nota4;
    float media;

    cout << "Dame nota 1 ";
    cin >> nota1;
    cout << "Dame nota 2 ";
    cin >> nota2;
    cout << "Dame nota 3 ";
    cin >> nota3;
    cout << "Dame nota 4 ";
    cin >> nota4;
    media = (float)(nota1 + nota2 + nota3 + nota4) / (float)4;
    if(( media < 0) || ( media > 100 ))
        cout << "fuera de rango ";
    else if( media >= 90)
        cout << " media = " << media << " A";
    else if(media >= 80)
        cout << "media = " << media << " B";
    else if(media >= 70)
        cout << "media = " << media << " C";
    else if(media >= 60)
        cout << "media = " << media << " D";
}

```

```

    else
        cout << "media = " << media << "E";
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 4.3.** Se escribe un programa que lee las Horas, la Tasa, y calcula las horas extras, así como el SalarioBruto y el SalarioNeto de acuerdo con la especificación, visualizando los resultados.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float Horas, Extras, Tasa, SalarioBruto, SalarioNeto;

    cout << " dame Horas\n";
    cin >> Horas;
    if ( Horas <= 38 )
        Extras = 0;
    else
    {
        Horas = 38;
        Extras = Horas - 38;
    }
    cout <<"introduzca Tasa\n";
    cin >> Tasa;
    SalarioBruto = Horas * Tasa + Extras * Tasa * 1.5;
    if (SalarioBruto < 50000.0)
        SalarioNeto = SalarioBruto;
    else
        SalarioNeto = SalarioBruto * 0.9;
    cout <<" Salario bruto " << SalarioBruto << endl;
    cout <<" Salario neto " << SalarioNeto << endl ;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Un resultado de ejecución es el siguiente:

```

dame Horas
45
introduzca Tasa
25
Salario bruto 950
Salario neto 855
Presione una tecla para continuar . . .

```

- 4.4.** Se solicitan los dos números. Si numero1 es menor que numero2, la condición es “verdadera” (true); en caso contrario la condición es “falsa” (false). De este modo se visualiza numero1 cuando es menor que numero2.

```

#include <cstdlib>
#include <iostream>

```

```

using namespace std;
int main(int argc, char *argv[])
{
    int numero1, numero2;

    cout << "Introduzca dos enteros:";
    cin >> numero1 >> numero2;
    if (numero1 < numero2)
        cout << numero1 << endl;
    else
        cout << numero2 << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

4.5. Para resolver el problema se ha tenido en cuenta que:

- Si $a \neq 0$ se presentan tres casos: el primero con dos soluciones dadas por la fórmula que da la solución de la ecuación de segundo grado cuando el discriminante $d = b^2 - 4ac$ es positivo $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. El segundo con una solución dada por la fórmula cuando el discriminante es cero $x = \frac{-b}{2a}$. El tercero con dos soluciones complejas, dadas por la fórmula $\frac{-b}{2a} \pm i \frac{\sqrt{b^2 - 4ac}}{2a}$ y $\frac{-b}{2a} \mp i \frac{\sqrt{b^2 - 4ac}}{2a}$ cuando el discriminante es negativo.
- Si $a = 0$ se presentan a su vez otros tres casos: el primero es cuando $b \neq 0$ cuya solución es $x = -\frac{c}{b}$. 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.

```

#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char *argv[])
{
    float a, b, c, d, x1, x2;

    cout << "introduzca los tres coeficientes\n";
    cin >> a >> b >> c;
    if (a != 0)
    {
        d = b * b - 4 * a * c;
        if (d > 0)
        {
            cout << " dos soluciones reales y distintas\n";
            x1 = (-b + sqrt(d)) / (2 * a);
            x2 = (-b - sqrt(d)) / (2 * a);
            cout << " x1= " << x1 << " x2= " << x2 << endl;
        }
        else if (d == 0)
        {
            cout << " dos soluciones reales e iguales\n";
            x1 = (-b) / (2 * a);
            cout << " x= " << x1;
        }
    }
}

```

```

        else
    {
        cout << " no tiene solucion real\n";
        cout << " tiene dos soluciones complejas \n";
        x1 = -b / (2 * a);
        x2 = sqrt(-d) / (2 * a);
        cout << " primera solucion\n";
        cout << " parte real "<< x1 << endl;
        cout << " parte imaginaria "<< x2 << endl;
        cout << " segunda solucion\n";
        cout << " parte real "<< x1 << endl;
        cout << " parte imaginaria "<< -x2 << endl;
    }
}

else if (b != 0)
    cout << " una solucion simple= " << -c / b;
else if (c == 0)
    cout << " se introdujo la identidad 0 = 0\n";
else
    cout << " sin solucion\n";
system("PAUSE");
return EXIT_SUCCESS;
}

```

- 4.6.** La codificación usa la sentencia ?: y además la sentencia de selección if.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x, y, z;
    bool ordenados;

    cin >> x >> y >> z;
    ordenados = x >= y ? true : false;
    ordenados = ordenados && (y >= z ? true : false);
    if (ordenados)
        cout << " estan ordenados" << endl;
    else
        cout << " no ordenados " << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 4.7.** Para que el triángulo existe 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 obtenido comprueba que los datos leídos cumplen las condiciones, y escribe en caso de formar un triángulo su área y su perímetro.

```

#include <cstdlib>
#include <iostream>
#include <math.h>           // contiene la función pow
using namespace std;

```

```

int main(int argc, char *argv[])
{
    float a, b, c, p, area;

    cout << "Introduzca el valor de los tres lados";
    cin >> a >> b >> c;
    if ((a <= 0) || (b <= 0) || (c <= 0) ||
        ((a + b) < c) || ((a + c) < b) || ((b + c) < a))
        cout << " Los lados no dan un triángulo \n";
    else
    {
        p =(a + b + c)/ 2;
        area = pow(p * ( p - a ) * ( p - b ) * ( p - c ), 0.5);
        cout << "la solucion es\n";
        cout << "area = " << area << endl;
        cout << " perimetro = " << p * 2 << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 4.8.** El programa lee los dos operadores, el operando y visualiza la salida de acuerdo con lo solicitado.

```

#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char *argv[])
{ int operando1, operando2;
char operador;

cout << " Introduzca dos numeros enteros ";
cin >> operando1 >> operando2;
cout << " Introduzca operador + - * / % ";
cin >> operador;
switch(operador)
{
    case '+': cout << operando1 + operando2;
                 break;
    case '-': cout << operando1 - operando2;
                 break;
    case '*': cout << operando1 * operando2;
                 break;
    case '/': cout << operando1 / operando2;
                 break;
    case '%': cout << operando1 % operando2;
                 break;
    default: cout << " fuera de rango";
}
system("PAUSE");
return EXIT_SUCCESS;
}

```

- 4.9.** Se calcula la hipotenusa por la fórmula del teorema de pitágoras, y se obtiene el ángulo mediante la función `inverso del seno` que es `asin()` que se encuentra en `math.h`. Además, se convierte el valor de vuelto por la función arco seno a grados (la función arco seno da su resultado en radianes).

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#define pi 3.141592
using namespace std;

int main(int argc, char *argv[])
{
    float a, b, h, angulo;

    cout << "Introduce los lados ";
    cin >> a >> b;
    if ((a <= 0) || (b <= 0))
        cout << " no solucion\n";
    else
    {
        h = sqrt(a * a + b * b);
        angulo = 180 / pi * asin(a / h); // ángulo en grados
        cout << " hipotenusa = " << h << endl;
        cout << "     angulo = " << angulo << endl;
        cout << "otro angulo = " << 90 - angulo << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Un resultado de ejecución es:

```
Introduce los lados 5 6
hipotenusa = 7.81025
    angulo = 39.8056
otro angulo = 50.1944
```

EJERCICIOS PROPUESTOS

- 4.1.** 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.

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

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

```
if (x = 0) cout << x << " = 0\n";
else cout << x << " != 0\n";
```

- 4.3.** ¿Cuál es el error del siguiente código?

```
if (x < y < z) cout << x << "<" << y << "<"
    << z << endl;
```

- 4.4.** ¿Cuál es el error de este código?

```
cout << "Introduzca n:";  
cin >> n;  
if (n < 0)  
    cout << "Este número es negativo. Pruebe  
    de nuevo .\n";
```

```
cin >> n;  
else  
    cout << "conforme. n= " << n << endl;
```

- 4.5.** Determinar si el carácter asociado a un código introducido por teclado corresponde a un carácter alfabetico, dígito, de puntuación, especial o no imprimible.

PROBLEMAS PROPUESTOS

- 4.1.** 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:

$$\begin{aligned} A &= \text{año} \bmod 19 \\ B &= \text{año} \bmod 4 \\ C &= \text{año} \bmod 7 \\ D &= (19 * A + 24) \bmod 30 \\ E &= (2 * B + 4 * C + 6 * D + 5) \bmod 7 \\ N &= (22 + D + E) \end{aligned}$$

donde N indica el número de día del mes de marzo (si N es igual o menor que 30) o abril (si es mayor que 31). Construir un programa que determine fechas de dominigos de Pascua.

- 4.2.** Construir un programa que indique si un número introducido por teclado es positivo, igual a cero, o negativo.
- 4.3.** 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.

- 4.4.** 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.
- 4.5.** 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.
- 4.6.** Escribir un programa que acepte un número de tres dígitos escrito en palabras 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.
- 4.7.** 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.

CAPÍTULO 5

Estructuras de control repetitivas (while, for, do-while)

Introducción

En este capítulo se estudian las *estructuras de control iterativas* o *repetitivas* que realizan la 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.

5.1. La sentencia while

Un bucle while tiene una *condición* del bucle (expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que un bucle while es un bucle *pretest* de modo que cuando se ejecuta el mismo, se evalúa la condición *antes* de que se ejecute el cuerpo del bucle.

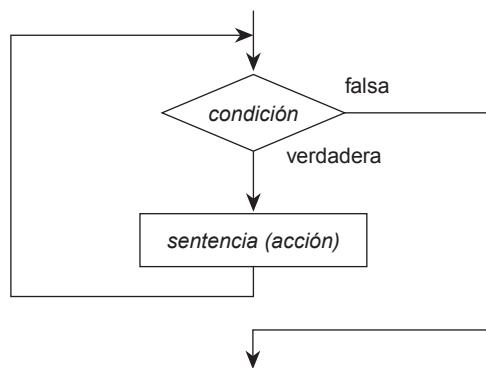


Figura 5.1. Diagrama del bucle while

Sintaxis

```
1 while (condición_bucle)
    sentencia; → cuerpo
```

```

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

```

cuerpo

Las sentencias del cuerpo del bucle se repiten **mientras** que la expresión lógica (condición del bucle) sea verdadera. Cuando se evalúa la expresión lógica y resulta falsa, se termina y se *sale* del bucle y se ejecuta la siguiente sentencia de programa después de la sentencia `while`.

EJEMPLO 5.1. Bucle **mientras** para escribir de los números del 1 al 10. En cada iteración se escribe el símbolo carácter `X`, el contenido de `x`, se incrementa `x` en una unidad y se salta de línea. Comienza el bucle con el valor de `x` en 1 y se sale del bucle con el valor de `x` a 11, pero el último valor escrito es 10, ya que el incremento del valor de `x` en una unidad se realiza después de haber sido escrito el valor de `x`.

```

int x = 1;

while ( x <= 10)
    cout <<"X: " << x++ << endl;

```

EJEMPLO 5.2. Bucle infinito. El contador se inicializa a 1 (menor de 100) y como `contador--` decrementa en 1 el valor de contador en cada iteración, el valor del contador nunca llegará a valer 100, que es el valor necesario para que la condición del bucle sea falsa.

```

int contador = 1;
while (contador < 100)
{
    cout << contador << endl;
    contador--;                                //decrementa en 1 contador
}

```

Bucles controlados por contadores

Son bucles en los cuales la variable de control `contador` se incrementa o decrementa en cada iteración en una cantidad constante. La variable de control `contador` se inicializa antes de comenzar el bucle a un valor. Se comprueba el valor de `contador` antes de que comience la repetición de cada bucle, y en cada iteración o pasada se incrementa o decrementa en una cantidad constante.

EJEMPLO 5.3. La suma de la serie $1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/50$ se realiza mediante el uso de un contador `n`. Por cada valor de la variable `contador n` del rango 1 hasta 50 se ejecuta la sentencia de acumular en `suma` el valor de $1/n$. El fragmento de programa siguiente inicializa el acumulador `suma` a 0, el contador `n` a 1 posteriormente realiza la suma mediante el contador real `n` (obliga a que el cociente $1/n$ sea real) y presenta el resultado.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{

```

```

float suma = 0, n = 1;

while (n <= 10)
{
    // n es real para que el cociente 1/n sea en coma flotante
    suma += 1 / n;
    n++;
}
cout << suma ;
system("PAUSE");
return EXIT_SUCCESS;
}

```

El resultado de ejecución del programa anterior es: 2.92897

EJEMPLO 5.4. El bucle adecuado para resolver la tarea de sumar los enteros del intervalo 11..50, es un bucle controlado por un contador *n*. Se inicializa el acumulador *suma* a 0. El contador entero *n* se inicializa a 11, se incrementa en cada iteración en una unidad hasta llegar a 50. El contador *n* se acumula en el acumulador *suma* en cada iteración del bucle *while* para los valores del rango comprendido entre 11 y 50.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int suma = 0, n = 11;

    while (n <= 50)
    {
        suma = suma + n;
        n++;
    }
    cout << suma ;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

La ejecución del programa anterior da como resultado: 1220.

Bucles controlados por centinelas

Un centinela es un valor definido y especificado que sirve para terminar el proceso del bucle. Este valor debe ser elegido con cuidado por el programador para que no sea un posible dato y además no afecte al normal funcionamiento del bucle.

EJEMPLO 5.5. Leer las notas de un alumno usando como valor centinela para la entrada de notas el valor de -1. Se define una constante entera *centinela* con el valor de -1. En la variable *nota* se leen los datos de la entrada. La variable *contador* cuenta el número total de notas introducidas, y el acumulador *suma* contiene la suma total de las notas introducidas. El bucle *while* está controlado por el valor de *centinela*. En cada iteración se incrementa el contador de en una unidad, se lee una nueva *nota* y se acumula en *suma*. Obsérvese que la variable *contador*, siempre contiene una unidad menos del número de datos que han sido introducidos.

```

#include <cstdlib>
#include <iostream>
using namespace std;

```

```

int main(int argc, char *argv[])
{
    const int centinela = -1;
    float nota, contador = 0, suma = 0;

    cout << "Introduzca siguiente nota -1 centinela: ";
    cin >> nota;
    while (nota != centinela)
    {
        contador++;
        suma += nota;
        cout << "Introduzca la siguiente nota: -1 centinela: ";
        cin >> nota;
    }                                         // fin de while
    if (contador > 0)
        cout << "media = " << suma / contador << endl;
    else
        cout << " no hay notas ";
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Bucles controlados por indicadores (banderas)

Las variables tipo `bool` se utilizan como indicadores o *banderas de estado*. El valor del indicador se inicializa (normalmente a `false`) antes de la entrada al bucle y se redefine (normalmente a `true`) cuando un suceso específico ocurre dentro del bucle. Un *bucle controlado por bandera-indicador* se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

EJEMPLO 5.6. Se leen repetidamente caracteres del teclado y se detiene , cuando se introduce un dígito. Se define una bandera `digito_leido` que se inicializa a `false`, y se cambia al valor de `true` cuando se lee un dígito. El bucle que resuelve el problema está controlado por la bandera `digito_leido`, y en cada iteración solicita un carácter, se lee en la variable `car`, y si es un dígito cambia el valor de la bandera. Al final del bucle se escribe el dígito leído.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char car;
    bool digito_leido = false;           // no se ha leído ningún dato

    while (!digito_leido)
    {
        cout << "Introduzca un carácter dígito para salir del bucle :";
        cin >> car;
        digito_leido = (('0' <= car) && (car <= '9'));          // fin de while
    }
    cout << car << " es el dígito leído" << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

La sentencia break en los bucles

La sentencia `break` se utiliza, a veces, para realizar una terminación anormal del bucle. Dicho de otro modo, una terminación antes de lo previsto. Su sintaxis es: `break;`

EJEMPLO 5.7. *El siguiente código extrae y visualiza valores enteros de la entrada hasta que se encuentra un valor entero especificado, previamente leído del teclado.*

```
int Clave;
int Entrada_entera;

cin >> Clave;
while (cin >> Entrada_entera)
{
    if (Entrada_entera != Clave)
        cout << Entrada_entera << endl;
    else
        break;
} //Salida del bucle
```

5.2. 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.

Sintaxis

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

El bucle `for` contiene las cuatro partes siguientes:

- La parte de *Inicialización* inicializa las variables de control del bucle.
- La parte de *Condición de Iteración* contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias.
- La parte de *Incremento* incrementa la variable o variables de control del bucle.
- Las *Sentencias*, acciones o sentencias que se ejecutarán por cada iteración del bucle.

La sentencia `for` es equivalente al siguiente código `while`:

```
inicialización;
while (condiciónIteración)
{
    sentencias del bucle for
    incremento;
}
```

EJEMPLO 5.8. *Bucle for ascendente que escribe los 5 primeros números naturales, su cuadrado y su cubo. Se inicializa la variable entera n a 1 y mientras el valor de n sea menor o igual a 5 se escribe el número n, su cuadrado y su cubo. Posteriormente se incrementa en una unidad el valor de n.*

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

```

int main(int argc, char *argv[])
{
    cout << "n      n*n      n*n*n" << endl;
    for (int n = 1; n <= 5; n++)
        cout << n << '\t' << n * n << '\t' << n * n * n << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Resultados de ejecución:

n	n*n	n*n*n
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125

EJEMPLO 5.9. Bucle for descendente que escribe números reales y su raíz cuadrada. Se inicializa la variable *n* a 16. En cada iteración del bucle se decremente *n* en 2.5. El bucle termina cuando *n* es menor que 1.

```

#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char *argv[])
{
    float n;
    cout << "n      raiz(n)" << endl;
    for (n = 16; n >= 1 ; n = n - 2.5)
        cout << n << '\t' << sqrt (n) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Resultados de ejecución:

n	raiz(n)
16	4
13.5	3.67423
11	3.31662
8.5	2.91548
6	2.44949
3.5	1.87083
1	1

EJEMPLO 5.10. Bucle for que no termina nunca. La salida del bucle se realiza con la sentencia break. Cuando se cumple la condición de salida del bucle. El ejemplo suma los *vmax* primeros términos de la serie $\sum_{c=1}^{v_{\max}} \frac{1}{c * c}$, siendo *vmax* un dato de programa. En cada iteración se incrementa el contador *c* en una unidad, se acumula en *suma* el valor del término de la serie $\frac{1}{c * c}$. La condición if decide si hay que sumar el término $\frac{1}{c * c}$ a la serie o si hay que terminar mediante la ejecución de la sentencia break. El programa inicializa el contador *c* y el acumulador *suma* a 0 y en cada iteración del bucle, cada vez que se suma un término a la serie presenta los valores del contador *c* y de la suma parcial del acumulador *suma*.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int c = 0;
    float suma = 0 ;
    int vmax;
    cout << "Cuantos terminos sumo de la serie ? ";
    cin >> vmax;
    for (;;) // bucle for que no termina nunca
    {
        if(c <= vmax) //test
        {
            c++; //incremento
            suma +=1/(float)(c*c);
            cout << c << " " << suma << endl;
        }
        else
            break;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución del programa anterior:

```
Cuantos terminos sumo de la serie ? 4
1 1
2 1.25
3 1.36111
4 1.42361
```

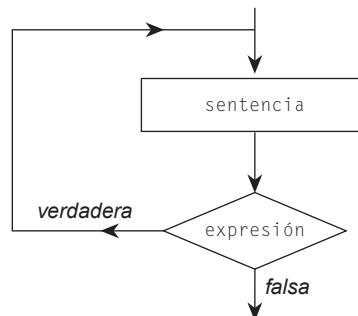
5.4. Repetición: el bucle do...while

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

Sintaxis

```
do
    sentencia
    while (expresión)
```

Semántica



Después de cada ejecución de sentencia se evalúa expresión. Si es verdadera se repite el cuerpo del bucle (sentencia). Si es falsa, se termina el bucle y se ejecuta la siguiente sentencia.

Figura 5.2. Diagrama de flujo de la sentencia do

EJEMPLO 5.11. Bucle que escribe las letras mayúsculas del alfabeto. Se inicializa la variable carácter `car` a 'A', y mediante un bucle `do while` que termina cuando en `car` hay un carácter mayor que 'Z', se itera escribiendo el valor de `car` e incrementando el valor de `car` en una unidad por lo que `car` toma el siguiente carácter del código ASCII.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char car = 'A';

    do
    {
        cout << car << ' ';
        car++;
    } while (car <= 'Y');
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de la ejecución:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y
```

5.5. Comparación de bucles while, for y do-while

C++ proporciona tres sentencias para el control de bucles: `while`, `for` y `do-while`. El bucle `while` se repite *mientras* la condición de repetición del bucle sea verdadera; el bucle `for` se utiliza normalmente cuando el conteo esté implicado, o bien cuando el número de iteraciones requeridas se puede determinar al principio de la ejecución del bucle. El bucle `do-while` se ejecuta de un modo similar a `while` excepto que las sentencias del cuerpo del bucle se ejecutan siempre al menos una vez.

EJEMPLO 5.12. Se escriben los números 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, con un bucle `while`, con un bucle `for`, y con un bucle `do while` en un mismo programa.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int num = 10;

    while (num <= 100) //con bucle while
    {
        cout << num << " ";
        num += 10;
    }
    cout << endl << endl; // con bucle for
    for (num = 10; num <= 100; num += 10)
        cout << num << " ";
    cout << endl << endl;
```

```
num = 10;                                //con bucle do while
do
{
    cout << num << " ";
    num += 10;
}
while (num <= 100);
system("PAUSE");
return EXIT_SUCCESS;
}
```

EJEMPLO 5.13. Leer un número entero positivo en un bucle do while y calcular su factorial, mediante un bucle for, un bucle while y un bucle do while. (Nota: Factorial de $n = n * (n - 1) * (n - 2) * \dots * 2 * 1$).

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int numero, i, factorial;

    do
    {
        cout << "dame numero entero: ";
        cin >> numero;
    } while (numero <= 0);
    for( factorial = 1, i = 1; i <= numero; i++)           //con bucle for
        factorial *= i;
    cout << factorial << endl;
    factorial = 1;                                         //con bucle while
    i = 1;
    while( i < numero)
    {
        i++;
        factorial *= i ;
    }
    cout << factorial << endl;
    factorial = 1;                                         //con bucle do-while
    i = 0 ;
    do
    {
        i++;
        factorial *= i;
    } while(i < numero);
    cout << factorial << endl;;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

5.6. Bucles anidados

Los bucles *anidados* constan de un bucle externo con uno o más bucles internos. Cada vez que se repite el bucle externo, los bucles internos iteran reevaluándose las componentes de control y ejecutándose las iteraciones requeridas.

EJEMPLO 5.14. El siguiente programa muestra dos bucles *for* anidados que presentan las tablas de multiplicar del 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, pero de forma inversa. Para cada valor de la variable *n* en el rango 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, se ejecuta una orden de escritura y el bucle interno *for* controlado por la variable entera *m* que toma los valores 10, 9, 8, 7, ..., 1, escribiendo en cada iteración los valores de *n*, *m* y su producto *n * m*.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int n, m;

    for (n = 1; n <= 10; n++)
    {
        cout << " tabla de multiplicar del " << n << endl;
        for (m = 10; m >= 1; m--)
            cout << n << " veces " << m << " = " << n * m << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJERCICIOS

5.1. ¿Cuál es la salida del siguiente segmento de programa?

```
for (int cuenta = 1; cuenta < 5; cuenta++)
    cout << (2 * cuenta) << " ";
```

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

a) `for (int n = 10; n > 0; n = n - 2)`

```
{
    cout << "Hola ";
    cout << n << endl ;
}
```

b) `for (double n = 2; n > 0; n = n - 0.5)`

```
cout << m << " ";
```

5.3. Considerar el siguiente código de programa.

```
using namespace std;

int main(int argc, char *argv[])
{
    int i = 1, n;
    cin >> n;
    while (i <= n)
        if ((i % n) == 0)
            ++i;
    cout << i << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- a) ¿Cuál es la salida si se introduce como valor de n, 0?
- b) ¿Cuál es la salida si se introduce como valor de n, 1?
- c) ¿Cuál es la salida si se introduce como valor de n, 3?

5.4. Suponiendo que $m = 3$ y $n = 5$ ¿Cuál es la salida de los siguientes segmentos de programa?

```
a) for (int i = 0; i < n; i++)
{
    for (int j = 0; j < i; j++)
        cout << "*";
    cout << endl;
}

b) for (int i = n; i > 0; i--)
{
    for (int j = m; j > 0; j--)
        cout << "*";
    cout << endl;
}
```

5.5. ¿Cuál es la salida de este bucle?

```
int i = 1;
while (i * i < 10)
{
    int j = i;
    while (j * j < 100)
    {
        cout << i + j << " ";
        j *= 2;
    }
    i++;
    cout << endl;
}
cout << "\n*****\n";
```

PROBLEMAS

5.1. Escriba un programa que calcule y visualice $1 + 2 + 3 + \dots + (n-1) + n$, donde n es un valor de un dato positivo.

5.2. Escribir un programa que visualice la siguiente salida:

```

1
1      2
1      2      3
1      2      3      4
1      2      3
1      2
1
1

```

5.3. Diseñar e implementar un programa que lea un total de 10 números y cuente el número de sus entradas que son positivos, negativos y cero.

5.4. Diseñar e implementar un programa que solicite a su usuario un valor no negativo n y visualice la siguiente salida ($n = 6$):

```

1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

```

5.5. Escribir 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 límite máximo.

5.6. Diseñar un algoritmo que sume los $m = 30$ primeros números pares.

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

5.8. Escribir un programa que presente los valores de la función $\cos(3x) - 2x$ para los valores de x igual a 0, 0.5, 1.0, ... 4.5, 5.

5.9. Escribir y ejecutar un programa que invierta los dígitos de un entero positivo dado leído del teclado.

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

5.11. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de n números ($n > 0$).

5.12. Encontrar un número natural n más pequeño tal que la suma de los n primeros términos de la serie $\sum_{i=1}^n i * i - i - 2$ exceda de una cantidad introducida por el teclado máximo.

5.13. Calcular todos los números de exactamente tres cifras tales que la suma de los cuadrados de sus dígitos es igual al cociente de la división entera del número entre 3.

5.14. El valor de e^x se puede aproximar por la suma:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} = \sum_{i=0}^n \frac{x^i}{i!}$$

Escribir un programa que lea un valor de x como entrada y visualice las sumas parciales de la serie anterior, cuando se ha realizado una suma, dos sumas, tres sumas, ..., 15 sumas.

- 5.15.** 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 estrictos de 6 son 1, 2, 3 y $1 + 2 + 3 = 6$. Escribir un programa que lea un número entero positivo tope y escriba todos los números perfectos menores o iguales que él.

- 5.16.** Diseñar un programa que produzca la siguiente salida:

```

ZYXWVTSRQPONMLKJIHGFEDCBA
YXWVTSRQPONMLKJIHGFEDCBA
XWVTSRQPONMLKJIHGFEDCBA
WVTSRQPONMLKJIHGFEDCBA
VTSRQPONMLKJIHGFEDCBA
...
...
...
FEDCBA
EDCBA
DCBA
CBA
BA
A

```

- 5.17.** Calcular la suma de la serie $1/1^3 + 1/2^3 + \dots + 1/n^3$ donde n es un número positivo que se introduce por teclado.

- 5.18.** Calcular la suma de los 20 primeros términos de la serie: $1^2/3^2 + 2^2/3^2 + 3^2/3^2 + \dots + n^2/3^n$.

- 5.19.** Determinar si un número natural mayor o igual que la unidad que se lee del teclado es primo.

SOLUCIÓN DE LOS EJERCICIOS

- 5.1.** La variable `cuenta` toma los valores 1, 2, 3, 4, 5. Para los valores 1, 2, 3, 4, de `cuenta` se ejecuta la sentencia `cout << (2 * cuenta) << " "`; del interior del bucle `for`, por lo que se escribe el doble del valor de `cuenta` seguido de un espacio en blanco. Cuando la variable `cuenta` toma el valor de 5 se sale del bucle. La salida total del segmento de programa es: 2 4 6 8.

- 5.2.** a) La variable `n` se inicializa al valor 10. En cada iteración del bucle, la variable `n` se decremente en dos unidades. Es decir los valores que toma `n` son 10, 8, 6, 4, 2, 0. Las sentencias interiores del bucle `cout << "Hola ";` `cout << n << endl;` se ejecutan para los valores positivos de `n`. Por tanto la salida del bucle es:

```

Hola 10
Hola 8
Hola 6
Hola 4
Hola 2

```

- b) En este segundo caso la variable `n` se inicializa al valor 2 y se decremente en cada iteración 0.5 unidades saliendo del bucle cuando es negativa o nula. La sentencia `cout << n << " "`; sólo se ejecuta para los valores positivos de `n`. Por tanto la salida es: 2 1.5 1 0.5.

- 5.3.** La solución es la siguiente:

- a) En este primer caso cuando `n` toma el valor de 0, no se entra en el bucle `while`, ya que la `i` se ha inicializado a 1. Por lo tanto se escribe 1.

- b) Si n vale 1, se entra en el bucle una vez, ya que $1 \leq 1$. Como el resto de la división entera de 1 entre 1 es 0, se ejecuta la sentencia $++i$ por lo que el valor de i es 2. Al tomar i el valor de 2, el bucle termina y la salida del programa es 2.
- c) Al ser 3 el valor de n , se entra en el bucle pero nunca se ejecuta la sentencia $++i$, ya que el resto de la división entera de 1 entre 3 es siempre 1 que es distinto de 0. De esta forma el bucle nunca termina. Es un bucle infinito.

5.4. La solución es la siguiente:

- a) El bucle exterior se ejecuta cinco veces tomando la variable i los valores de 0, 1, 2, 3 y 4 al comienzo de cada iteración. El bucle interior comienza con el valor de j en 0, y termina con el valor de j en $i-1$. De esta forma cuando i toma el valor 0 no entra, cuando i toma el valor 1 se ejecuta una sola vez, cuando i toma el valor 2 se ejecuta 2 veces, etc. El bucle interior controlado por el contador j escribe un blanco seguido de asterisco (*) cada vez que se ejecuta. El bucle exterior controlado por el contador i ejecuta el bucle interior y da un salto de línea. De esta forma los dos bucles anidados escriben un triángulo rectángulo de asteriscos.
- b) El bucle exterior toma ahora los valores de 5, 4, 3, 2, 1, 0, por lo que se ejecuta seis veces. El bucle interior comienza con el valor de j en 3, y termina con el valor de j en 1, por lo que cada vez que se ejecuta escribe tres blancos seguidos de *. De esta forma se escribe un rectángulo de asteriscos que tiene seis filas y tres columnas de asteriscos (*).

Al ser ejecutado el siguiente programa se obtiene el resultado indicado en la salida .

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int n = 5, m = 3;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
            cout << " *";
        cout << endl;
    }
    cout << endl;
    for (int i = n; i > 0; i--)
    {
        for (int j = m; j > 0; j--)
            cout << " * ";
        cout << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución:



- 5.5.** Se trata de dos bucles anidados controlados por una condición. El bucle externo controlado por el contador *i*, y el interno controlado por la variable *j*. La variable *i*, toma los valores 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. Por consiguiente, se escriben los valores de $i + j = 2, 3, 5, 9$. 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. Por tanto, se escriben los valores de $i + j = 4, 6, 10$. Cuando *i* vale 3, la variable *j* toma los valores 3, 6, y cuando toma el valor 12 se sale del bucle. De esta forma se escriben los valores 6, 9. Si se introducen los bucles anteriores en un programa principal y se ejecuta se obtiene el siguiente resultado:

```
2 3 5 9
4 6 10
6 9
*****
```

SOLUCIÓN DE LOS PROBLEMAS

- 5.1.** Se declaran las variables enteras *i*, *n* y *suma*, inicializando *i* a 1 así como *suma* a 0. Mediante un bucle *do while* se lee el valor de *n* asegurando que sea positivo. Mediante un bucle *while* controlado por el contador *i*, se van acumulando en *suma*, previamente inicializado a 0, los valores $1 + 2 + 3 + \dots + (n-1) + n$. Al final del programa se escribe el resultado.

Codificación

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int n, i = 1; suma = 0;

    do
    {
        cout << " valor de n > 0 ";
        cin >> n;
    } while (n <= 0);

    while (i <= n)
    {
        suma += i;
        i++;
    }
    cout << " valor de la suma " << suma << endl ;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Un resultado de ejecución cuanto se introduce por teclado el valor de m es el siguiente:

```
valor de n > 0 4
valor de la suma 10
```

- 5.2. El triángulo solicitado tiene dos partes diferenciadas. La primera está constituida por las cuatro primeras filas y la segunda por las tres siguientes. Para escribir las cuatro primeras filas basta con anidar dos bucles. El bucle exterior varía desde las posiciones $i = 1, 2, 3, 4$, el interior desde las posiciones $j = 1, 2, \dots, i$. Para escribir la segunda parte, se anidan de nuevo los dos mismos bucles, pero ahora el exterior controlado por el contador i que varía entre las posiciones 3, 2, 1. El bucle interior varía desde las posiciones $j = 1, 2, \dots, i$. El triángulo pedido en el problema es fácilmente generalizable, para que en lugar de llegar al valor máximo 4 llegue a un valor máximo m . El programa que se codifica hace la generalización indicada. Se pide y valida un valor positivo m que en el problema debería ser 4, y posteriormente se escriben los bucles descritos anteriormente.

La codificación de este problema se encuentra en la página Web del libro.

Resultado de una posible ejecución para el valor de entrada $m = 6$

```
introduzca valor de m positivo 6
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

- 5.3. Se declaran contadores de números positivos np , número negativos nn , y números nulos que se inicializan en la propia declaración a 0. Un bucle controlado por el contador i itera entre los valores 1, 2, ..., max, siendo max una constante declarada en una sentencia define con el valor de 10. En cada iteración se lee un número entero en la variable dato, y dependiendo de que el valor leído sea positivo negativo o nulo se incrementa el contador de positivos, negativos o nulos. Al final del programa se escriben los contadores.

Codificación

```
#include <cstdlib>
#include <iostream>
#define max 10
using namespace std;

int main(int argc, char *argv[])
{
    int np = 0, nn = 0, nulos = 0, dato;

    for (int i = 1; i <= max; i++)
    {
        cin >> dato;
        if (dato > 0)
            np++;
        else if (dato < 0)
            nn++;
        else
            nulos++;
    }
}
```

```

        else
            nulos++;
    }
    cout <<"positivos    negativos    nulos " << endl;
    cout << np << "                " << nn << "                " << nulos;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 5.4.** Si n es positivo, el “triángulo” genérico solicitado, tiene un total de n filas, por lo que es necesario codificar un bucle controlado por un contador que escriba en cada iteración una fila de números. Como el número de columnas del triángulo va decreciendo, el bucle que itera las filas es conveniente se codifique decrementando la variable de control del bucle, para que mediante otro bucle interno que sea ascendente se puedan escribir cada una de las columnas. Es decir , el cuerpo del programa está gestionado por dos bucles anidados. El bucle externo está controlado por la variable i que toma los valores $n, n - 1, \dots, 1$. El bucle interno es iterado por el contador j que comienza en 1 y termina en i avanzando de uno en uno. Este bucle interno escribe el valor de j . Al terminar cada ejecución del bucle más interno hay que dar un salto de línea.

Codificación

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i, j, n;

    do
    {
        cout <<"valor de n positivo ";
        cin >>n;
    }
    while (n <= 0);                                // termina la lectura de n
    for (i = n; i >= 1; i--)                        // para cada una de las filas descendentemente
    {
        for (j = 1 ; j <= i; j++)                  // para cada una de las columnas
            cout << " " << j;
        cout << endl;                            // salta de línea
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Una ejecución del programa anterior es la siguiente:

```

valor de n positivo 7
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

```

- 5.5.** El problema planteado se resuelve mediante tres bucles. El primero es un bucle `do-while`, que valida la entrada del valor especificado límite_maximo positivo. El segundo bucle `do-while`, valida la entrada de la base entera positiva. Por último, mediante el bucle `for` controlado por el valor de potencia, se escriben las distintas potencias pedidas. Este bucle inicializa potencia a 1, en cada iteración escribe el valor de potencia y la multiplica por la base. La condición de terminación es potencia > límite_máximo.

La codificación de este problema se encuentra en la página Web del libro.

- 5.6.** Para sumar los $m = 30$ números pares, se usa el acumulador `suma` previamente inicializado a 0. En este acumulador se suman los respectivos números pares. Un bucle `for` controlado por el contador `i` recorre los números 1, 2, 3, ..., 20. El número par que ocupa la posición `i` es $2 * i$, por lo que para obtener la suma de todos basta con ejecutar la sentencia `suma += 2 * i`. El programa que se implementa presenta, además, el resultado de la suma mediante la fórmula de los m primeros números pares $\frac{(2 - 2 * m)}{2} m$ que se obtiene al aplicar la correspondiente suma de los términos de una progresión aritmética $S = \frac{a_1 + a_n}{2} n$.

Codificación

```
#include <cstdlib>
#include <iostream>
#define m 30
using namespace std;

int main(int argc, char *argv[])
{
    int i, suma = 0;

    for (i = 1; i <= m; i++)
        suma += 2 * i ;
    cout << "La suma de los 20 primeros pares: " << suma << endl;
    cout << " mediante formula: " << (2+ 2*m)*m/2 << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultados de ejecución del programa anterior:

**La suma de los 20 primeros pares: 930
mediante formula: 930**

- 5.7.** Teniendo en cuenta que las fórmulas que dan el área y volumen de una esfera son: $area = 4\pi radio^2$, $volumen = 4/3\pi radio^3$ para resolver el problema sólo se tiene que leer el radio (positivo), validarla en un bucle `do while` y aplicar las fórmulas. Se declara π como constante en una sentencia `define`.

Codificación

```
#include <cstdlib>
#include <iostream>
#define pi 3.2141592
using namespace std;
```

```

int main(int argc, char *argv[])
{
    float radio, area, volumen;

    do
    {
        cout << "valor del radio positivo " << endl;
        cin >> radio;
    } while (radio <= 0);                                // fin entrada de datos

    area = 4 * pi * radio * radio;
    volumen = 4.0 / 3 * pi * radio * radio * radio;
    cout << "el area y volumen de la esfera de radio r = " << radio << endl;
    cout << "area = " << area << " volumen = " << volumen ;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 5.8.** Se define la constante simbólica M como 5 y una “función en línea” $f(x)$ (también denominada “macro con argumentos”). El bucle se realiza 11 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 <cstdlib>
#include <iostream>
#include <math.h>
#define M 5
#define f(x) cos( 3 * x ) - 2 * x                         //función en línea
using namespace std;

int main(int argc, char *argv[])
{
    for (double x = 0.0; x <= M; x += 0.5)
        cout << x << " " << f(x) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 5.9.** Para resolver el problema se inicializa una variable *invertido* a cero. Se lee en un bucle *do while* el valor del dato *número* asegurando que es positivo. En un bucle *for* se invierte el dato numérico leído en *número*, destruyendo el dato y almacenando el resultado en el número *invertido*. En cada iteración del bucle se calcula en la propia variable *número*, el valor del cociente entero de *número* entre 10. De esta forma, si la variable *número* toma el valor de 234, en las sucesivas iteraciones irá tomando los valores 234, 23, 2 y 0. En cada iteración del bucle también se va calculando el resto del cociente entero de *número* entre 10. Es decir, se van calculando los valores 4, 3, 2 y almacenándolos sucesivamente en la variable *resto*. Para conseguir obtener el número invertido, basta con observar que $432 = 4 * 10 * 10 + 3 * 10 + 2 = (((0 * 10 + 4) * 10 + 3) * 10 + 2)$. (método de Horner de evaluación de polinomios). Es decir, basta con acumular en *invertido* el valor de *invertido* multiplicado por 10 y sumar el resto de la división entera.

La codificación de este problema se encuentra en la página Web del libro.

Un posible resultado de ejecución del programa anterior es:

```
introduzca valor del numero :4357
numero invertido : 7534
```

- 5.10.** El algoritmo transforma un par de enteros positivos (*mayor, menor*) en otro par (*menor, resto*), 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 otro entero de la pareja será el máximo común divisor de la pareja original.

Ejemplo mcd: (532, 112)

	4	1	3	→ Cocientes
532	112	84	28	
Restos	84	28	00	↓ mcd = 28

La codificación que se realiza, lee primeramente los números enteros *mayor* y *menor* validando la entrada en un bucle *do while*. Posteriormente, mediante otro bucle *while* se efectúan las correspondientes transformaciones para obtener el máximo común divisor. Se itera mientras el último *resto* de la división entera sea distinto de 0. En el cuerpo del bucle se realizan los cambios indicados anteriormente y además se escriben los resultados intermedios.

La codificación de este problema se encuentra en la página Web del libro.

Una ejecución del programa anterior para la entrada 1100 436 es la siguiente:

```
introduzca dos numeros positivos : 1100 436
calculando el maximo comun de 1100 y 436
resultados intermedios
436 228
228 208
208 20
20 8
8 4
4 0
el maximo comun divisor es : 4
```

- 5.11.** El valor de *n* se solicita y valida al principio del programa mediante un bucle *do while*. Los números son introducidos por el usuario. El primero de la serie inicializa las variables *Mayor*, *menor* y *suma*. En un bucle *for* se leen 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 acabo de leer) se recalculan los nuevos máximos, mínimos, y suma en las variables *Mayor*, *menor* y *suma*. Al final se escriben los resultados de *Mayor*, *menor* y media que es el cociente *suma / n*.

La codificación de este problema se encuentra en la página Web del libro.

Un resultado de ejecución del programa anterior es:

```
valor de n positivo: 3
introduzca 3 numeros
5
1
6
media = 4
menor = 1
mayor = 6
```

- 5.12.** En primer lugar se lee el valor de la cantidad introducida por teclado `máximo` validando la entrada. Posteriormente, se acumula la serie dada hasta que se excede el valor introducido mediante un bucle `for`. Este bucle inicializa `suma` y el contador `i` a 0. En cada iteración se incrementa el contador `i` en 1, se calcula el término $i * i - i - 2$ y se acumula en `suma`. El bucle `for` termina cuando `suma >= máximo`. El número total de términos `n` sumados viene dado por el valor de la variable `i`.

Codificación

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i, n;
    float suma, maximo;

    do
    {
        cout << "valor maximo positvo ";
        cin >> maximo;
    } while (maximo <= 0);

    for ( suma = 0, i = 0; suma <= maximo;)
    {
        i++;
        suma = suma + i * i - i - 2;
    }
    n = i;
    cout << " valor de la suma = " << suma << endl;
    cout << " numero de terminos = " << n << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Un resultado de ejecución del programa anterior es:

```
valor maximo positivo 30
valor de la suma = 58
numero de terminos = 6
```

- 5.13.** La solución se plantea mediante un bucle que recorre todos los números de tres cifras. Este bucle comienza en 100 y termina en 999. En cada iteración, se calcula, a su vez, cada una de los dígitos del número y se comprueba la condición en cuyo caso se escribe. Si el número $i = d_3d_2d_1$ entonces la condición indicada es $i / 3 = d_1 * d_1 + d_2 * d_2 + d_3 * d_3$. Para calcular las cifras basta con usar el cociente y la división entera dos veces tal y como aparece en la codificación.

Codificación

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

```

int main(int argc, char *argv[])
{
    int d1, d2, d3, i, x;

    cout << " lista de numeros que cumplen la condicion\n";
    for(i = 100; i <= 999; i++)
    {
        x = i ;
        d1 = x % 10;
        x = x / 10;
        d2 = x % 10;
        x = x / 10;
        d3 = x;
                                // ya se han calculado las tres cifras
        if( d1 * d1 + d2 * d2 + d3 * d3 == i / 3)
            cout << " numero " << i << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

El resultado de ejecución del programa anterior es el siguiente:

```

lista de numeros que cumplen la condicion
numero 116
numero 155
numero 267

```

- 5.14.** El problema se resuelve teniendo en cuenta que para calcular el valor de la serie , basta con ir acumulando los sucesivos valores del término en eaproximado, 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 o 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 , el término 2 es $x^2/2!$ y así sucesivamente. La codificación que se presenta lee el dato x , inicializa adecuadamente el valor de eaproximado a 1 y el termino a 1. Mediante un bucle controlado por el contador i se itera 15 veces, se calcula en cada iteración el nuevo término, se acumula su valor en eaproximado y escribe el resultado. Al final se escribe el valor de biblioteca de e^x .

La codificación de este problema se encuentra en la página Web del libro.

- 5.15.** Se lee el número tope en un bucle do while validando la entrada. Para resolver el problema basta con programar dos bucles for anidados. El bucle externo recorre todos los números menores o iguales que tope, y el interno decide si el número es perfecto. En la codificación que se realiza el bucle for externo está controlado por la variable entera n que recorre todos los números menores o iguales que tope. El bucle for, interno, se realiza con el contador i encargándose de probar todos los posibles candidatos a divisores menores que n (basta con empezar i 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). Todos los divisores del número n (aquellos cuyo resto de la división entera de n entre i sea 0) se van acumulando en acumulador, previamente inicializado a 0, para que una vez se termina el bucle comprobar la condición de perfecto ($n == acumulador$) y dar el mensaje correspondiente.

La codificación de este problema se encuentra en la página Web del libro.

- 5.16.** En la primera línea se escriben todas las letras mayúsculas del abecedario en orden inverso comenzando en Z. En la segunda línea se escriben las letras mayúsculas, en orden inverso, pero comenzando en la Y. Se observa que hay tantas líneas como letras mayúsculas existen en el abecedario, que todas las líneas terminan en el carácter A, y que una línea consecu-

tiva de otra comienza siempre por la letra mayúscula inmediatamente anterior. Para codificar el programa se necesitan dos bucles anidados. El bucle externo: comienza en 'Z'; termina en 'A'; disminuye en cada iteración en una unidad; ejecuta el bucle más interno; y realiza un salto de línea. El bucle interno itera comenzando en la letra del bucle externo y termina también en 'A'. En cada iteración escribe el carácter y decrementa el contador en una unidad.

Codificación

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    char car, Comienzo = 'Z';

    for (Comienzo = 'Z'; Comienzo >= 'A'; Comienzo --)
    {
        for (car = Comienzo; car >= 'A'; car--)
            cout << car << ' ';
        cout << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- 5.17. Para realizar la suma de la serie, basta con acumular en una variable `suma` los distintos valores de los términos $\text{termino}_i = 1 / (i * i * i)$. Previamente se lee de la entrada el valor del número de términos n validando la entrada en un bucle `do while` y posteriormente con un bucle `for` controlado por la variable `i` se va realizando la correspondiente acumulación en el acumulador `suma`, que se inicializa antes de comenzar el bucle a 0.

La codificación de este problema se encuentra en la página Web del libro.

- 5.18. Para realizar la suma de la serie, basta con acumular en un acumulador `suma`, previamente inicializado a 0 los términos ($\text{terminos}_i = i * i / 3^i$). El número de términos a sumar se declara como una constante `veinte` en una sentencia `define` y posteriormente se realiza la acumulación de los valores de `termino` en el acumulador `suma` mediante un bucle `for` que recorre los valores 1, 2, 3, ..., 20.

Codificación

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#define veinte 20
using namespace std;

int main(int argc, char *argv[])
{
    float termino, suma = 0;

    for (float i = 1; i <= veinte; i++)
    {
        termino = i * i / pow(3,i);
        suma += termino;
    }
}
```

```

cout << " valor de la suma = " << suma << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

```

- 5.19.** Un número mayor que la unidad es primo, si sus únicos divisores son el uno y el propio número. Teniendo en cuenta que si hay un número natural divisor que divide a la variable numero que es menor que la raíz cuadrada del numero, entonces hay otro número natural que también lo divide que es mayor que la raíz cuadrada del numero, se tiene que: basta con comprobar los posibles divisores menores o iguales que la raíz cuadrada del número dado. El programa se realiza con un solo bucle, en el cual se van comprobando los posibles divisores, siempre y cuando, no se haya encontrado ya algún divisor anterior (primo == false), o no se tenga que controlar ningún otro divisor (contador * contador <= numero). Primariamente, se lee en un bucle do while el valor del numero mayor o igual que 2. Se inicializa una variable primo a true, se itera con un bucle for controlado por contador e inicializado a 2, comprobando los posibles divisores que son menores o iguales que la raíz cuadrada del numero e incrementando en cada iteración el contador en una unidad. En el cuerpo del bucle for la variable primo se pone a false si contador divide a numero. En otro caso primo sigue estando a true.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

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

```

suma = 0;
while (suma < 100)
    suma += 5;
cout << suma << endl;

```

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

a) `for (int i = 0; i < 10; i++)
 cout << " 2* " << i << " = " << 2 * i
 << endl;`

b) `for (int i = 0; i <= 5; i++)
 cout << 2 * i + 1 << " ";`

c) `for (int i = 1; i < 4; i++)
{
 cout << i;
 for (int j = i; j >= 1; j--)
 cout << j << endl;
}`

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

a) `for (int i = 1; i <= 5; i++)
{
 cout << i << endl;
 for (int j = i; j >= 1; j-2)
 cout << j << endl;
}`

b) `for (int i = 3; i > 0; i--)
 for (int j = 1; j <= i; j++)
 for (int k = i; k >= j; k--)
 cout << i << j << k << endl;`

c) `for (int i = 1; i <= 3; i++)
 for (int j = 1; j <= 3; j++)
 {
 for (int k = i; k <= j; k++)
 cout << i << j << k << endl;
 cout << endl;
 }`

PROBLEMAS PROPUESTOS

- 5.1. 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.
- 5.2. 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.
- 5.3. 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.
- 5.4. Realizar un programa que escriba todos los números pares comprendidos entre 1 y 50.
- 5.5. Escribir un programa que calcule y visualice una tabla con las 20 potencias de 2.
- 5.6. Imprimir los cuadrados de los enteros de 1 a 20.
- 5.7. Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (1988), excepto los múltiplos de 100 que no son bisiestos salvo que a su vez también sean múltiplos de 400 (1800 no es bisiesto, 2000 sí).
- 5.8. 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 idénticos. 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.

- 5.9. 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 muertos, ¿cuántas parejas habrá después de un año? Mejorar el problema calculando el número de meses necesarios para producir un número dado de parejas de conejos.
- 5.10. Calcular la suma de la serie $1/1 + 1/2 + \dots + 1/N$ donde N es un número que se introduce por teclado.
- 5.11. Calcular la suma de los términos de la serie:

$$1/2 + 2/2^2 + 3/2^3 + \dots + n/2^n$$
- 5.12. Encontrar el número mayor de una serie de números.
- 5.13. Escribir un programa que calcule la suma de los 50 primeros números enteros.
- 5.14. Calcular todos los números de tres cifras tales que la suma de los cubos de las cifras sea igual al valor del número.

CAPÍTULO 6

Funciones y módulos

Introducción

Las funciones contienen varias sentencias bajo un solo nombre, que un programa puede utilizar una o más veces para ejecutar dichas sentencias. Ahorran espacio, reduciendo repeticiones y haciendo más fácil la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables.

Si se agrupan funciones en bibliotecas, otros programas pueden reutilizar las funciones; por esta razón se puede ahorrar tiempo de desarrollo. Y dado que las bibliotecas contienen rutinas presumiblemente comprobadas, se incrementa la fiabilidad del programa completo.

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 el lector, 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.

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.

6.1. Concepto de función

Para escribir un programa largo en C++, se divide éste en varios módulos o funciones. 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 realice el programa. Cada función realiza una determinada tarea y cuando se ejecuta la sentencia `return` o termina el código de la función y se retorna al punto en que fue llamada por el programa o función principal.

Consejo: Una buena regla para determinar la longitud de una función (número de líneas que contiene) es que no ocupe más longitud que el equivalente a una pantalla.

6.2. Estructura de una función

Una función es un conjunto de sentencias —con un nombre común— que se pueden llamar desde cualquier parte de un programa. En C++ todas las funciones son externas o globales, es decir, pueden ser llamadas desde cualquier punto del programa. Las funciones en C++ no se pueden anidar (no puede declararse una función dentro de otra función). La estructura de una función en C++ se muestra a continuación:

```
tipo_de_retorno nombre_Función (Lista_de_Parámetros_Formales)
{
    cuerpo de la función
    return expresión;
}
```

tipo_de_retorno

Tipo de valor devuelto por la función o la palabra reservada `void` si la función no devuelve ningún valor.

nombre_Función

Identificador o nombre de la función.

Lista_de_Parámetros_Formales

Lista de declaraciones de los parámetros de la función separados por comas.

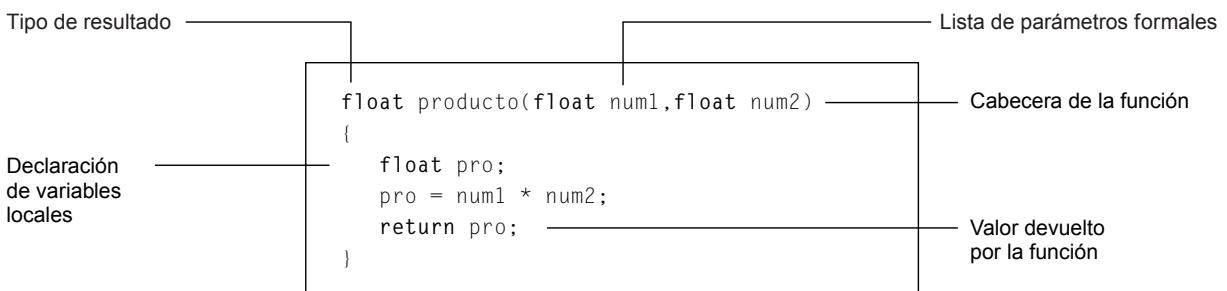
expresión

Valor que devuelve la función.

Los aspectos más sobresalientes en el diseño de una función son:

- *Tipo_de_retorno*. Es el tipo de dato que devuelve la función C++. El tipo debe ser uno de los tipos simples de C++, tales como `int`, `char` o `float`, o un puntero a cualquier tipo C++, o un tipo `struct`, previamente declarado (véase Capítulo 8). Si no devuelve ningún valor el tipo es `void`.
- *nombre_Función*. Un nombre de una función comienza con una letra o un subrayado (`_`) y puede contener tantas letras, números o subrayados como desee.
- *Lista_de_Parámetros_Formales*. 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*. Posteriormente se verá que el paso de parámetros en C++ se puede hacer por valor y por referencia.
- *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*. Una función puede devolver un único valor. Mediante la palabra reservada `return` se puede devolver el valor de la función. Una función puede tener cualquier número de sentencias `return`. Tan pronto como el programa encuentra cualquiera de las sentencias `return`, se retorna a la sentencia llamadora.
- *La llamada a una función*. Una llamada a una función redirigirá el control del programa a la función nombrada. Debe ser una sentencia o una expresión de otra función que realiza la llamada. 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.
- *No se pueden declarar funciones anidadas*. Todo código de la función debe ser listado secuencialmente, a lo largo de todo el programa. Antes de que aparezca el código de una función, debe aparecer la llave de cierre de la función anterior.

EJEMPLO 6.1. Codificar la función `producto()`, mostrar su estructura y realizar una llamada.



```
cout << producto(3,4);
```

La llamada anterior visualiza el valor 12 que es el resultado retornado por la función `producto`, ya que multiplica `num1` por `num2`, cuyos valores en la llamada son 3 y 4 respectivamente.

EJEMPLO 6.2. La función `min` devuelve el menor de los dos números enteros que se pasan como parámetro. La función `main` itera llamadas a `min`.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int min (int x, int y)
{
    if (x < y)
        return x;
    else
        return y;
}

int main(int argc, char *argv[])
{
    int m, n;

    do {
        cout << "introduzca dos numeros. Si primero es cero fin ";
        cin >> m >> n;
        if (m != 0)
            cout << " el menor es :" << min(m, n) << endl;           //LLamada a min
    }while(m != 0);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Un resultado de ejecución es el siguiente:

```
introduzca dos numeros. Si primero es cero fin 2 4
el menor es :2
introduzca dos numeros. Si primero es cero fin 5 3
el menor es :3
introduzca dos numeros. Si primero es cero fin 0 4
Presione una tecla para continuar . . . -
```

EJEMPLO 6.3. La función `norma` devuelve la norma euclídea de las tres coordenadas de un vector de R^3 .

$$\text{Norma}(x, y, z) = \sqrt{x^2 + y^2 + z^2}$$

```
#include <cstdlib>
#include <iostream>
#include <math.h>           //contiene función Sqrt

using namespace std;

float Norma (float x, float y, float z)
{
    return sqrt(x * x + y * y + z * z);
}
```

```

int main(int argc, char *argv[])
{
    float x, y, z;

    cout << " vector : (" << 3 << "," << 4 << "," << 5 << ")" ;
    cout << " norma = " << Norma(3, 4, 5) << endl; //Llamada a norma
    cout << " introduzca las tres coordenadas de vector " ;
    cin >> x >> y >> z;
    cout << " vector : (" << x << "," << y << "," << z << ")" ;
    cout << " norma = " << Norma(x, y, z) << endl; //Llamada a norma
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Un resultado de ejecución es el siguiente:

```

vector : <3,4,5> norma = 7.07107
introduzca las tres coordenadas de vector 0 3 4
vector : <0,3,4> norma = 5
Presione una tecla para continuar . . .

```

EJEMPLO 6.4. *Función que calcula y devuelve la suma de los divisores de un número entero positivo que recibe como parámetro.*

La función `divisores` calcula la suma de todos los divisores del número incluyendo 1 y el propio número. Para realizarlo basta con inicializar un acumulador `acu` a 0, y mediante un bucle `for` recorrer todos los números naturales desde 1 hasta el propio `n`, y cada vez que un número sea divisor de `n` sumarlo al acumulador `acu` correspondiente. Una vez terminado el bucle la función toma el valor de `acu`.

```

int divisores(int n)
{
    int i, acu;

    acu = 0;
    for(i = 1; i <= n; i++)
        if (n % i == 0)
            acu += i;
    return acu;
}

```

6.3. Prototipos de las funciones

C++ requiere que una función se declare o defina antes de su uso. La *declaración* de una función se denomina *prototipo*. Específicamente un prototipo consta de los siguientes elementos: nombre de la función; lista de parámetros formales encerrados entre paréntesis y un punto y coma. Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la función `main()`. La *definición* completa de la función debe existir en algún lugar del programa; por ejemplo, antes o después de `main`.

El compilador utiliza los prototipos para validar que el número y los tipos de datos de los parámetros actuales de la llamada a la función son los mismos que el número y tipo de parámetros formales. 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 6.5. *Prototipo, llamada y definición de función. La función media recibe como parámetros dos números y retorna su media aritmética.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

double media (double x1, double x2);      //Declaración de media. Prototipo

int main(int argc, char *argv[])
{
    double med, numero1, numero2;

    cout << "introduzca dos numeros ";
    cin >> numero1 >> numero2;
    med = media (numero1, numero2);           //Llamada a la función
    cout << " lamedia es :" << med << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

double media(double x1, double x2)           //Definición de media
{
    return (x1 + x2)/2;
}
```

Resultado de una ejecución:

```
introduzca dos numeros 3 4
lamedia es :3.5
Presione una tecla para continuar . . .
```

EJEMPLO 6.6. *Prototipo sin nombres de parámetros en la declaración y sin parámetros formales. Calcula el área de un rectángulo. El programa se descompone en dos funciones, además de main(). La función entrada retorna un número real que lee del teclado. La función area calcula el área del rectángulo cuyos lados recibe como parámetros.*

```
#include <cstdlib>
#include <iostream>

using namespace std;

float area_r(float, float);      //Prototipo. Nombres parámetros omitidos
float entrada();                 //Prototipo sin parámetros

int main(int argc, char *argv[])
{
    float base, altura;

    cout << " Base del rectangulo. ";
    base = entrada();
    cout << " Altura del rectangulo. ";
    altura = entrada();
    cout << " Area rectangulo: " << area_r(base,altura) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

```

float entrada() //Retorna un numero positivo
{
    float m;

    do
    {
        cout << " Numero positivo: ";
        cin >> m;
    } while (m <= 0.0);
    return m;
}

float area_r(float b, float a) //Se declaran los nombres de parámetros
{
    return (b * a);
}

```

Resultado de ejecución:

```

Base del rectangulo. Numero positivo: 3
Altura del rectangulo. Numero positivo: 4
Área rectangulo: 12
Presione una tecla para continuar . .

```

6.4. Parámetros de una función

C++ proporciona dos métodos para pasar variables (*parámetros*) entre funciones. Una función puede utilizar *parámetros por valor* y *parámetros por referencia*, o puede no tener parámetros.

Paso por valor (también llamado *paso por copia*) significa que cuando C++ compila la función y el código que llama a la misma, la función recibe una copia de los valores de los parámetros actuales. La función receptora no puede modificar la variable de la función (parámetro pasado). En la Figura 6.1 se muestra el paso de *x* por valor. La ejecución del programa, produce la siguiente salida: 6 6 7 6.

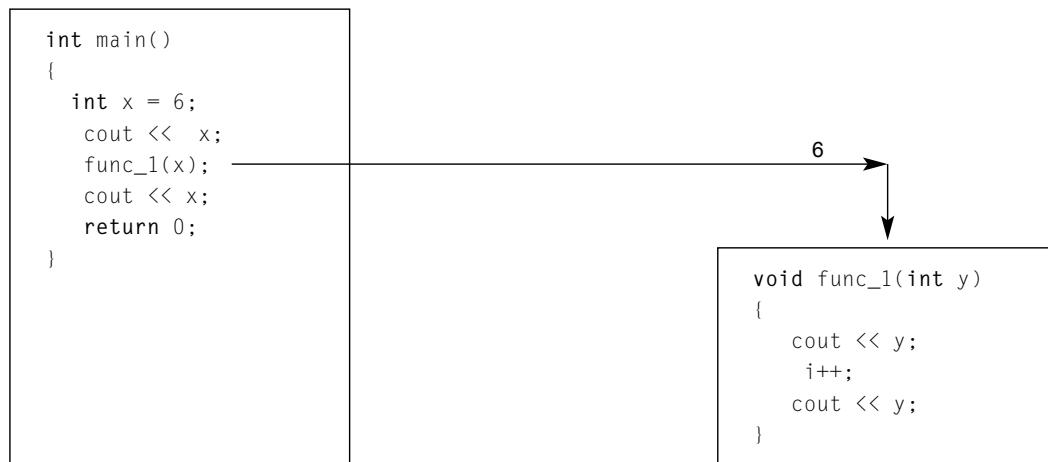


Figura 6.1. Paso de la variable *x* por valor

EJEMPLO 6.7. *Paso de parámetros por valor a una función.*

```

#include <cstdlib>
#include <iostream>
using namespace std;

```

```

void paso_por_valor(int x); //Prototipo

int main(int argc, char *argv[])
{
    int x = 20;

    cout << " antes de la llamada a paso_por_valor " << x << endl;
    paso_por_valor(x);
    cout << " despues de la llamada a paso_por_valor " << x << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

void paso_por_valor(int x)
{
    cout << " dentro de paso_por_valor " << x << endl;
    x *= 2;
    cout << " despues de x *=2 y dentro de paso_por_valor " << x << endl;
}

```

Resultado de ejecución:

```

antes de la llamada a paso_por_valor 20
dentro de paso_por_valor 20
despues de x *=2 y dentro de paso_por_valor 40
despues de la llamada a paso_por_valor 20
Presione una tecla para continuar . . .

```

Paso por referencia. Cuando una función debe modificar el valor del parámetro pasado y de volver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por *referencia* o *dirección*.

Para declarar una variable que es parámetro formal, como paso por referencia, el símbolo & debe preceder al nombre de la variable en la cabecera de la función, y en la llamada el parámetro actual correspondiente debe ser el nombre de una variable. También puede usarse el método de los punteros de C: en la declaración de la variable que es parámetro formal, el símbolo * debe preceder al nombre de la variable; en la llamada a la función debe realizarse el parámetro actual que debe ser & variable.

Cuando se modifica el valor del parámetro formal de un parámetro por referencia (la variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado.

EJEMPLO 6.8. *Paso de parámetros por referencia a una función, estilo C++.*

```

#include <cstdlib>
#include <iostream>
using namespace std;

void referencia(int& x) //Parámetro por referencia
{
    x += 2;
}

int main(int argc, char *argv[])
{
    int x = 20;

    cout << " antes de la llamada " << " x= " << x << endl;
    referencia (x); //Llamada con nombre de variable
}

```

```
cout << " despues de la llamada " << x << endl;
system("PAUSE");
return EXIT_SUCCESS;
}
```

Resultado de ejecución:

antes de la llamada x= 20
despues de la llamada x= 22
Presione una tecla para continuar . . .

EJEMPLO 6.9. *Paso de parámetros por referencia a una función, estilo C.*

```

#include <cstdlib>
#include <iostream>
using namespace std;

void intercambio(int* x, int* y) //Declaración como puntero
{
    int aux = *x;

    *x = *y;
    *y = aux;
}

int main(int argc, char *argv[])
{
    int x = 20, y = 30 ;

    cout << " antes de la llamada " ;
    cout << " x= " << x << " y= " << y << endl;
    intercambio (&x, &y); //Llamada con dirección
    cout << " despues de la llamada " ;
    cout << " x= " << x << " y= " << y << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Resultado de ejecución:

antes de la llamada x= 20 y= 30
despues de la llamada x= 30 y= 20
Presione una tecla para continuar . . .

Parámetros const en una función. El especificador `const`, indica al compilador que sólo es de lectura en el interior de la función. Si se intenta modificar en este parámetro se producirá un mensaje de error de compilación.

EJEMPLO 6.10. *Uso de const en la declaración.*

Argumentos por omisión o defecto. Una característica poderosa de las funciones C++ es que en ellas pueden establecer valores por *omisión* o *ausencia* (“por defecto”) para los parámetros. Se pueden asignar argumentos por defecto a los parámetros de una función. Cuando se omite el argumento de un parámetro que es un argumento por defecto, se utiliza automáticamente éste. La única restricción es que se deben incluir todas las variables desde la izquierda hasta el primer parámetro omitido. Si se pasan valores a los argumentos omitidos se utiliza ese valor; si no se pasa un valor a un parámetro opcional, se utiliza el valor por defecto como argumento. El valor por defecto debe ser una expresión constante.

EJEMPLO 6.11. La función asteriscos tiene tres parámetros. El primero indica el número de filas, el segundo indica el número de columnas y el tercero el carácter a escribir. El segundo y el tercer parámetros son por omisión.

```
#include <cstdlib>
#include <iostream>
using namespace std;

void asteriscos(int fila, int col = 3, char c = '*')
{
    for(int i = 0; i < fila; i++)
    {
        for (int j = 0; j < col; j++)
            cout << c;
        cout << endl;
    }
}

int main(int argc, char *argv[])
{
    asteriscos(4);           //Correcto dos parámetros por omisión
    cout << endl;
    asteriscos( 4,6);        //Correcto un parámetro por omisión
    cout << endl;
    asteriscos(4,6,'@');     //asteriscos()llamada incorrecta. Primer parámetro
                           //obligatorio
                           //asteriscos(4, , '@');   llamada incorrecta
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución:



Reglas de construcción de argumentos por defecto

- Los argumentos por defecto se deben pasar por valor. Nunca por referencia.
- Los valores de los argumentos por defecto pueden ser valores literales o definiciones const. No pueden ser variables.
- Todos los argumentos por defecto deben colocarse al final en el prototipo de la función. Después del primer argumento por defecto, todos los argumentos posteriores deben incluir también valores por defecto.

6.5. Funciones en línea (*inline*)

Existen dos opciones disponibles para la generación del código de las funciones en C++: *funciones en línea* y *fueras de línea*.

Las **funciones en línea** (*inline*) sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función se utiliza muchas veces en el programa y su código es pequeño. Para una *función en línea* (*inline*), el compilador inserta realmente el código para la función en el punto en que se llama a la función. Esta acción hace que el programa se ejecute más rápidamente.

Una función normal, **fueras de línea**, es un bloque de código que se llama desde otra función. El compilador genera código para situar la dirección de retorno en la pila. La dirección de retorno es la dirección de la sentencia que sigue a la instrucción que llama a la función. A continuación, el compilador genera códigos que sitúan cualquier argumento de la función en la pila a medida que se requiera. Por último, el compilador genera una instrucción de llamada que transfiere el control a la función.

Tabla 6.1. Ventajas y desventajas de la función en línea

	Ventajas	Desventajas
Funciones en línea	Rápida de ejecutar.	Tamaño de código grande.
Funciones fuera de línea	Pequeño tamaño de código.	Lenta de ejecución.

Para crear una función en línea (*inline*), insertar la palabra reservada *inline* delante de una declaración normal y del cuerpo, y situarla en el archivo fuente antes de que sea llamada. La sintaxis de declaración es:

```
inline TipoRetorno NombreFunción (Lista parámetros con tipos)
{ cuerpo}
```

EJEMPLO 6.12. *Funciones en línea para calcular el volumen y el área total de un cilindro del que se leen su radio y altura.*

El volumen de un cilindro viene dado por: $volumen = \pi * radio^2 * altura$. El Areatotal viene dada por $Areatotal = 2 * \pi * radio * altura + \pi * radio^2$. Para resolver el problema basta con declarar las variables correspondientes, declarar la constante pi y las dos funciones en línea que codifican las fórmulas respectivas. El programa principal lee el radio y la altura en un bucle while garantizando su valor positivo.

```
#include <cstdlib>
#include <iostream>
using namespace std;

const float Pi = 3.141592;

inline float VOLCILINDRO(float radio, float altura) // Función en línea
{
    return (Pi * radio * radio * altura);
}
```

```

inline float AREATOTAL(float radio,float altura)      // Función en línea
{
    return (2 * Pi * radio * altura + Pi * radio * radio);
}

int main(int argc, char *argv[])
{
    float radio, altura, Volumen, Areatotal;
    do
    {
        cout << "Introduzca radio del cilindro positivo: ";
        cin >> radio;
        cout << "Introduzca altura del cilindro positiva: ";
        cin >> altura;
    }while (( radio <= 0 ) || (altura <= 0));
    Volumen = VOLCILINDRO(radio, altura); //llamada sustituye el código
                                         // la sentencia anterior es equivalente a
                                         // Volumen = Pi*radio*radio*altura;
    Areatotal = AREATOTAL(radio, altura); //llamada sustituye el código
                                         // la sentencia anterior es equivalente a
                                         // Areatotal = 2*Pi*radio*altura+Pi*radio*radio;
    cout << "El volumen del cilindro es:" << Volumen << endl;
    cout << "El Área total del cilindro es:" << Areatotal << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Resultado de ejecución:

```

Introduzca radio del cilindro positivo: 4
Introduzca altura del cilindro postivia: 5
El volumen del cilindro es:251.327
El Área total del cilindro es:175.929
Presione una tecla para continuar . . .

```

También son funciones en línea las definidas con la sentencia `define`. 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 las macros con argumentos *no hay comprobación de tipos*.

EJEMPLO 6.13. Función en línea para definir una función matemática.

```

#include <cstdlib>
#include <iostream>
using namespace std;

#define fesp(x) (x * x + 2 * x -1)

int main(int argc, char *argv[])
{
    float x;

```

```
for (x = 0.0; x <= 6.5; x += 0.3)
    cout << x << " " << fesp(x) << endl;
system("PAUSE");
return EXIT_SUCCESS;
}
```

6.6. Ámbito (alcance)

El ámbito es la zona de un programa en el que 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 `static` es aquella que tiene una *duración fija*. El espacio para el objeto se establece en tiempo de compilación; existe en tiempo de ejecución y se elimina sólo cuando el programa desaparece de memoria en tiempo de ejecución.
 - 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 6.14. Ámbito de programa y de función. Calcula el área de un círculo y la longitud de una circunferencia de radio leído de la entrada.

El área de un círculo se calcula mediante la fórmula $\text{Area} = \pi * \text{radio}^2$; la longitud de una circunferencia por $\text{Longitud} = 2 * \pi * \text{radio}$. El parámetro formal `r` de las funciones `longitud` y `Area` tienen ámbito de bloque. La variable `Pi` (no modificable) tiene ámbito de programa. La variable `radio` tiene ámbito de función. El programa principal lee el `radio` positivo, usando un bucle `do while`, y mediante llamadas a las funciones `longitud` y `Area` visualiza los resultados.

```
#include <cstdlib>
#include <iostream>
using namespace std;

const float Pi = 3.141592; // Ámbito de programa

float longitud(float r)
{
    return Pi * r; // El parámetro r tiene ámbito de función
}

float Area(float r)
{
    return (Pi * r * r); // El parámetro r tiene ámbito de función
}

int main(int argc, char *argv[])
{
    float radio; //Tiene ámbito de función
```

```

do
{
    cout << "Introduzca radio positivo: ";
    cin >> radio;
}while ( radio <= 0);

cout << " La longitud de la circunferencia es: " ;
cout << longitud(radio) << endl;
cout << " El area del circulo es: " << Area(radio) << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

```

Un resultado de ejecución del programa es:

```

Introduzca radio positivo: 2
La longitud de la circunferencia es: 6.28318
El area del circulo es: 12.5664
Presione una tecla para continuar . . . -

```

EJEMPLO 6.15. Ámbito de función y de bloque. La función factorial se define de la siguiente forma: $\text{factorial}(n) = 1$ si $n=0$, y $\text{factorial}(n) = n * \text{factorial}(n-1)$ si $n>0$.

La función factorial, se programa no recursivamente, usando un bucle ascendente, inicializando el acumulador a 1 (Ámbito de función) a1 y multiplicando en cada iteración el acumulador f por la variable de control del bucle i (Ámbito de bloque for). El programa principal lee los valores positivos valor1 y valor2 y mediante un bucle for controlado por la variable i llama sucesivamente a la función factorial visualizando los resultados. Las variables valor1, valor2 e i tienen ámbito de función.

```

#include <cstdlib>
#include <iostream>
using namespace std;

long int factorial (int n); //Prototipo de función

int main(int argc, char *argv[])
{
    int valor1, valor2, i; // Ámbito función main

    do
    {
        cout << "Introduzca dos numeros positivos x < y: ";
        cin >> valor1 >> valor2;
    }while (( valor1 <= 0) || (valor2 <= 0) || (valor1 > valor2));

    for (i = valor1; i <= valor2; i++)
        cout << i << " factorial " << factorial(i) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

long int factorial (int n)
{
    long int f; //Ámbito función

```

```

f = 1.0 ;
for (int i = 1; i <= n; i++)
    f = f * i;
return f;
}

```

Una ejecución del programa es:

```

Introduzca dos numeros positivos x < y: 3 6
3 factorial 6
4 factorial 24
5 factorial 120
6 factorial 720
Presione una tecla para continuar . . .

```

6.7. 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 tipos: `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 6.16. Declaración de variables automáticas.

<code>auto int xl;</code>	<i>es igual que</i>	<code>int xl;</code>
<code>auto float a,b;</code>	<i>es igual que</i>	<code>float a,b;</code>
<code>auto char ch, chal</code>	<i>es igual que</i>	<code>char ch, chl;</code>

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 externas. Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro archivo fuente y que puede ser usada en el archivo actual. Una variable global definida en un archivo, puede ser usada en la compilación de otro archivo distinto pero sin reservar nuevo espacio en memoria, para que al ser montadas juntas, ambas compilaciones funcionen correctamente. Si no se hiciera la declaración de variable externa, entonces ocuparían posiciones de memoria distintas y al montar los dos archivos no funcionaría. Se declaran precediendo a la declaración de variable, la palabra `extern`.

EJEMPLO 6.17. Las funciones `LeerReal` y `EscribirReal` leen y escriben respectivamente la variable real `r`. Esta variable `r` es global y no está en el archivo fuente de las funciones.

```

// archivo fuente  axtern1.cpp
#include <iostream>
using namespace std;

void LeerReal(void)
{
    extern float f;      // variable definida en otro archivo (extern2.cpp)
    cout << " introduzca dato ";
    cin >> f;
}

```

```

void EscribirReal(void)
{
    extern float f;           // variable definida en otro archivo (extern2.cpp)
    cout << f;
}

// archivo fuente  extern2.cpp
#include <cstdlib>
#include <iostream>
using namespace std;

float f;

void LeerReal(void);
void EscribirReal();

int main(int argc, char *argv[])
{
    LeerReal();
    EscribirReal();
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

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. Se declaran precediendo a la declaración de la variable con la palabra reservada `static`.

EJEMPLO 6.18. Calcula las sucesivas potencias $a^0, a^1, a^2, \dots, a^n$, usando una variable estática.

La función `potenciaS` tiene como variable `f` como variable estática, por lo que recuerda su valor en cada llamada. En la primera llamada el valor de `f` es 1; al multiplicar `f` por `a`, `f` toma el valor de a^1 , y se almacena en memoria. En la siguiente llamada, al multiplicar `f` por `a`, `f` toma el valor de a^2 , y así sucesivamente. De esta forma, al ser llamado con el bucle `for` del programa principal, se van visualizando las sucesivas potencias de `a`.

También se programa la función `potencia`, mediante un bucle que multiplica por sí mismo el primer parámetro `a` tantas veces como indique el segundo. El programa principal lee la base `a` y el exponente positivo `n` y realiza las llamadas correspondientes. Cambiando la llamada a la función `potenciaS` por la función `potencia` se obtienen los mismos valores de salida.

```

#include <cstdlib>
#include <iostream>
using namespace std;

float potenciaS (float a, int n)
{
    static float f = 1.0 ;

    f *= a;
    return f;
}

float potencia (float a, int n)
{
    float f = 1.0;

```

```

    for ( int i = 1; i <= n; i++)
        f *= a;
    return f;
}

int main(int argc, char *argv[])
{
    float a;
    int n;

    cout << " valor de a ";
    cin >> a;
    do
    {
        cout << " valor de n ";
        cin >> n;
    } while (n<=0);

    for( int i = 1 ; i <= n ; i++)
        cout << a << " elevado a " << i << " = " << potenciaS(a,i) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Resultado de ejecución:

```

valor de a 2
valor de n 5
2 elevado a 1 = 2
2 elevado a 2 = 4
2 elevado a 3 = 8
2 elevado a 4 = 16
2 elevado a 5 = 32
Presione una tecla para continuar . .

```

EJEMPLO 6.19. Calcula la función con variables estáticas. La función se define de la siguiente forma: $f(n) = f(n-1) + 2f(n-2) + f(n-3)$ si $n > 3$, $f(0) = 0$, $f(1) = 1$, $f(2) = 2$.

Para programarla se definen tres variables locales estáticas que almacena los últimos valores obtenidos de la función. De esta forma, si se le llama desde un bucle `for` 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, ...

```

#include <cstdlib>
#include <iostream>
int resto(int n, int m);
using namespace std;

long int funcionx();

int main(int argc, char *argv[])
{
    int n , i;

    cout << " valor n de la funcionx ?: ";
    cin >> n;

```

```

cout << " Secuencia de funcionx: 0,1,2";
for (i = 3; i <= n; i++)
    cout << "," << funcionx();
cout << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

long int funcionx()
{
    static int x = 0, y = 1, z = 2;
    int aux;

    aux = x + 2 * y + z ;
    x = y;
    y = z;
    z = aux;
    return z;
}

```

Resultados de ejecución:

```

valor n de la funcionx ?: 8
Secuencia de funcionx: 0,1,2,4,9,19,41,88,189
Presione una tecla para continuar . .

```

6.8. Concepto y uso de funciones de biblioteca

Todas las versiones de C++ ofrecen una biblioteca estándar de funciones que proporcionan 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:

<cassert.h>	<cctype.h>	<errno.h>	<float.h>
<climits.h>	<locale.h>	<math.h>	<setjmp.h>
<signal.h>	<stdarg.h>	<stddef.h>	<stdio.h>
<stdlib.h>	<string.h>	<time.h>	

En los módulos de programa se pueden incluir líneas `#include` con los archivos de cabecera correspondientes en cualquier orden.

6.9. Miscelánea de funciones

Funciones de carácter. El archivo de cabecera `<cctype.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). (Véase la Tabla 6.2.)

EJEMPLO 6.20. Realizar un bucle que itera hasta que se introduzca s o n.

```

#include <cstdlib>
#include <iostream> // contiene <cctype.h>
using namespace std;

```

```

int main(int argc, char *argv[])
{
    char resp;                                //respuesta del usuario

    do
    { cout << " introduzca S = Si N = NO? ";
        cin >> resp;
        resp = toupper(resp);
    } while ( (resp != 'S') && (resp != 'N'));

    cout <<" respuesta leida " << resp;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Tabla 6.2. Funciones de caracteres

Función	Prueba (test) de
int isalpha(int c)	Verdadero si es letra mayúscula o minúscula.
int isdigit(int c)	Verdadero si es dígito (1, 2, ..., 9).
int isupper(int c)	Verdadero si es letra mayúscula (A-Z).
int islower(int c)	Verdadero si es letra minúscula (a-z).
int isalnum(int c)	isalpha(c) isdigit(c).
int iscntrl(int c)	Verdadero si es carácter de control. (Códigos ASCII 0-31).
int isxdigit(int c)	Verdadero si es dígito hexadecimal.
int isprint(int c)	Verdadero si Carácter imprimible incluyendo ESPACIO. código ASCII 21 a 127.
int isgraph(int c)	Verdadero si es carácter imprimible e excepto ESPACIO.
int isspace(int c)	Verdadero si c es carácter un espacio, nueva línea (\n), retorno de carro (\r), tabulación (\t) o tabulación vertical (\v).
int ispunct(int c)	Verdadero si es carácter imprimible no espacio, dígito o letra.
int toupper(int c)	Convierte a letra mayúscula.
int tolower(int c)	Convierte a letras minúscula.

Funciones numéricas. Virtualmente, cualquier operación aritmética es posible en un programa C++. Las funciones matemáticas disponibles son las siguientes: trigonométricas; logarítmicas; exponentiales; 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`.

EJEMPLO 6.21. Generar 10 números aleatorios menores que 100 y visualiza el menor y el mayor.

```

#include <cstdlib>
#include <iostream>
#include <time.h>

#define randomize ( srand ( time(NULL) ) )      //Macro para definir randomize
#define random(num) ( rand()%num)                // Macro para definir random
#define Tope 100
#define MAX( x, y)( x > y ? x : y )           // Macro para Maximo
#define MIN( x, y)( x < y ? x : y )           // Macro para Mínimo
using namespace std;

int main(int argc, char *argv[])
{
    int max, min, i;
    randomize;
    max = min = random(Tope);

```

```

for (i = 1; i < 10; i++)
{
    int x = random(Tope);

    min = MIN(min,x);
    max = MAX(max,x);
}
cout << "minimo " << min << " maximo " << max << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

```

Funciones de fecha y hora. Los microprocesadores tiene 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 (Unidad Central de Proceso) 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)` que convierte los dígitos de la cadena `cad` a número real; `atoi(cad)`, `atol(cad)` que 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 dónde 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. Si desea, por razones de legibilidad, puede utilizar la palabra reservada `extern` con el prototipo de función. Se puede hacer una función visible al exterior de un archivo fuente utilizando la palabra reservada `static` con la cabecera de la función y la sentencia del prototipo de función. Si se escribe la palabra `static` antes del tipo de valor devuelto por la función, la función no será pública al enlazador, de modo que otros módulos no tendrán acceso a ella.

6.10. Sobrecarga de funciones (polimorfismo)

La **sobrecarga** de funciones permite escribir y utilizar múltiples funciones con el mismo nombre, pero con diferente lista de argumentos. La lista de argumentos es diferente si tiene un argumento con un tipo de dato distinto, si tiene un número diferente de argumentos, o ambos. La lista de argumentos se suele denominar *signatura de la función*.

Las reglas que sigue C++ para seleccionar una función sobrecargada son:

- Si existe una correspondencia exacta entre los tipos de parámetros de la función llamadora y una función sobrecargada, se utiliza dicha función.
- Si no existe una correspondencia exacta, pero sí se produce la conversión de un tipo a un tipo superior (tal como un parámetro `int` a `long`, o un `float` a un `double`) y se produce, entonces, una correspondencia, se utilizará la función seleccionada.
- Se puede producir una correspondencia de tipos, realizando conversiones forzadas de tipos (*moldes-cast*).
- Si una función sobrecargada se define con un número variable de parámetros (mediante el uso de puntos suspensivos (...)) se puede utilizar como una coincidencia potencial.

EJEMPLO 6.22. Sobre carga de funciones.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int Sobrecarga(int);
int Sobrecarga(int, int);
float Sobrecarga(float, float);
float Sobrecarga (float, float, float);

int main(int argc, char *argv[])
{
    int x = 4, y = 5;
    float a = 6.0 , b = 7.0, c = 9.0;

    cout << "\n El cuadrado de " << x << " es: "
        << Sobrecarga(x);
    cout << "\n El producto de " << x << "por " << y << " es: "
        << Sobrecarga(x, y);
    cout << "\n La suma de " << a << "y " << b << " es: "
        << Sobrecarga(a, b);
    cout << "\n La suma de " << a << "y " << b << " es: "
        << Sobrecarga(a, b) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

// Sobre carga, calcula el cuadrado de un valor entero
int Sobrecarga(int valor)
{
    return (valor * valor);
}

// Sobre carga, multiplica dos valores enteros
int Sobrecarga(int valor1, int valor2)
{
    return(valor1 * valor2);
}

// Sobre carga, calcula la suma de dos valores reales
float Sobrecarga(float valor1, float valor2)
{
    return (valor1 + valor2);
}

// Sobre carga, calcula la media de tres valores reales
float Sobrecarga (float valor1, float valor2 , float valor3)
{
    return (valor1 + valor2 + valor3)/3;
}
```

6.11. Plantillas de funciones

Las **plantillas de funciones** (*function templates*) proporcionan un mecanismo para crear una *función genérica*. Una función genérica es una función que puede soportar simultáneamente diferentes tipos de datos para su parámetro o parámetros.

Una plantilla de función de tipo no genérico pero de argumentos genéricos tiene el siguiente formato:

```
template <class Tipo>
Tipo_de_funcion Func1(Tipo arg1, Tipo arg2)
{
    // cuerpo de la función Func1()
}
```

EJEMPLO 6.23. Función que retorna el menor con tipos de datos genéricos.

```
#include <cstdlib>
#include <iostream>
using namespace std;

template <class T>
T man(T a, T b)
{
    if (a < b)
        return a;
    else
        return b;
}

int main(int argc, char *argv[])
{
    int x = 4, y = 5;
    float a = 6.0 , b = 7.0;
    char c = 'C', d = 'A';

    cout << "\n El menor de " << x << " y " << y << " es: " << man(x, y);
    cout << "\n El menor de " << a << " y " << b << " es: " << man(a, b);
    cout << "\n El menor de " << c << " y " << d << " es: " << man(c, d) << endl ;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJERCICIOS

- 6.1. Escribir una función que tenga un argumento de tipo entero y que devuelva la letra P si el número es positivo, y la letra N si es cero o negativo.
- 6.2. Escribir una función lógica de dos argumentos enteros, que devuelva true si uno divide al otro y false en caso contrario.
- 6.3. Escribir una función que convierta una temperatura dada en grados Celsius a grados Fahrenheit. La fórmula de conversión es:

$$F = \frac{9}{5} C + 32$$

- 6.4. Escribir una función lógica Vocal que determine si un carácter es una vocal.

PROBLEMAS

- 6.1. Escribir una función que tenga como parámetros dos números enteros positivos num1 y num2, y calcule el resto de la división entera del mayor de ellos entre el menor mediante sumas y restas.
 - 6.2. Escribir una función que tenga como parámetros dos números enteros positivos num1 y num2 y calcule el cociente de la división entera del primero entre el segundo mediante sumas y restas.
 - 6.3. Escribir un programa que calcule los valores de la función definida de la siguiente forma:
- funciony(0) = 0,
 funciony(1) = 1
 funciony(2) = 2
 funciony(n) = funciony(n - 3) + 3 * funciony(n - 2) - funciony(n - 1) si n > 2.
- 6.4. Un número entero n se dice que es perfecto si la suma de sus divisores es incluyendo 1 y excluyéndose él coincide consigo mismo. Codificar una función que decida si un número es perfecto. Por ejemplo 6 es un número perfecto $1 + 2 + 3 = 6$.
 - 6.5. Escribir una función que decida si dos números 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.
 - 6.6. Dado el valor de un ángulo, escribir una función que muestre el valor de todas las funciones trigonométricas correspondientes al mismo.
 - 6.7. Escribir una función que decida si un número entero positivo es primo.
 - 6.8. Escribir 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 inicial v a intervalos de 0.1 s.

- 6.9. Escribir 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.

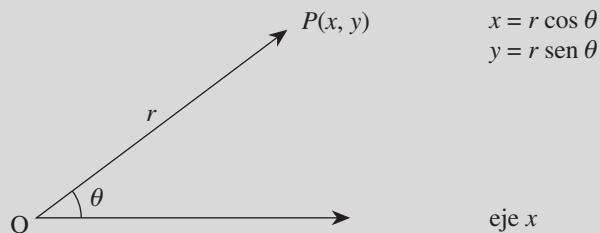
- 6.10.** Se define el número combinatorio m sobre n de la siguiente forma: $\binom{m}{n} = \frac{m!}{n!(m-n)!}$. Escribir un programa que lea los valores de m y de n y calcule el valor de m sobre n .

- 6.11.** Dado un número real p entre cero y uno, 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:

$$\text{Probabilidad}(X = i) = \binom{n}{i} p^i (1-p)^{n-i}$$

Escribir un programa que lea los valores de p , n e i , y calcule el valor dado por la función `binomial`.

- 6.12.** Escribir un programa utilice una función para convertir coordenadas polares a rectangulares.



- 6.13.** La ley de probabilidad de que ocurra el suceso r veces de la distribución de Poisson de media m viene dado por:

$$\text{Probabilidad}(X = r) = \frac{m^r}{r!} e^{-m}$$

Escribir 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.

- 6.14.** Escribir funciones que calculen el máximo común divisor y el mínimo común múltiplo de dos números enteros.

- 6.15.** Escribir funciones para leer fracciones y visualizarlas, representando las fracciones por dos números enteros numerador y denominador.

- 6.16.** Escribir un programa que mediante un menú permita gestionar las operaciones suma resta, producto y cociente de fracciones.

- 6.17.** La descomposición en base 2 de todo número, permite en particular que todo número en el intervalo $[1,2]$, se pueda escribir como límite de la serie $\sum_{i=1}^n \pm sg(i) \frac{1}{2^i}$ 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:

$$\sum_{i=1}^n sg(i) \frac{1}{2^i} > x$$

en caso contrario, $sg(n+1) = -1$.

Escribir 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).

- 6.18.** La función seno (*sen*) viene definida mediante el siguiente desarrollo en serie.

$$\text{sen}(x) = \sum_{i=0}^n \frac{x^{2i+1}}{(2i+1)!}$$

Escribir 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. Ejecute la función en un programa con varios valores de prueba.

- 6.19.** La función coseno viene definida mediante el siguiente desarrollo en serie de Taylor.

$$\cos(x) = \sum_{i=0}^n \frac{x^{2i}}{(2i)!}$$

Escribir 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.

- 6.20.** La función *clotoide* viene definida por el siguiente desarrollo en serie, donde A y Θ son datos.

$$x = A\sqrt{2\Theta} \sum_{i=0}^n (-1)^i \frac{\Theta^{2i}}{(4i+1)(2i)!} \quad y = A\sqrt{2\Theta} \sum_{i=0}^n (-1)^i \frac{\Theta^{2i+1}}{(4i+3)(2i+1)!}$$

Escribir un programa que 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 $1 e^{-10}$.

SOLUCIÓN DE LOS EJERCICIOS

- 6.1.** Usando la función *isdigit(c)* y teniendo en cuenta que devuelve un valor distinto de cero si es un dígito comprendido entre 0 y 9, y 0 en otro caso, una codificación es la siguiente:

```
bool esdigito( char c )
{
    return isdigit(c);
}
```

- 6.2.** Usando el operador % su codificación es:

```
bool divisible( int m, int n )
{
    return m % n;
}
```

- 6.3.** Una codificación es:

```
int Fahrenheit(int celsius)
{
    return 9 * celsius / 5 + 32;
}
```

- 6.4.** Se comprueba si es un carácter alfabético no numérico, en cuyo caso se convierte la letra en mayúscula, para posteriormente comprobar la condición.

```
bool Esvocal( char c)
{
    if (isalnum(c) && !isdigit(c))
        c = toupper(c);
    return ( c == 'A'||c == 'E'||c == 'I'||c == 'O'||c == 'U');
```

SOLUCIÓN DE LOS PROBLEMAS

- 6.1.** Un programa principal lee los dos números asegurándose que son positivos mediante un bucle `do-while` y llama a la función `resto` que se encargará de resolver el problema. La función `resto`, en primer lugar, 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` más el valor de la variable `menor`.

La codificación de este problema se encuentra en la página Web del libro.

- 6.2.** Mediante un acumulador `acu` inicializado a la variable `num2` y un contador `c`, inicializado a 0, se cuenta el número de veces que es necesario sumar `num2` para sobrepasar (ser estrictamente mayor) al número `num1`. Como `c` se ha inicializado a 0, cuando en el acumulador `acu` ya se ha sumando una vez `num2`, el resultado final pedido será el dado por el contador `c`.

```
int cociente(int num1, int num2)
{
    int c = 0, acu = num2;

    while (acu <= num1)
    {
        acu += num2;
        c++;
    }
    return c;
}
```

- 6.3.** El programa principal lee el valor de `n`, y llama sucesivamente a `funciony`. El valor de la `funciony` se obtiene mediante un bucle `for` que itera calculando en cada iteración el siguiente valor de la función. Para ello, almacena en la variable `x` el valor de `funciony(n - 3)`, en `y` el valor de `funciony(n - 2)`, y en `z` el valor de `funciony(n - 1)`. Para calcular el valor de `funciony(n)` basta con realizar la asignación `aux = x + 3 * y - z`, y reasignar los nuevos valores `x`, `y`, `z` para la siguiente iteración.

La codificación de este problema se encuentra en la página Web del libro.

- 6.4.** 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 tanto, devuelve el valor de la expresión `acu == n`.

Codificación

```
bool perfecto(int n)
{
    int i, acu = 0;

    for(i = 1; i < n; i++)
        if (n % i == 0)
            acu += i;
    return (acu == n);
}
```

- 6.5.** Para resolver el problema basta con usar la función *divisores* implementada en el Ejemplo 6.4 y escribir la función *amigos* que detecta la condición.

$(n == \text{divisores}(m)) \&\& (m == \text{divisores}(n))$. Se codifica la función *divisores*.

```
bool divisores(int n)
{
    int i, acu = 0;

    for(i = 1; i < n; i++)
        if (n % i == 0)
            acu += i;
    return acu;
}

bool amigos (int n, int m)
{
    return ((n == divisores(m)) && (m == divisores(n)));
}
```

- 6.6.** Se usa el archivo de cabecera *math.h*. La función recibe el ángulo en grados, lo transforma a radianes para presentar los resultados. El programa principal lee el ángulo positivo y llama a la función trigonométrica.

La codificación de este problema se encuentra en la página Web del libro.

Un resultado de ejecución es:

```
Introduzca valor del angulo en grados > 0:30
seno de 30 = 0.5
coseno de 30 = 0.866025
tangente de 30 = 0.57735
secante de 30 = 1.1547
cosecante de 30 = 2
cotangente de 30 = 1.73205
Presione una tecla para continuar . . . -
```

- 6.7.** Un número entero positivo n 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 **dos** hasta uno menos del 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 $2^2 = 64$. De esta forma para decidir si un número es primo basta con comprobar si tiene divisores positivos menores o iguales que su raíz cuadrada, por supuesto eliminando la unidad.

```

bool primo(int n)
{
    int i, tope;
    bool p = true;

    tope = (int)sqrt(n);
    i = 2;
    while (p && (i <= tope))
    {
        p = !(n % i == 0);
        i++;
    }
    return (p);
}

```

- 6.8.** Las fórmulas que dan las coordenadas x e y del proyectil son:

$$\begin{aligned}x &= v * \cos(\alpha) * t \\y &= v * \sin(\alpha) - a * t^2 / 2\end{aligned}$$

donde α es un ángulo que está en el primer cuadrante v es la velocidad inicial y $a = 40m/s^2$ es la aceleración.

La función recibe como parámetros al ángulo α en grados sexagesimales y la velocidad inicial v . Convierte el ángulo a un ángulo equivalente en radianes, inicializa las variables t e y , e itera en un bucle `while` hasta que se obtiene un valor negativo de la ordenada y .

La codificación de este problema se encuentra en la página Web del libro.

- 6.9.** El programa se codifica de la siguiente forma:

La función `menu`, se encarga de devolver un carácter entre las opciones 1 a 4.

El programa principal llama a la función `menu`, lee los datos, y realiza las distintas llamadas a las funciones. Se iter a mediante un bucle `for` que termina cuando `menu` retorna la opción 4.

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:

$$tt = \frac{\log\left(1 + \frac{cr}{a(1+r)}\right)}{\log(1+r)}$$

La codificación de este problema se encuentra en la página Web del libro.

- 6.10.** El programa se codifica usando la función `factorial`, que calcula el producto de los x primeros números naturales positivos y programando la función `combinatorio`, con sus correspondientes llamadas a la función `factorial`. El programa principal se encarga de leer los datos m y n y de llamar a la función `combinatorio`.

La codificación de este problema se encuentra en la página Web del libro.

- 6.11.** En el problema 6.10 se ha codificado la función combinatorio. Se programa la función potencia iterativamente multiplicando el valor de su parámetro a por sí mismo tantas veces como indica el parámetro n . La función binomial realiza las llamadas correspondientes a la función combinatorio y potencia. El programa principal, realiza la lectura de p , n e i dentro de los rangos indicados, y llama a la función binomial.

La codificación de este problema se encuentra en la página Web del libro.

- 6.12.** La función polares, transforma las coordenadas polares r , ζ en coordenadas cartesianas x , y , mediante las fórmulas correspondientes. El programa principal lee los valores del ángulo ζ y el módulo r positivos, de las coordenadas polares, llama a la función polares, para obtener las coordenadas cartesianas y presenta los resultados.

La codificación de este problema se encuentra en la página Web del libro.

Un resultado de ejecución es:

```
Introduzca valor de r > 0 y zeta en grados >0:30 26
coordenadas cartesianas
x = 26.964 y = 13.1508
Presione una tecla para continuar . . .
```

- 6.13.** Se programa una función menu que elige entre las tres opciones correspondientes.

Se programa una función Poisson que calcula Probabilidad($X = r$) = $\frac{m^r}{r!} e^{-m}$ y la Probabilidad($X \leq r$) = $\sum_{i=0}^r \frac{m^i}{i!} e^{-m}$ que resuelven el apartado a y el b. Estos valores lo retornan en sus parámetros por referencia, x_igual_r , x_menor_r . En la variable $valor$ se van calculando los sucesivos valores de $\frac{m^0}{0!} e^{-m}$, $\frac{m^1}{1!} e^{-m}$, $\frac{m^2}{2!} e^{-m}$, ..., $\frac{m^r}{r!} e^{-m}$, y en la variable acu se van acumulando los valores, por lo que al final del bucle for, los resultados están almacenados en las variables $valor$ y acu . 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 a menu.

La codificación de este problema se encuentra en la página Web del libro.

- 6.14.** El algoritmo de Euclides para calcular el máximo común divisor de dos números enteros positivos n y d es el siguiente: calcular el resto r de la división entera de n entre d . Si el resto es 0 parar y en otro caso: asignar a n el valor de d ; a d el valor de r e iterar. La función mcd recibe dos números enteros n y d con $d \neq 0$, los convierte en positivo y aplica el algoritmo de Euclides. El cálculo del mínimo común múltiplo mcm, se calcula usando la siguiente propiedad: el máximo común divisor mcd multiplicado por el mínimo común múltiplo mcm de dos números n y d coincide con el producto de los dos números. $mcd * mcm = n * d$.

La codificación de este problema se encuentra en la página Web del libro.

- 6.15.** Las fracciones se representan por dos números enteros numerador n y denominador d , siendo $d > 0$. La función leerfracciones lee dos fracciones realizando dos llamadas a la función leerkf. La función leerkf 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, para lo que necesita llamar a la función simplificaf. Esta función simplificaf, simplifica una fracción haciendo que el numerador y el denominador de la fracción sean primos entre sí, dejando siempre el signo negativo, en caso de que la fracción sea negativa en el numerador. Para hacerlo basta con dividir el numerador y el denominador por el máximo común divisor mcd. La función mcd se ha codificado en el Problema 6.14. La función escribef, recibe como parámetros el numerador n y el denominador d y lo visualiza.

La codificación de este problema se encuentra en la página Web del libro.

- 6.16.** Escribir un programa principal que mediante un menú se encarga de llamar a las correspondientes opciones solicitadas además de la opción de leer fracciones. Se usan las funciones codificadas en los Problemas 6.14, 6.15, de las que se han incluido sus prototipos, no así su codificación que puede consultarse en los dos problemas anteriores. Las funciones pedidas usan las fórmulas que a continuación se exponen, convenientemente simplificadas, para hacer que la fracción resultante sea irreducible. Las cuatro funciones, llevan cuatro parámetros por valor para recibir las dos fracciones, y dos por referencia para retornar el resultado.

$$\text{sumaf}(n1, d1, n2, d2) = \frac{\frac{n1 * \text{mcm}(d1, d2)}{d1} + \frac{n2 * \text{mcm}(d1, d2)}{d2}}{\text{mcm}(d1, d2)}$$

$$\text{retaf}(n1, d1, n2, d2) = \frac{\frac{n1 * \text{mcm}(d1, d2)}{d1} - \frac{n2 * \text{mcm}(d1, d2)}{d2}}{\text{mcm}(d1, d2)}$$

$$\text{multiplicaf}(n1, d1, n2, d2) = \frac{n1 * n2}{d1 * d2}$$

$$\text{dividef}(n1, d1, n2, d2) = \frac{n1 * d2}{n2 * d1}$$

La codificación de este problema se encuentra en la página Web del libro.

- 6.17.** 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:

- Alog2 que calcula el logaritmo en base 2 de cualquier x . Esta función llama a las funciones: alog01, si x está en el intervalo $(0,1)$; alog12, si x está en el intervalo $[1,2]$; o bien alog1i si x es mayor que 2. Si x es menor o igual que 0 da un mensaje de error, ya que no existe el logaritmo.
- Alog12 se programa teniendo en cuenta el desarrollo de la serie en el intervalo $[1,2]$. En la variable término se van obteniendo los correspondientes términos de la serie, y en suma se van acumulando los correspondientes términos. Para controlar el signo del siguiente término a sumar, se usa la expresión $\exp(\log(2) * \text{suma}) \leq x$, que es equivalente a la dada en la fórmula del enunciado.
- La función signo no se codifica, ya que queda implícita en el cálculo de la función alog12.
- 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. Un bucle va dividiendo por dos x , y sumando 1 al acumulador acu, hasta conseguir que x esté en el intervalo $[1,2]$.
- 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 biblioteca log() de biblioteca, para comprobar los resultados.

La codificación de este problema se encuentra en la página Web del libro.

- 6.18.** Las fórmula que calcula los valores de $\sin(x)$ puede obtenerse de la siguiente forma:

$$\sin(x) = t1 + t3 + t5 + t7 + t9 + \dots$$

$$\text{donde } t1 = x \quad y \quad t_i = -\frac{x * x}{i(i-1)} t_{i-2}$$

La función seno se programa teniendo en cuenta las fórmulas anteriores y, además, la parada se realiza, cuando se han sumado, como máximo 20 términos ($i==20$), o el siguiente término a sumar tiene un valor absoluto menor que una cota de error que se le pasa como parámetro. La variable xx contiene $x*x$ para ahorrar en cada iteración un producto.

El programa principal lee los valores de `valor1`, `valor2`, `incremento`, y `cota de error` y llama a la función seno de biblioteca y la compara con el valor calculado mediante la función programada, para los valores:

`valor1, valor1+incremento, valor1 + 2 * incremento, ...`

hasta que se sobrepase el `valor2`.

Como la función seno de biblioteca trabaja con radianes, se define una variable `propor` que debidamente inicializada $3.1415/180$ transforma los grados en radianes. Si `valor1` está en grados, entonces `valor1*propor` está en radianes.

La codificación de este problema se encuentra en la página Web del libro.

- 6.19.** La fórmula que calcula los valores de $\cos(x)$ puede obtenerse de la siguiente forma:

$$\cos(x) = t_0 + t_2 + t_4 + t_6 + t_8 + \dots$$

$$\text{donde } t_0 = 1 \quad \text{y} \quad t_i = -\frac{x * x}{i(i-1)} t_{i-2}$$

La función `coseno` se programa de una forma semejante a la función `seno`, sólo se diferencia en la inicialización del término `term` y del acumulador `suma`.

```
float coseno( float x, float error)
{
    float term = 1.0, suma = 1.0, xx = x*x;
    int i = 0;

    while (fabs( term ) > error && i < 20)
    {
        i += 2;
        term = -term * xx/(i * (i - 1));
        suma = suma + term;
    }
    return(suma);
}
```

- 6.20.** Las fórmulas que calculan los distintos valores de x y de y , pueden obtenerse de la siguiente forma:

$$x = t_0 - t_2 + t_4 - t_6 + t_8 - \dots$$

$$y = t_1 - t_3 + t_5 - t_7 + t_9 - \dots$$

$$\text{siendo } t_0 = A\sqrt{2\Theta} \quad y \quad t_i = \frac{2i-1}{i(2i+1)} \Theta t_{i-1}$$

Se programa la función `term`, que calcula t_i en función de t_{i-1} , i , θ .

La función `clotoide` recibe como datos el valor de a y el de θ y devuelve como resultado los valores de x e y . En cada iteración del bucle se recalculan dos términos que son acumulados respectivamente a los acumuladores x e y , por lo que en cada iteración hay que cambiar de signo el término. Es decir, en primer lugar se suman dos positivos, en la siguiente iteración, dos negativos, en la siguiente dos positivos y así sucesivamente. Antes de comenzar una iteración del bucle se tiene ya calculado un valor del siguiente término para sumar, que es acumulado en la variable x . Posteriormente, se calcula el siguiente término impar y se acumula a la variable y , para terminar calculando de nuevo el siguiente término par. El bucle termina cuando el valor absoluto del siguiente término a sumar sea menor que ϵ .

El programa principal, calcula el valor de π mediante la función `arcocoseno`, ($\pi = \acos(-1.0)$) para posteriormente, mediante un bucle, realizar las distintas llamadas a la función `clotoide`.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 6.1. Escribir una función Redondeo que acepte un valor real Cantidad y un valor entero Decimales y devuelva el valor Cantidad redondeado al número especificado de Decimales. Por ejemplo, Redondeo (20.563,2) devolverá los valores 20.0, 20.5 y 20.54 respectivamente.
- 6.2. Escribir un programa que permita al usuario elegir el cálculo del área de cualquiera de las figuras geométricas: círculo, cuadrado, rectángulo o triángulo, mediante funciones.
- 6.3. Determinar y visualizar el número más grande de dos números dados, mediante un subprograma.
- 6.4. Codifique una función que escriba n líneas en blanco.

PROBLEMAS PROPUESTOS

- 6.1. Escribir una función que calcule la media de un conjunto de $n > 0$ números leídos del teclado.
 - 6.2. Escribir una función que decida si un número entero es capicúa. El número 24842 es capicúa. El número 134 no lo es.
 - 6.3. Escribir un programa que encuentre el valor mayor, el valor menor y la suma de los datos de entrada. Obtener la media de los datos mediante una función.
 - 6.4. Escribir una función que permita calcular la serie:

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$= n * (n+1) * (2 * n + 1) / 6$$
 - 6.5. Escribir una función que sume los 30 primeros números impares.
 - 6.6. Escribir una función que calcule la suma de los 20 primeros números primos.
 - 6.7. Escribir una función que encuentre y escriba todos los números perfectos menores que un valor constante max.
 - 6.8. Escribir un programa que lea dos números x y n y calcule la suma de la progresión geométrica: $1 + x + x^2 + x^3 + \dots + x^n$
 - 6.9. Escribir un programa que mediante funciones determine el área del círculo correspondiente a la circunferencia circunscrita de un triángulo del que conocemos las coordenadas de los vértices.
 - 6.10. Escribir 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.
 - 6.11. Escribir un programa que solicite del usuario un carácter y que sitúe ese carácter en el centro de la pantalla. El usuario debe poder a continuación desplazar el carácter pulsando las letras A (arriba), B (abajo), I (izquierda), D (derecha) y F (fin) para terminar.
 - 6.12. Escribir una función que determine si una cadena de caracteres es un palíndromo (un palíndromo es un texto que se lee igual en sentido directo y en inverso: radar).
 - 6.13. Escribir el inverso de un número entero dado (1234, inverso 4321).
 - 6.14. Escribir un programa mediante una función que acepte un número de día, mes y año y lo visualice en el formato: dd/mm/aa. Por ejemplo, los valores 8, 10 y 1946 se visualizan como: 8/10/46.
 - 6.15. Escribir un programa, mediante funciones, que visualice un calendario de la forma:

L	M	M	J	V	S	D
			1	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	23
27	28	29	30			
- El usuario indica únicamente el mes y el año. La fórmula que permite conocer el día de la semana correspondiente a una fecha dada es:

a) Meses de enero o febrero $n = a + 31 * (m - 1) + d(a - 1) \text{ div } 4 - 3 * ((a + 99) \text{ div } 100) \text{ div } 4;$

b) Restantes meses $n = a + 31 * (m - 1) + d - (4 * m + 23) \text{ div } 10 + a \text{ div } 4 - (3 * (a \text{ div } 100 + 1)) \text{ div } 4;$

donde $a = \text{año}$, $m = \text{mes}$, $d = \text{día}$.

Nota: $n \bmod 7$ indica el día de la semana (1 = lunes, 2 = martes, etc.).

6.16. Escribir una función con dos parámetros, x y n , que devuelva lo siguiente:

$$x + \frac{x^n}{n} - \begin{cases} \frac{x^{n+2}}{n+2} & x \geq 0 \\ \frac{x^{n+1}}{n+1} - \frac{x^{n-1}}{n-1} & x < 0 \end{cases}$$

CAPÍTULO 7

Arrays (arreglos, listas o tablas)

Introducción

En este capítulo aprende el concepto y tratamiento de los arrays. Un **array (arreglo)** 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 de ellas 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.

7.1. Arrays (arreglos)

Un **array (arreglo, lista o tabla)** es una secuencia de objetos del mismo tipo, que se numeran consecutivamente 0, 1, 2, 3. Estos números se denominan *valores índice* o *subíndice* del array. Cada ítem del array se denomina *elemento*. El tipo de elementos del array puede ser cualquier tipo de dato de C++, incluyendo estructuras definidas por el usuario. Si el nombre del array es *a*, entonces *a[0]* es el nombre del elemento que está en la posición 0, *a[1]* es el nombre del elemento que está en la posición 1, etc. En general, el elemento *i*-ésimo está en la posición *i*-1. De modo que si el array tiene *n* elementos, sus nombres son *a[0], a[1], ..., a[n-1]*.

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 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];
```

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*. Todos los subíndices de los arrays comienzan con 0. 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[4]* está a 4 pasos o posiciones del elemento *a[0]*. En los programas se pueden referenciar elementos del array utilizando fórmulas o expresiones enteras para los subíndices.

Los elementos de los arrays se almacenan en bloques contiguos de memoria. Los arrays que tienen un solo subíndice se conocen como arrays *unidimensionales* (una sola dimensión). Cada bloque de memoria tiene el tamaño del tipo de dato. Hay que tener en cuenta que C++ no comprueba que los índices del array están dentro del rango definido, por lo que debe ser controlado por el usuario. Se debe tener cuidado de no asignar valores fuera del rango de los subíndices, debido a que se sobreescritirían datos o código.

EJEMPLO 7.1. Posiciones válidas de un array. En la declaración del array `int a[7]` los índices válidos son `a[0], a[1], ..., a[6]`. Pero si se pone `a[10]` no se da mensaje de error y el resultado puede ser impredecible.

EJEMPLO 7.2. Posiciones ocupadas por los elementos de un array. Si `a` es un array de números reales y cada número real ocupa 4 bytes, entonces si el elemento `a[0]` ocupa la dirección del elemento `a[i]`, ocupa la dirección de memoria $d + (i - 1) * 4$.

El operador `sizeof` devuelve el número de bytes necesarios para contener su argumento. Al usar `sizeof` para solicitar el tamaño de un array, se obtiene el número de bytes reservados para el array completo. Conociendo el tipo de dato almacenado en el array y su tamaño, se puede calcular la longitud del array, mediante el cociente `sizeof(a)/tamaño(dato)`.

EJEMPLO 7.3. Protección frente a errores en el intervalo (rango) de valores de una variable de índice que representa un array. Si `n` está fuera de rango se retorna error, en otro caso se calcula la media de los números almacenados en el array.

```
float media (double m[], int n)
{
    if (n * sizeof(float) > sizeof(m))
        return -32767; // error
    float Suma = 0;

    for(int i = 0; i < n; i++)
        Suma += m[i];
    return Suma / n;
}
```

EJEMPLO 7.4. Almacenamiento en memoria de un array de 5 elementos. Se declara el array `a` de 5 elementos y se almacenan en las posiciones de memoria los 5 primeros números pares positivos.

```
int a[4];

for(int i = 0; i < 5; i++)
    a[i] = 2 * i + 2;
```

Posición índice	0	1	2	3	4
Valor almacenado	2	4	6	8	10

7.2. Inicialización de arrays

Antes de que un array sea usado se deben asignar valores a sus elementos, tal como se asignan valores a variables. Para asignar valores a cada elemento del array de 4 enteros `a`, se puede escribir:

```
a[0] = 0; a[1] = 10; a[2] = 20; a[3] = 30
```

La primera sentencia pone en `a[0]` el valor 0, en `a[1]` el valor 10, etc. Sin embargo, este método no es práctico cuando el array contiene muchos elementos. En este caso pueden usarse sentencias de asignación en bucles.

EJEMPLO 7.5. El siguiente fragmento de programa declara un array de 10 enteros lo inicializa, y lo presenta.

```
#define max 50
int a[max];

for(int i = 0; i < max; i++)
    a[i] = 10 * i;
```

```
for(int i = 0; i < max; i++)
    cout << a[i] << endl;
```

Un método utilizado, para inicializar un array completo es hacerlo en una sentencia de inicialización, en la que se incluyen las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves.

```
int vector[] = {15, 25, -45, 0, 50, 12, 60};
El compilador asigna automáticamente 7 elementos a vector.
```

EJEMPLO 7.6. Declaraciones e inicialización simultánea de arrays.

```
int num[5]={10,20,30,40,50};      //declara e inicializa un array de 5 enteros
float x[ ] = {1.0,22.0,3.5}      //declara e inicializa un array de 3 reales
// Se pueden asignar constantes simbólicas como valores numéricos
const int ENE = 31, FEB = 28, MAR = 31, ABR = 30, MAY = 31, JUN = 30,
       JUL = 31, AGO = 31, SEP = 30, OCT = 31, NOV = 30, DIC = 31;
int meses[12] = {ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT,
                 NOV, DIC};
```

Arrays de caracteres y cadenas de texto

Una **cadena de texto** es un conjunto de caracteres, tales como "abcdef". Las *cadenas* contienen *carácter nulo* (`\0`), cuyo valor en el código ASCII es 0 al final del array de caracteres. El medio más fácil de inicializar un array de caracteres es hacer la inicialización de la declaración.

EJEMPLO 7.7. Inicialización de arrays de caracteres y cadenas en la declaración.

```
char ch[] = {'L','u','i','s',' ', 'y',' ','L','u','c','a','s'};
//Declara un array de 12 caracteres
char ch[] = {"Ejemplo"}
char cadena[] = "abcdef";

//Los arrays pueden inicializarse con una constante cadena. El array
//anterior tiene 7 elementos, ya que se almacena el carácter nulo \0.
```

Las cadenas se deben almacenar en arrays de caracteres, pero no todos los arrays de caracteres contienen cadenas. Sólo son cadenas aquellas que terminan en el carácter nulo.

EJEMPLO 7.8. Asignación de cadenas de caracteres. Las cadenas de caracteres pueden asignarse en la inicialización de un array mediante la función de la biblioteca estándar strcpy() ("copiar cadenas") permite copiar una constante de cadena en una cadena. Sin embargo, no puede asignarse un valor a una cadena con la asignación =.

```
char Cadena[6];
strcpy(Cadena,"abcde");
```

Es equivalente a la siguiente declaración.

```
char Cadena[6] = "abcde";
char Cadena[7] = "abcde";
0   1   2   3   4   6
Cadena | a | b | c | d | e | \0
```

Las sentencias anteriores colocan la cadena en memoria tal y como indica la tabla anterior. Pero, no se puede asignar una cadena a un array del siguiente modo:

```
char Cadena[6];
Cadena = "abcde";           //asignación incorrecta
```

7.3. Arrays multidimensionales

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, ésto es, tres, cuatro o más dimensiones. La sintaxis para la declaración de un array de dos dimensiones es:

```
<TipoElemento><nombarray>[<NúmeroDeFilas>][<NúmeroDeColumnas>]
```

Un array de dos dimensiones en realidad es un *array de arrays*. Es decir, es un array unidimensional, y cada elemento no es un valor entero de coma flotante o carácter, sino que cada elemento es otro array.

La sintaxis para la declaración de un array de tres dimensiones es:

```
<tipodedatoElemento><nombarray> [<Cota1>] [<Cota2>][<Cota3>]
```

EJEMPLO 7.9. Declaración y almacenamiento de un array bidimensional en memoria. La declaración de un array de enteros que tiene 4 filas y 6 columnas, su representación y almacenamiento por filas es el siguiente:

```
int a[4][6];           //la declaración se almacena en memoria por filas
```

Tabla 7.1. Almacenamiento de Array bidimensional de 4 filas y 6 columnas

A[5][6]	C	o	1	u	m	n	a
f	a[0][0]	a[0][1]	.	.	.	a[0][5]	a[0][6]
i
l
a	a[3][0]	a[3][1]	.	.	.	a[3][5]	a[3][6]

a[0][0], a[0][1], a[0][2],..., a[0][5], a[0][6], a[1][0], a[1][1],
a[1][2],..., a[1][5] a[1][6],....., a[3][0], a[3][1],
a[3][2],..., a[3][5], a[3][6].

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.

EJEMPLO 7.10. Inicialización de arrays bidimensionales en la declaración.

```
int ejemplo[2][3] = {1,2,3,4,5,6};
int ejemplo [2][3] = {{1,2,3}, {4,5,6}}
```

Los arrays multidimensionales (si no son globales) no se inicializan a valores específicos a menos que se asignen valores en el momento de la declaración o en el programa. Si se inicializan uno o más elementos, pero no todos, C++ rellena el resto con ceros o valores nulos ('\0'). Si se desea inicializar a cero un array multidimensional, utilice una sentencia tal como ésta: `float a[3][4] = {0.0};`

El formato general para asignación directa de valores a los elementos de un array bidimensional es la siguiente:

```
<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];
```

EJEMPLO 7.11. *Lectura y visualización de un array de 3 filas y 5 columnas. Se declaran las constantes maxf y maxc a 5 y, posteriormente, se declara el array bidimensional a y se lee y visualiza por filas.*

```
#include <cstdlib>
#include <iostream>
#define maxf 3
#define maxc 5
using namespace std;

int main(int argc, char *argv[])
{
    float a[maxf] [maxc];
    int f, c;
    // leer el array
    for(f = 0; f < maxf; f++)
        for(c = 0; c < maxc; c++)
            cin >> a[f][c];

    // escribir el array
    for(f = 0; f < maxf; f++)
    {
        for(c = 0; c < maxc; c++)
            cout << a[f] [c] ;
        cout << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJEMPLO 7.12. *Leer una matriz cuadrada de orden 5 (5 filas y 5 columnas), y escribir la suma de todos los números que no están en la diagonal principal. El problema se resuelve en un solo programa principal. Dos bucles for anidados leen la matriz, otros dos bucles for 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 es distinto de j, siendo i y j los índices de la fila y columna respectivamente.*

```
#include <cstdlib>
#include <iostream>
#define filas 5
using namespace std;

int main(int argc, char *argv[])
{
    int i,j, suma, A[filas][filas];
                                // lectura por filas
    for(i = 0; i < filas; i++)
        for(j = 0; j < filas; j++)
            cin >> A[i][j];
                                //realización de la suma
    for(i = 0; i < filas; i++)
        for (j = 0; j < filas; j++)
            if(!(i == j))
                suma += A[i][j];
    cout << " suma " << suma;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJEMPLO 7.13. *Array tridimensional. Se declara un array tridimensional para almacenar las temperaturas de cada uno de los 60 minutos de las 24 horas de un mes de 31 días. Se declaran como constantes días, horas y minutos. En tres bucles anidados se leen la temperatura de cada uno de los 60 minutos, de las 24 horas, de los 31 días de un mes. Posteriormente, se calcula y se escribe la media de temperatura de cada día.*

```
#include <cstdlib>
#include <iostream>
#define dias 31
#define horas 24
#define minutos 60
using namespace std;

int main(int argc, char *argv[])
{
    int i, j, k;
    float A[dias][horas][minutos], media;
                                //lectura de las temperaturas.
    for(i = 0; i < dias; i++)
        for (j = 0; j < horas;j++)
            for (k = 0; k < minutos; k++)
                cin >> A[i][j][k];

    for(i = 0; i < dias; i++)
    {
                                //cálculo de la media de cada uno de los días.
        media = 0;
        for ( j = 0 ; j < horas; j++)
            for ( k = 0; k < minutos; k++)
                media += A[i][j][k];
        cout<<" dia " << i+1 << " " << media /(horas * minutos) << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

7.4. Utilización de arrays como parámetros

En C++ *todos los arrays se pasan por referencia* (dirección). C++ trata automáticamente la llamada a la función como si hubiera situado el operador de dirección & delante del nombre del array.

La declaración en la función de que un parámetro es un array se realiza:

```
<tipo de datoElemento> <nombre array> [<Cota1>]. O mejor
<tipo de datoElemento> <nombre array> []. O bien
<tipo de datoElemento> *<nombre array>
```

En este segundo caso es conveniente declarar otro parámetro que indique el tamaño del array.

EJEMPLO 7.14. Declaración de funciones con parámetros array, y llamadas.

```
float suma(float a[5]);
float calcula(float a[], int n); //n es número de datos no obligatorio
float media (float * a, int n); //n es no obligatorio, pero conveniente
```

Dadas la declaraciones de arrays siguientes:

```
int b[5], a[6];
```

Son posibles llamadas válidas:

```
cout << suma(a);
cout << calcula (b, 5);
cout << media (b, 5);
```

EJEMPLO 7.15. Lectura escritura de un vector con funciones. El número de elementos del vector es indicado a cada función en el parámetro *n*, y el vector se declara en el parámetro con corchetes vacíos. El programa declara una constante *max* para dimensionar los vectores. Se codifican las funciones *lee*, *escribe*, que leen y escriben respectivamente un vector de *n* datos que reciben como parámetros, así como la función *suma* que recibe como parámetros dos vectores de *n* datos, y calcula el vector *suma* almacenado el resultado en *c*. El programa principal llama a las funciones anteriores.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#define max 11
using namespace std;

void lee( float a[], int n);
void escribe (float a[], int n);
void suma(float a[], float b[], float c[], int n);

int main(int argc, char *argv[])
{
    int n;
    float a[max], b[max],c[max];

    n = 3;
    lee(a, n);
    lee(b, n);
    cout << " vector a\n";
    escribe(a, n);
```

```

cout << " vector b\n";
escribe(b, n);
suma (a, b, c, n);
cout << " vector suma \n";
escribe(c, n);
system("PAUSE");
return EXIT_SUCCESS;
}

void suma( float a[], float b[], float c[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

void escribe(float a[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        cout << a[i] << "      ";
    cout << endl;
}

void leer(float a[], int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        cout << " dame dato posicion i =" << i + 1 << "  ";
        cin >> a[i];
    }
}

```

EJEMPLO 7.16. Las funciones *cero* y *resta*, reciben como parámetros vectores, pero el número de elementos del vector a tratar viene dado por la constante global *max* que es usada para el control de los bucles internos. En los parámetros de las funciones el vector se declara con el número de elementos que tiene mediante la constante *max* entre corchetes. La función *cero* rellena de ceros el vector. La función *resta* calcula la diferencia de los vectores que recibe como parámetro.

```

#include <cstdlib>
#include <iostream>
#include <math.h>           //librería de funciones matemáticas
#define max 10

void resta( float a[max], float b[max], float c[max]);
void cero(float c[max]);

int main(int argc, char *argv[])
{
    float a[max], b[max], c[max];

```

```

cero (a);
.....
.....
resta (a, b, c);
system("PAUSE");
return EXIT_SUCCESS;
}

void cero (float c[max])
{
    int i;

    for (i = 0; i < max; i++)
        c[i] = 0;
}

void resta( float a[max], float b[max], float c[max])
{
    // el vector c es la diferencia de a y b; c = a - b.
    int i;

    for(i = 0; i < max; i++)
        c[i] = a[i]- b[i];
}

```

EJEMPLO 7.17. Función que lee los elementos de un vector. El número de elementos leídos se devuelve por la función en el parámetro por referencia *n*. La constante global *max*, es usada por la función para asegurar que los datos leídos están dentro del rango permitido. El bucle de lectura de datos es interrumpido cuando se lee el valor de cero mediante el uso de una sentencia *if* y una sentencia *break*.

```

#define max 10

void leerArray(int a[], int& n)
{
    cout << "Introduzca datos. Para terminar pulsar 0:\n";
    for(n = 0; n < max; n++)
    {
        cout " dato " << n << ":" ;
        cin >> a[n];
        if (a[n] == 0)
            break;
    }
}

```

EJEMPLO 7.18. Función que calcula la media cuadrática y armónica de un vector que recibe como parámetro de *n* elementos. La media cuadrática *mc* y media armónica *ma* de un vector de *n* elementos vienen dadas por

$$\text{las expresiones: } mc = \sqrt{\frac{\sum_{i=0}^{n-1} a(i)^2}{n}} \quad ma = \frac{n}{\sum_{i=0}^{n-1} \frac{1}{a(i)}}$$

```

float mc ( float a[], int n)
{
    float aux = 0;

```

```

    for(int i = 0; i < n; i++)
        aux += a[i] * a[i];
    return sqrt(aux / n);
}

float ma(float a[], int n)
{
    float aux = 0;

    for( int i = 0; i < n; i++)
        aux += 1 / a[i];
    return n / aux;
}

```

Paso de matrices como parámetros

Para arrays bidimensionales es obligatorio indicar el número de columnas en la declaración del numero de parámetros de la siguiente forma:

`<tipo de datoElemento> <nombre array> [<Cota1>][<Cota2>]. O bien,`
`<tipo de datoElemento> <nombre array> [][][][cota2]`

En este segundo caso es también conveniente declarar otro parámetro que indique el número de filas. La llamada a una función que tenga como parámetro un array se realiza con el nombre del array .

EJEMPLO 7.19. Declaración de funciones con parámetros array bidimensionales, y llamadas.

```

void matriz (float M[3][5]);
void matriz1(float M[][5], int nf); //nf no obligatorio, pero conveniente

```

Dada la declaración siguiente:

```
float M1 [3][5];
```

Son posibles llamadas válidas:

```

matriz (M1);
matriz1(M1, 3);

```

EJEMPLO 7.20. Funciones que reciben como parámetro dos matrices cuadradas las suman y las multiplican. La suma y producto de matrices vienen definidas por las expresiones:

$$\text{Suma} \quad c(i, j) = a(i, j) - b(i, j)$$

$$\text{Producto} \quad c(i, j) = \sum_{k=0}^{n-1} a(i, k) * b(k, j)$$

Para realizar la suma basta con recorrer con los índices las posiciones de la matriz `c` y sumar las correspondientes posiciones de la `a` y la `b`.

Para realizar el producto, hay que acumular en `acumulador` el valor de la suma, y el resultado almacenarlo en la posición `c[i][j]`.

```

#define max 11

void Suma(float a[][max] , float b[][max] , float c[][max], int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
}

```

```

void Producto(float a[][][max] , float b[][][max] , float c[][][max], int n)
{
    float acumulador;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
        {
            acumulador = 0;
            for (int k = 0; k < n; k++)
                acumulador += a[i][k] * b[k][j];
            c[i][j] = acumulador;
        }
}

```

EJEMPLO 7.21. Funciones que reciben como parámetro una matriz cuadrada con los dos índices constantes.

Identidad $c(i, j) = \begin{cases} 1 & \xrightarrow{\text{SI}} i = j \\ 0 & \xrightarrow{\text{SI}} i \neq j \end{cases}$

Asigna $a(i, j) = b(i, j)$

```

#define max 20

void Identidad(float c[max][max])
{
    for (int i = 0; i < max; i++)
        for (int j = 0; j < max; j++)
            if (i == j)
                c[i][j] = 1;
            else
                c[i][j] = 0;
}

void asigna( float a[max][max], float b[max][max])
{
    for (int i = 0; i < max; i++)
        for (int j = 0; j < max; j++)
            a[i][j]= b [i][j];
}

```

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 de él, 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:

- situar un valor de señal al final del array, que indique a la función que se ha de detener el proceso en ese momento;
- pasar un segundo argumento que indica el número de elementos del array .

Paso de cadenas como parámetros

La técnica de pasar arrays como parámetros se utiliza para pasar cadenas de caracteres a funciones. Las cadenas terminadas en nulo ('\0') utilizan el primer método dado anteriormente para controlar el tamaño de un array .

EJEMPLO 7.22. Función que recibe como parámetro un número entero como cadena de caracteres y convierte la cadena de caracteres en un número entero. 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 un dato leído previamente. Para convertir la cadena en número se usa la descomposición: $2345 = 2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5$. Estas operaciones se realizan por el método de Horner de evaluación de polinomios: $2345 = ((((0 \cdot 10 + 2) \cdot 10 + 3) \cdot 10 + 4) \cdot 10 + 5)$. La obtención, por ejemplo del número 3 se hace a partir del carácter '3' : $3 = '3' - '0'$.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int valor_numerico(char cadena[]);

int main(int argc, char *argv[])
{
    char cadena[80];

    cout << "dame numero: ";
    cin >> cadena;
    cout << " numero como cadena " << cadena << endl;
    cout << " valor leido como numero \n " << valor_numerico(cadena) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

int valor_numerico(char cadena[])
{
    int i, valor, signo;
                                // salto de blancos y tabularores
    for (i = 0; cadena[i] == ' ' || cadena[i] == '\t'; i++);
                                //determinación del signo
    signo = 1;
    if(cadena[i] == '+' || cadena[i] == '-')
        signo = cadena[i+1] == '+' ? 1:-1;
                                // conversión a número
    for (valor = 0 ; cadena[i] >= '0' && cadena[i] <= '9' ; i++)
        valor = 10 * valor + cadena[i] - '0';
    return (signo * valor);
}
```

EJERCICIOS

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

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i, Primero[21];

    for (i = 1; i <= 6; i++)
        cin >> Primero[i];
    for(i = 3; i > 0; i--)
        cout << Primero[2 * i];
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

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

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i,j,k, Primero[10];

    for(i = 0; i < 10; i++)
        Primero[i] = i + 3;
    cin >> j >> k;
    for(i = j; i <= k; i++)
        cout << Primero[i] << " ";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

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

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i, j ,k, Primero[11], Segundo[11];

    for(i = 0; i < 11; i++)
        Primero[i] = 2 * i + 2;
    for(j = 0; j < 6; j++)
        Segundo[j] = Primero[2 * j] + j;
```

```

for(k = 3; k < 6; k++)
    cout << Primero[k + 1]<< " " << Segundo [k - 1] << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

```

- 7.4.** Escribir una función que rellene un array con los 10 primeros números impares, y visualice los contenidos del vector cuyos índices son: 0, 2, 4, 6, 8.

PROBLEMAS

- 7.1.** Escribir un programa que lea 10 números enteros, los almacene en un vector, y calcule la suma de todos ellos, así como su media aritmética.
- 7.2.** Un vector se dice que es simétrico si el elemento que ocupa 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 . Por ejemplo el vector que almacena los valores 2, 4, 5, 4, 2 es simétrico. Escribir una función que decida si el vector de n datos que recibe como parámetro es simétrico.
- 7.3.** Un vector que almacena n datos se dice que es mayoritario, si existe un elemento almacenado en el vector que aparece en el vector más de $n / 2$ veces. Escribir una función que decida si un vector es mayoritario.
- 7.4.** Diseñar funciones que calculen el producto escalar de dos vectores, la norma de un vector y el coseno del ángulo que forman definidos de la siguiente forma:

$$pe(a, b, n) = \sum_{i=0}^{n-1} a(i) * b(i) \quad norma(a, n) = \sqrt{\sum_{i=0}^{n-1} a(i) * a(i)}$$

$$\cos(a, b, n) = \frac{\sum_{i=0}^{n-1} a(i) * b(i)}{\sqrt{\sum_{i=0}^{n-1} a(i) * a(i) * \sum_{i=0}^{n-1} a(i) * b(i)}}$$

- 7.5.** Escribir un algoritmo que calcule y escriba una tabla con los 100 primeros números primos. Un número es primo si sólo tiene por divisores la unidad y el propio número.
- 7.6.** Escribir un programa que genere aleatoriamente los datos de un vector, lo visualice, y calcule su media m , su desviación media dm su desviación típica dt , dadas por las siguientes expresiones:

$$m = \left(\sum_{i=0}^{n-1} a(i) \right) / n \quad dm = \frac{\sum_{i=0}^{n-1} abs(a(i) - m)}{n} \quad dt = \sqrt{\frac{\sum_{i=0}^{n-1} (a(i) - m)^2}{n}}$$

- 7.7.** Escribir una función que reciba como parámetro una matriz cuadrada de orden n , y calcule la traspuesta de la matriz almacenando el resultado en la propia matriz.

- 7.8.** 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.
- 7.9.** Escribir un programa que lea un número natural impar n menor o igual que 11, calcule y visualice el 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^2 , y cumple la condición de que la suma de los valores almacenados en cada fila y columna coincida.
- 7.10.** Escribir una función que encuentre el elemento mayor y menor de una matriz, así como las posiciones que ocupa y los visualice en pantalla.
- 7.11.** Escribir una función que reciba como parámetro una matriz cuadrada de orden n y decida si es simétrica. Una matriz cuadrada de orden n es simétrica si $a[i][j] == a[j][i]$ para todos los valores de los índices i, j .
- 7.12.** Codificar un programa C++ que permita visualizar el triángulo de Pascal:

		1						
			1	1				
				2	1			
		1			3	1		
			1			6	1	
1				4			4	
	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, un array de una sola dimensión.

- 7.13.** Escribir un programa que lea un texto de la entrada y cuente: el número de palabras leídas; número de líneas y vocales (letras a; letras e; letras i; letras o; letras u). El final de la entrada de datos viene dado por control + Z.
- 7.14.** Codificar un programa C++ que lea una frase, y decida si es palíndroma. Una frase se dice que es palíndroma si después de haber eliminado los blancos, se puede leer de igual forma en los dos sentidos.
- 7.15.** Escribir una función que reciba un número escrito en una cierta base b como cadena de caracteres y lo transforme en el mismo número escrito en otra cierta base b_1 como cadena de caracteres. Las bases b y b_1 son mayor es o iguales que 2 y menores o iguales que 16. Puede suponer que el número cabe en una variable numérica enter o largo, y usar la base 10 como base intermedia.
- 7.16.** Diseñar y codificar un programa que lea un texto y determine la frecuencia de aparición de cada letra mayúscula. El fin de lectura viene dado por (Control + Z).
- 7.17.** 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.

SOLUCIÓN DE LOS EJERCICIOS

- 7.1.** La salida del programa depende de la entrada de datos. Hay que tener en cuenta que se leen un total de 6 valores y se escriben los que ocupan las posiciones 6, 4, 2. Si la entrada de datos es por ejemplo: 3 7 4 -1 0 6, éstos se colocan en las posiciones del array número 1, 2, 3, 4, 5, 6 y, por 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.

- 7.2.** El programa rellena el vector *Primero* con los valores 3, 4, 5, ..., 11, 12 en las posiciones 0, 1, 2, ..., 9 del array. Si la entrada de datos es, por ejemplo, 8 3, el programa no tendrá ninguna salida ya que el segundo bucle es ascendente y el límite inferior $j = 8$ es mayor que el superior $k = 3$. Si la entrada de datos es, por ejemplo, 3, 8, el programa escribirá las posiciones del array 3, 4, 5, 6, 7, 8 que se han inicializado, previamente, con los valores 6, 7, 8, 9, 10, 11. Si la entrada de datos es tal que los índices que recorren la salida del vector no son menores que 9, entonces los resultados son imprevisibles.

- 7.3.** Los valores del vector *Primero* son rellenados por el primer bucle y son los siguientes:

Índice	0	1	2	3	4	5	6	7	8	9	10
valor	2	4	6	8	10	12	14	16	18	20	22

Los valores del vector *Segundo* son rellenados por el siguiente bucle, y son los siguientes:

Índice	0	1	2	3	4	5
valor	2	7	12	17	22	27

El tercer bucle del código escribe las siguientes posiciones del array.

Primero	Segundo
4	2
5	3
6	4

La salida del programa viene dada por los contenidos de esas posiciones que en este caso son:

10 12
12 17
14 22

Al ejecutar el programa se comprueba los resultados:

10 12
12 17
14 22

- 7.4.** Se declara la constante *max* como 10. Mediante un bucle *for* se rellena el vector previamente declarado como entero, almacenando en el índice *i* del vector el valor de $2 * i + 1$ ya que es el correspondiente número ímpar asociado. Observar que el bucle *for* se ha programado con el incremento de la variable de control del bucle dentro de la propia sentencia de asignación *vector[i++] = 2 * i + 1*. Mediante otro bucle *for*, se presentan los índices del array que ocupan posiciones pares 0, 2, 4, 6, 8.

Codificación

```
#include <cstdlib>
#include <iostream>
#define max 10
using namespace std;
```

```

int main(int argc, char *argv[])
{
    int i, vector[max];

    for(i = 0; i < max; )
        vector[i++] = 2 * i +1 ;
    for (i = 0; i < max; i += 2)
        cout << vector[i] << " ";
    cout << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

El resultado de ejecución del programa anterior es:

1 5 9 13 17

SOLUCIÓN DE LOS PROBLEMAS

- 7.1. La constante *max* se declara como 5. Se declara el vector de datos y en un primer bucle *for* se leen los valores de entrada almacenándolos en el vector *datos*. Mediante otro bucle *for* se presentan los valores leídos, al tiempo que se acumulan en el acumulador *suma* los valores. Finalmente se presentan los resultados.

La codificación de este problema se encuentra en la página Web del libro.

- 7.2. Teniendo en cuenta la definición; para que el vector sea simétrico, debe ocurrir que: el contenido de la posición 0 (el primero) coincida con el contenido de la posición $n - 1$ (el último); el contenido de la posición 1 (el segundo) coincide con el de la posición $n - 1 - 1$ (el penúltimo); etc. Es decir, el contenido de la posición *i* debe coincidir con el de la posición $n - i - 1$. Por tanto, para resolver el problema basta con comprobar solamente la mitad de los valores del vector (hasta $n / 2$). La función que se codifica recibe el vector *v* como parámetro, así como el número de elementos almacenados *n*. Mediante un bucle *for* se comprueba la condición de ser simétrico, y se itera mientras el contador sea menor o igual que $n / 2$ y la parte anteriormente recorrida sea simétrica.

Codificación

```

bool simetrica (float v[], int n)
{
    int i;

    bool sime = true;
    for (i = 0; i <= n / 2 && sime; i++)
        if (v[i] != v[n - 1 - i])
            sime = false;
    return sime;
}

```

- 7.3. Si el vector es mayoritario, necesariamente el elemento que se repite más de $n / 2$ veces debe aparecer en la primera mitad del array. De esta forma, para resolver el problema basta con comprobar el número de veces que se repiten los ele-

mentos que ocupan la primera mitad del vector, quedándose con el que más se repite, y si el número de veces que aparece el elemento más repetido es mayor que $n / 2$ la respuesta es afirmativa, y en otro caso es negativa. Mediante un bucle `for` controlado por el contador `i`, se recorren los elementos de la primera mitad del array `v`. Mediante otro bucle `for` interno, se cuenta el número de veces `nveces` que aparece repetido el elemento `v[i]` a su derecha. De todos estos valores obtenidos con el bucle externo hay que quedarse con el número de veces máximo `nvmaximo`, para al final retornar el valor `nvmaximo > n / 2` que será `true` cuando el vector sea mayoritario.

Codificación

```
bool mayoritario( float v[], int n)
{
    int i, j, nveces, nvmaximo = 0;

    for (i = 0 ; i <= n / 2 ; i++)
    {
        nveces = 1;
        for (j = i + 1 ; j < n; j++)
            if(v[i] == v[j])
                nveces++;
        if (nveces > nvmaximo)
            nvmaximo = nveces;
    }
    return (nvmaximo > n / 2);
}
```

- 7.4. Se programa la función producto escalar definido como $pe(a, a, n) = \sum_{i=0}^{n-1} a(i)*b(i)$ mediante un bucle `for`, controlado por el contador `i` que comienza en 0 y termina en $n - 1$. En cada iteración `i` del bucle se acumula en el acumulador `acu` el valor de `a[i] * b[i]`. Este acumulador `acu` se inicializa a 0 y se retorna su contenido al final de la función. Teniendo en cuenta que la norma de un vector satisface la igualdad:

$$\text{norma}(a, n) = \sqrt{\sum_{i=0}^{n-1} a(i)*a(i)} = \sqrt{pe(a, a, n)}$$
, para programar la función solicitada, basta con llamar a la función producto escalar con el vector `a` como parámetro en los dos argumentos, y extraer su raíz cuadrada.

El coseno del ángulo que forman dos vectores de n componentes satisface la siguiente igualdad:

$$\cos(a, b, n) = \frac{\sum_{i=0}^{n-1} a(i)*b(i)}{\sqrt{\sum_{i=0}^{n-1} a(i)*a(i) * \sum_{i=0}^{n-1} b(i)*b(i)}} = \frac{pe(a, b, n)}{\text{norma}(a, n)*\text{norma}(b, n)}$$

por lo que la programación de la función `coseno` se reduce a las llamadas correspondientes a la función producto escalar de los vectores `a` y `b` y las normas de los dos vectores citados vectores.

La codificación de este problema se encuentra en la página Web del libro.

- 7.5. La solución del problema se ha estructurado de la siguiente forma:

- Se declara la constante `max` como 100 para declarar un vector que almacene los 100 primeros números primos. El programa principal se encarga de declarar un array `a` para almacenar los `max` números primos, y mediante un bucle `for` llenar el vector de números primos, llamando a la función `primo` que decide si el número natural que se le pasa como pa-

rámetro es primo. Todos los números primos son impares, excepto el número 2 que además es el primer o, por lo que es introducido antes de comenzar el bucle en la posición 0 del vector *a*. Una vez que en *n* se tiene el primer candidato a número primo, en este caso el impar 3, los siguientes candidatos a números primos, se obtienen del anterior sumándole 2 unidades. De esta forma, si se inicializa la variable entera *n* a 3 para calcular el siguiente número primo, basta usar un bucle *while* que itere mientras, número *n* no sea primo el incrementando el contador *n* en 2 unidades *while* (*!primo(n)*) *n* += 2;. Cada vez que el bucle se para, es porque ha decidido que *n* es un número primo, por lo que hay que almacenarlo en el vector *a* y recalculiar el siguiente candidato a número primo *n* += 2.

- 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 $2 * 32 = 64$. 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* realiza la visualización de la lista de los 100 números primos en 10 líneas distintas. Esta función recibe en su parámetro vector la información, y escribe en un bucle *for* los *max* números que almacena.

La codificación de este problema se encuentra en la página Web del libro.

- 7.6. La función *main()* que se encarga de llamar a las distintas funciones que resuelven el problema. En el programa principal *main()* se declara el vector de *max* datos, siendo *max* una constante declarada en una sentencia *define*.

- La función *aleatorio* genera el número de elementos *n* que tiene el vector aleatoriamente, e introduce *n* números aleatorios en el vector que recibe *a* como parámetro.
- Es la función *escribe* la encargada de visualizar el vector que recibe como parámetro.
- El cálculo de la media lo realiza con una función *m*, sumando en un bucle voraz *for* los *n* elementos del vector *a* que recibe como parámetro, y dividiendo el resultado de la suma entre el número de elementos *n* que tiene.
- La desviación media se calcula mediante la función *dm*, que acumula en el acumulador *aux* los valores de *fabs(a[i] - media)*, siendo *media* el valor de la media de los elementos del vector *a*. El valor obtenido en el acumulador *aux* se divide entre *n*.
- La desviación típica se programa en la función *dt*. La programación de la función es muy similar a la de la función *dm*, con la diferencia que ahora se acumula el valor de *(a[i] - media) * (a[i] - media)*, y el resultado es la raíz cuadrada del acumulador dividido entre *n*.

La codificación de este problema se encuentra en la página Web del libro.

- 7.7. 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]* para todo los posibles índices *i, j*. Por tanto, para calcular la traspuesta de la matriz *a* sobre sí misma, basta con intercambiar los valores de *a[i][j]* con *a[j][i]*, para *i = 0, 1, ..., n - 1*, y *j = 0, 1, 2, ... i - 1*. La función *traspuesta* recibe como parámetro la matriz cuadrada *a*, así como la dimensión *n*.

Codificación

```
#define max 11
void traspuesta(float a[][max], int n)
{
    float aux;

    for (int i = 0; i < n; i++)
        for (int j = 0; j < i; j++)
    {
        aux = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = aux;
    }
}
```

```

    a[j][i] = aux;
}
}

```

- 7.8.** Para resolver el problema se supone que la matriz no tiene elementos repetidos. La función recibe como parámetro la matriz de reales $a[\max]$ (\max es una constante previamente declarada), y la dimensión n . La codificación se ha planteado de la siguiente forma: un primer bucle recorre las distintas columnas de la matriz. Dentro de ese bucle se calcula el elemento mayor de la columna y la posición de la fila en la que se encuentra. Posteriormente, se comprueba si la fila en la que se ha encontrado el elemento mayor de la columna cumple la condición de que es el elemento menor de la fila. En caso positivo se ha encontrado un punto de silla y se visualiza el resultado.

La codificación de este problema se encuentra en la página Web del libro.

- 7.9.** En un array bidimensional se almacenará el cuadrado mágico. La función `main()` lee y garantiza mediante un bucle `do while` que el número n que se lee es impar y está en el rango 1 a 11. El problema se resuelve usando las funciones:

- *siguiente* 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 . En la codificación se programa incrementando i en una unidad, y asignando a i el resto de la división entera de i entre n .
- *anterior* 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 .
- *comprueba* 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.
- *cuadrado* calcula el cuadrado mágico mediante el siguiente algoritmo:

Se pone toda la matriz a ceros, para indicar que las casillas están libres.

Se inicializa la fila $i = 0$ la columna $j = n / 2$.

Mediante un bucle que comienza por el valor 1, se van colocando los valores en orden creciente hasta el $n * n$ de la siguiente forma:

Si la posición fila i columna j de la matriz está libre (tiene el valor de 0) se almacena, y se recalcula la fila con la función *anterior* y la columna con la función *siguiente*.

Si la posición fila i columna j de la matriz está ocupada (tiene valor distinto de 0) se recalcula i aplicándole dos veces la función *siguiente* y se recalcula j aplicándole una vez la función *anterior*. Se almacena el valor en la posición fila i columna j (siempre está libre), para, posteriormente, recalcular la fila con la función *anterior* y la columna con la función *siguiente*.

La codificación de este problema se encuentra en la página Web del libro.

- 7.10.** La solución se presenta mediante una función que recibe la matriz a como parámetro, la dimensión n , y calcula el mayor y menor mediante la técnica voraz. Se inicializa *menor* y *mayor* al elemento $a[0][0]$, así como los respectivos índices. Dos bucles anidados recorren todos los índices de la matriz cuadrada y para cada elemento de la matriz hacen comparaciones con el anterior *menor* y el anterior *mayor*, decidiendo quedarse con un nuevo *menor* o un nuevo *mayor* así como los índices que ocupan, si es necesario. Al final de la función se presentan los resultados.

La codificación de este problema se encuentra en la página Web del libro.

- 7.11.** Una matriz cuadrada es simétrica si $a[i][j] == a[j][i]$ para todo i y para todo j . Por tanto, basta que un par de índices i, j no satisfaga la condición, para que la matriz sea no simétrica. Por otro lado, para comprobar que la matriz es

simétrica basta con hacerlo para los valores $i = 0, 1, \dots, n-1$, y $j = 0, \dots, i - 1$, ya que se comprueba la parte inferior de la diagonal principal con la parte superior de la diagonal principal y, por tanto, se comprueba toda la matriz. La programación se ha hecho con los dos bucles anidados indicados anteriormente, añadiendo como condición de salida de ambos bucles, que no se haya detectado alguna discrepancia entre $a[i][j]$ con $a[j][i]$, en cuyo caso la variable interruptor `sime` toma el valor de `false`, ya que se sabe que la matriz cuadrada recibida como parámetro no es simétrica.

Codificación

```
bool simetrica (float a[][][max], int n)
{
    bool sime = true;

    for (int i = 0; i < n && sime; i++)
        for (int j = 0; j < i && sime; j++)
            if (a[i][j] != a[j][i])
                sime = false;
    return sime;
}
```

- 7.12. El triángulo de Pascal puede representarse de la siguiente forma para facilitar la codificación del algoritmos, si bien la visualización es ligeramente distinta:

```
1
1      1
1      2      1
1      3      3      1
1      4      6      4      1
1      5      10     10     5      1
```

Se piden dos soluciones al problema. La primera con un array bidimensional matriz, y la segunda con un array unidimensional vector. Tanto la matriz como el vector se declaran de una longitud máxima de `max = 11`. La solución se estructura de la siguiente forma:

- La función principal `main()`, declara la matriz y el vector, lee el orden en un bucle do `while`, validando la entrada y realiza las llamadas a las funciones.
- Una función `triángulo en dos dimensiones` `Tdosdim` tiene como parámetros el array bidimensional `matriz` y una variable entera `orden` y calcula el triángulo de Pascal del orden que se pasa como parámetro. El triángulo de Pascal se coloca en la matriz por debajo de la diagonal superior. La codificación se realiza con dos bucles anidados. El bucle exterior es controlado por el contador `i`, itera desde 0 hasta `n - 1`, y el bucle interior que es controlado por el contador `j` que itera desde 0 hasta `i`. Se observa que `matriz[i][j]` toma el valor 1 si `j == 0`, o bien si `i == j`. En otro caso `matriz[i][j] = matriz[i - 1][j - 1] + matriz[i - 1][j]`.
- La función `escribe triángulo de dos dimensiones` `ETdosdim` tiene como parámetros el array bidimensional `matriz` y una variable entera `orden` se encarga de visualizar el triángulo de Pascal del orden que recibe como parámetro.
- La función `Triángulo en una dimensión` `Tunadim` tiene como parámetros el array unidimensional `vector` y una variable entera `orden`, calcula y escribe el triángulo de Pascal del orden que recibe como parámetro. En esta función el bucle interior `j` se hace descendente para no “pisar” los valores anteriormente calculados. En este caso si `i == j || j == 0` entonces `vector[j] = 1`. En otro caso `vector[j] = vector[j] + vector[j - 1];`

La codificación de este problema se encuentra en la página Web del libro.

En el resultado de ejecución del programa anterior , pueden visualizarse los dos triángulos de Pascal de orden 7 realizados con una matriz y con un vector.

```
introduzca valor de n <=11
?
Triangulo de Pascal de orden 7
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
Triangulo de Pascal de orden 7
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

- 7.13.** Se declara un vector entero contador de cinco componentes que sirve para contar las vocales correspondientes, que es inicializado a 0 en el bucle controlado por el contador *i*. El contador *npalabras*, cuenta el número de palabras de la entrada, y otro contador *nlineas* cuenta el número de líneas. Un bucle itera mientras no sea fin de fichero (control + Z), y otro bucle itera mientras se encuentre en una palabra. Cada vez que termina este bucle se cuenta una palabra. El bucle interior lee un carácter en la variable *c* e incrementa el contador de la vocal correspondiente en caso de que lo sea. Cada vez que *c* toma el valor '\n', se cuenta una nueva línea.

La codificación de este problema se encuentra en la página Web del libro.

- 7.14.** 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 *Cad*. Se copia la frase en un array de caracteres *Cad1*, eliminando los blancos, y contando los caracteres que se ponen en el contador *c1*. Posteriormente, mediante un bucle controlado por el contador *c* que comienza en 0 y termina en la mitad de la longitud de los caracteres de la frase, una vez eliminados los blancos, *c1 / 2*, se comprueba si la letra que ocupa la posición *c_ésima* coincide con la correspondiente que ocupa la posición *c1-1-c_ésima*, *Cadaux[c] == Cadaux[c1 - 1 - c]*. La variable booleana *Espalindroma* se pone a true, y sólo se cambia a false si la frase es no palíndroma.

La codificación de este problema se encuentra en la página Web del libro.

- 7.15.** Para resolver el problema se usa como base auxiliar intermedia la base 10, y se supone que el número que se recibe como parámetro cabe en un entero largo. La función pedida es *cambioboble*. Para pasar de un número almacenado en una cadena de caracteres entrada y escrito en la base *base* al mismo número almacenado en otra cadena de caracteres salida y escrito en otra base *base1*, basta llamar a las funciones *numero = valorBase10* (entrada, *base*), que transforma la entrada a un dato numérico *numero* escrito en base 10 y, posteriormente, llamar a la función *cambioBase(numero, base1, salida)*, que transforma el número en base 10 a la cadena de caracteres *salida* escrita en *base1*.

La función *cambiobase* recibe el dato en base 10 en la variable entera larga *numero* y transforma el dato a una cadena de caracteres *salida* que representa el dato escrito como cadena de caracteres en la base *base*. 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', etcétera (sólo pueden usarse bases entre 2 y 16). Se define un vector de enteros auxiliar que almacena en cada una de las posiciones (como resultado intermedio), el dígito entero al que corresponde cuando se pasa el dato a la base que se recibe como parámetro. Para hacer esto, se usa el esquema siguiente: mientras la variable *numero* sea distinta de cero hacer en la posición *c* del vector *auxiliar* almacenar el resto de la división entera del número *numero* entre la *base*, para posteriormente almacenar en la variable *numero* el cociente de la división entera del número *numero* entre la *base*. Para terminar se

pasa el número a cadena de caracteres salida usando el array bases, como indexación, `salida[n - c] = bases[auxiliar[c]]`.

La función `valorBase10` recibe como parámetro el dato almacenado en la cadena de caracteres cadena y la base en el que está escrito, y lo trasforma a su correspondiente valor numérico en base 10 devolviéndolo en la propia función, que es declarada como de tipo entero largo. Para realizar la operación se usa, 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.
```

Mediante un bucle `for` se recorren todas las posiciones de la cadena, hasta que no queden datos `cadena[c] == '\0'`. El carácter almacenado en la posición `c` de cadena se transforma en un dato numérico con la expresión `valor = cadena[c] - '0'`, si es un dígito, y mediante la expresión `valor = cadena[c] - 'A' + 10`, en el caso de que sea un carácter de entre 'A', 'B', 'C', 'D', 'E', 'F'.

La codificación de este problema se encuentra en la página Web del libro.

- 7.16.** Se declara la constante máxima, que sirve para dimensionar el array contador, que es donde se cuentan las letras mayúsculas que se leen del texto. En un bucle `for`, se inicializa el array contador a 0. En un bucle `while`, mientras no se dé fin de fichero (control + Z), se lee un carácter, y si es una letra mayúscula se incrementa el contador de índice `c - 'A'`, en una unidad (`contador[c - 'A']++`). Una vez terminada la lectura, se escriben todos aquellos índices de contador es no nulos, que son aquellos que hacen referencia a letras mayúsculas que han aparecido en el texto.

La codificación de este problema se encuentra en la página Web del libro.

- 7.17.** Se declaran dos cadenas de caracteres. La primera Cad sirve para leer la frase de la entrada. La segunda Cadaux se usa para almacenar cada una de las palabras que se extraen de Cad y poder escribir el número de letras que lo forman. Se consideran separadores de palabras los caracteres (coma, blanco y punto, ',', ' ' y '.'). La codificación se realiza mediante un bucle `while` que itera hasta que se termina la cadena de caracteres Cad. Este bucle lo controla el contador `c` que se inicializa a 0, e indica en cada momento la posición del carácter almacenado en Cad que se está tratando. Dentro de este bucle hay otro bucle `while` que es el encargado de extraer la palabra y contar el número de caracteres que tiene. Este bucle está controlado por el fin de cadena, de Cad y el no haber encontrado un separador en Cad. El contador `c1`, se inicializa a 0, y cada vez que se debe almacenar un carácter en la cadena auxiliar Cadaux, se incrementa en una unidad. Una vez terminado este bucle, se convierte Cadaux en una cadena añadiéndole el carácter de fin de cadena '\0', y se escribe la palabra y su número de letras dado por el contador `c1`.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 7.1.** ¿Cuál es el error del fragmento del siguiente fragmento de programa?

```
int j, x[8];
for ( j = 1; j<= 9; )
    x[j] = 2 * j;
```

- 7.2.** Determinar la salida del cada segmento de programa:

```
int i ,j ,k;
int Primero[6];
for( j = 0; j < 7; )
    cin >> Primero[j++];
```

```
i = 0; j = 1;
```

```
while (( j < 6) && (Primero[j - 1] < Primero[j]))
{
    i++; j++;
}
for( k = -1; k < j + 2; k+=2)
    cout << Primero[ ++ k] << endl;
```

si la entrada de datos es la siguiente: 20 60 70 10 0
40 30 90

- 7.3.** ¿Cuál es el error del siguiente fragmento de programa?

```
float X[char]
char j =1;
X[j] = 14.325;
```

- 7.4.** 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)
    {
        cout << i << " " << j << " " <<
            Tercero[i][j]);
        j += 3;
    }
}
```

- 7.5.** ¿Cuál es la salida del siguiente segmento de programa para la entrada de datos dada al final?

```
int i, Primero[10];
for (i = 0; i < 6 ; i++)
    cin >> Primero [i];
for (i = 2; i >= 0; i--)
    cout << Primero [2 * i] << " ";
```

3 7 4 -1 0 6

- 7.6.** ¿Cuál es la salida del siguiente segmento de programa para la entrada de datos dada al final?

```
int i, j, k , Primero[10];
for (i = 0; i < 10 ; i++)
    Primero[i] = i + 3;
cin >> j >> k;
for (i = j; i <= k ; i++)
    cout << Primero[i] << endl;
```

7 2 3 9

- 7.7.** Escribir un programa que lea el array:

4	7	1	3	5
2	0	6	9	7
3	1	2	6	4

y lo visualice como:

4	2	3
7	0	1
1	6	2
3	9	6
5	7	4

- 7.8.** Escribir una función que intercambie la fila *i*-ésima por la *j*-ésima de un array mxn.

PROBLEMAS PROPUESTOS

- 7.1.** Escribir un programa que convierta un número romano (en forma de cadena de caracteres) en número arábigo.
Reglas de conversión:

M	1000
D	500
C	100
L	50
X	10
V	5
I	1

- 7.2.** Escribir una función a la que se le proporcione una fecha (día, mes, año), así como un número de días a añadir a esta fecha. La función debe calcular la nueva fecha y visualizarla.

- 7.3.** Escribir una función que acepte como parámetro un vector que puede contener elementos duplicados. La función debe sustituir cada valor repetido por -5 y devolver al punto donde fue llamado el vector modificado y el número de entradas modificadas.

- 7.4.** Escribir un programa que lea una colección de cadenas de caracteres de longitud arbitraria. Por cada cadena leída, su programa hará lo siguiente:

- a) Visualizar la longitud de la cadena.

- b) Contar el número de ocurrencia de palabras de cuatro letras.
- c) Sustituir cada palabra de cuatro letras por una cadena de cuatro asteriscos e imprimir la nueva cadena.
- 7.5. 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.
- 7.6. Escribir un programa que desplace una palabra leída del teclado desde la izquierda hasta la derecha de la pantalla.
- 7.7. Escribir un programa que lea una serie de cadenas, a continuación, determine si la cadena es un identificador válido C++. *Sugerencias:* utilizar los siguientes subprogramas: 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).
- 7.8. Escribir una función con versión que reciba como parámetro una cadena representando una fecha en formato 'dd/mm/aa', como 17/11/91 y la devuelva en formato de texto: 17 noviembre 1991.
- 7.9. Escribir un programa que permita escribir en sentido inverso una cadena de caracteres.
- 7.10. Buscar una palabra en una cadena y calcular su frecuencia de aparición.
- 7.11. 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.
- 7.12. 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.

CAPÍTULO 8

Registros (estructuras y uniones)

Introducción

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. Por otra parte, la *unión* es otro tipo de dato no tan importante como las estructuras array , pero sí necesaria en algunos casos. Un tipo de dato enumerado (enumeración) es una colección de miembros con nombre que tienen valores enteros equivalentes. Un *tipo definido por el usuario*, *typedef*, es de hecho, no un nuevo tipo de dato sino, simplemente un sinónimo de un tipo existente.

8.1. Estructuras

Una *estructura* es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar . Una *estructura* es una colección de uno o más tipos de elementos denominados *miembros*, cada uno de los cuales puede ser un tipo de dato diferente. Puede contener cualquier número de miembros, cada uno de los cuales tiene un nombre único, denominado *nombre* del miembro. Los miembros de las estructuras pueden ser arrays. Del mismo modo que sucede en otras situaciones, en C++ existen dos conceptos similares a considerar, *declaración* y *definición*.

Declaración de una estructura. Una *declaración* especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria. 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>
};
```

Definición de variables de estructura. Una *definición* de variable para una estructura dada, crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado en la definición de la estructura.

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.
2. Listando el tipo de la estructura creada seguida por las variables correspondientes en cualquier lugar del programa antes de utilizarlas.

A una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura.

Uso de estructuras en asignaciones. Como una estructura es un tipo de dato similar a un `int` o un `char`, se pueden asignar variables de una estructura a otra entre sí.

EJEMPLO 8.1. Declaración de una estructura, definición de variables de tipo estructura y asignación de variables. Se declara una estructura `libro` cuyos miembros son `titulo`, `autor`, `editorial`, `anyo`, `precio`, `fecha_compra`, y variables de tipo `libro` `l1, l2, l3, l4, l5, l6`.

```
struct libro                                // nombre de la estructura
{
    char titulo[30];
    char autor[25];
    char editorial[30];
    int anyo;
    float precio;
    char fecha_compra[8];
} l1, l2, l3;                                // l1, l2, l3 son variables de tipo libro

libro l4, l5, l6;                            // l4, l5, l6 son variables de tipo libro

int main(int argc, char *argv[])
{
    ...
    l1= l2 =l3;                                // asignación de estructuras
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Inicialización de una declaración de estructuras. Se puede inicializar una estructura de dos formas:

1. Como parte de la sección de código de su programa.
2. 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
};
```

EJEMPLO 8.2. Inicialización de estructuras. Se inicializa la variable `l1` de tipo `libro`, y se hace otra declaración con una inicialización del tipo de dato complejo:

```
struct complejo
{
    float real;
    float imaginaria;
} c1 = { 23.2, 10,1}
```

```
//dada la declaración de libro realizada en el ejemplo 8.1
//la siguiente es una declaración de variable e inicialización

libro l7 = { "Programación en C",
    "Joyanes, Luis",
    "McGrawHill",
    2002,
    20,
    "02/6/05"
};
```

El tamaño de una estructura. El operador `sizeof` se puede aplicar para determinar el tamaño que ocupa en memoria una estructura. El resultado se obtiene determinando el número de bytes que ocupa la estructura:

EJEMPLO 8.3. Visualiza el tamaño de dos estructuras.

```
#include <cstdlib>
#include <iostream>
using namespace std;
struct complejo
{
    float real;
    float imaginaria;
};

struct Persona
{
    char nombre[30];
    int edad;
    float altura;
    float peso;
};

int main(int argc, char *argv[])
{
    cout << "sizeof(Persona): " << sizeof(Persona) << endl;
    cout << "sizeof(complejo): " << sizeof(complejo) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

El resultado de ejecución:

```
sizeof(Persona): 44
sizeof(complejo): 8
Presione una tecla para continuar . . .
```

Determinación del número de bytes que ocupa la estructura:

Complejo	Persona	Miembros	Tamaño (bytes) Persona	Tamaño (bytes) complejo
real, imaginaria	nombre[30]	char(1)	30	
	edad	int(2)	2	
	altura	float(4)	4	8
	peso	float(4)	4	
	Total		42	

Acceso a estructuras. El acceso a los miembros de una estructura se realiza cuando : se accede a una estructura; se almacena información en la estructura; 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, utilice 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 new.

EJEMPLO 8.4. Asignación de valores a los miembros de variables estructura. Se realizan asignaciones a variables de tipo Persona declarada en el Ejemplo 8.3.

```
Persona p, *p1;
                                // p es una variable estructura.
strcpy(p.nombre , " Angel Hernández");           // Asignación de cadenas
p.edad = 20;
p.altura = 1.75;
p.peso = 78;
                                // p1 es una variable puntero a estructura.
strcpy(p1->nombre , " Fernando Santos");         // Asignación de cadenas
p1->edad = 21;
p1->altura = 1.70;
p1->peso = 70;
```

Si se desea introducir la información en la estructura basta con acceder a los miembros de la estructura con el operador punto o flecha(puntero). Se puede introducir la información desde el teclado o desde un archivo, o asignar valores calculados (cin). Se recupera información de una estructura utilizando el operador de asignación o una sentencia de salida (cout). Igual que antes, se puede emplear el operador punto o el operador flecha (puntero). El formato general toma uno de estos formatos:

```
<nombre variable> = <nombre variable estructura>.<nombre miembro>;
```

o bien

```
<nombre variable> = <puntero de estructura> -> <nombre miembro>;
```

EJEMPLO 8.5. Lectura, recuperación de información de estructuras del tipo de dato Persona definida en el Ejemplo 8.3.

```
Persona p, *p1;
char nombre1[30];
int edad1;
float altura1;
float peso1;
                                // variable p
cout << " introduzca nombre : ";
cin.getline( p.nombre, 30);      // lee nombre de variable estructura p
```

```

cout << " introduzca edad altura peso :";
cin >> p.edad >> p.altura >> p.peso;           // lee resto de información
cout << p.nombre << " " << p.edad << " " << p.altura << " " << p.peso << endl;
                                                //variable *p1
cin.getline(p1->nombre, 2); // leer retorno de carro no leído en cin
cout << " introduzca nombre : ";
cin.getline(p1->nombre, 30);
                                // lee nombre de variable puntero estructura p1
cin >> p1->edad >> p1->altura >> p1->peso;
cout << p1->nombre << " " << p1->edad << p1->altura << p1->peso << endl;
strcpy(nombre1,p.nombre);
strcpy(p.nombre, p1->nombre);
edad1 = p.edad;
altural = p1-> altura

```

Estructuras anidadas. 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.

EJEMPLO 8.6. *Estructuras anidadas. Se declara un tipo de dato estructura para representar a un alumno. Los miembros que tiene son: nombre, curso, edad, dirección y notas de las 6 asignaturas. Se declara otro tipo estructura para representar a un profesor. Los miembros que debe tener son: nombre, cargo, nombre de las 4 asignaturas que puede impartir y dirección. Se declara la estructura de datos que contiene los atributos nombre y dirección. Las estructuras alumno y profesor tienen anidadas la estructura datos.*

La representación gráfica de las estructuras anidadas es:

alumno					profesor							
	Curso Edad					Cargo						
	datos:		nombre dirección			datos:		nombre dirección				
	Notas	0	1	2	3	4	5	asignaturas	0	1	2	3

```

struct datos
{
    char nombre[40],direccion[40];
};

struct alumno
{
    datos dat;
    int curso, edad, notas[6];
};

struct profesor
{
    int cargo;
    datos dat;
    char asignaturas[4][10];           // nombre de asignaturas que imparte
};

```

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 almacenados en 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.

EJEMPLO 8.7. *Array de estructuras. Hay un total de 100 clientes. Los clientes tienen un nombre, el número de unidades solicitadas de un determinado producto, el precio de cada unidad y el estado en que se encuentra: moroso, atrasado, pagado. El programa lee la información de los 100 clientes.*

Se define la estructura `cliente` siguiendo las especificaciones y con ella se declara un array que almacenará la información de cada cliente. El *array de estructuras* funciona como una base de datos relacional, en la que cada miembro de la estructura funciona como una columna de la base de datos y cada estructura corresponde a una línea o registro de dicha base.

```
#include <cstdlib>
#include <iostream>
using namespace std;

struct cliente
{
    char nombre[20];
    int numUnidades;
    float precio;
    char estado;
};

int main(int argc, char *argv[])
{
    cliente listado [100],buffer[4]; //buffer para limpiar entrada de cin

    for (int i = 0; i < 100; i++)
    {
        cout << "Introduzca nombre del cliente: ";
        cin.getline (listado[i].nombre, 20);
        cout << "\n Introduzca el número de unidades solicitadas: ";
        cin >> listado[i].numUnidades;
        cout << "\n Introduzca el precio de cada unidad:";
        cin >> listado[i].precio;
        cout << "\n Introduzca el estado del cliente (m/a/p)";
        cin >> listado[i].estado;
        cin.getline(buffer,2); // limpiar buffer de entrada por número
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

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 8.8. *Parámetros tipo estructura por referencia a funciones. Se lee la información de las estructuras declaradas en el Ejemplo 8.6 de un alumno y de un profesor, mediante funciones.*

La función `leer_datos` lee los datos de la estructura `datos`. La función `leer_alumno`, lee la información de un alumnos usando la función `leer_datos`. La lectura de la información de un profesor se realiza con `leer_profesor`.

```

void leer_datos (datos &dat)           // estructura por referencia
{
    cout << " nombre : ";
    cin.getline(dat.nombre, 40);
    cout << " direccion: ";
    cin.getline (dat.direccion, 40);
}

void leer_alumno(alumno &a)           // estructura por referencia
{
    cout << " datos de alumno: " << endl;
    leer_datos(a.dat);
    cout << " curso: ";
    cin >> a.curso;
    cout << " edad: ";
    cin >> a.edad;
    for (int i = 0; i < 6 ;i++)
    {
        cout << " nota " << i;
        cin >> a.notas[i];
    }
}

void leer_profesor(profesor &p)         // estructura por referencia
{
    cout << " datos de profesor: " << endl;;
    leer_datos(p.dat);
    cout << " cargo: ";
    cin >> p.cargo;

    for (int i = 0; i < 4 ;i++)
    {
        cout << " asignatura " << i;
        cin.getline(p.asignaturas[i],10);
    }
}

```

EJEMPLO 8.9. Parámetros tipo estructura por valor a funciones. Se escribe la información de las estructuras declaradas en el Ejemplo 8.6 de un alumno y de un profesor, mediante sendas funciones.

La función escribir_alumno, escribe la información de un alumno usando la función escribir_datos. Esta última función escribe los datos de la estructura datos. La escritura de la información de un profesor se realiza con escribir_profesor.

```

void escribir_datos (datos dat)
{
    cout << " nombre : " << dat.nombre << endl;
    cout << " direccion: " << dat.direccion << endl;
}

void lescribir_alumno(alumno a)
{
    cout << " datos de alumno: " << endl;;
    escribir_datos(a.dat);
    cout << " curso: " << a.curso << endl;
    cout << " edad: " << a.edad << endl;
}

```

```

for (int i = 0; i < 6 ;i++)
{
    cout << " nota " << i << " " << a.notas[i];
}
cout << endl;
}

void escribir_profesor(profesor p)
{
    cout << " datos de profesor: " << endl;
    escribir_datos(p.dat);
    cout << " cargo: " << p.cargo << endl;
    for (int i = 0; i < 4 ;i++)
    {
        cout << " asignatura " << i << " " << p.asignaturas[i];
    }
    cout << endl;
};

```

8.2 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, sus efectos son también diferentes. Una estructura (`struct`) permite almacenar variables relacionadas juntas y almacenarlas 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: es analizado el tamaño de cada variable de la unión, el mayor tamaño de variable será el tamaño de la unión. 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 comparte memoria con los otros miembros de la unión.

La sintaxis de una unión es la siguiente:

```

union nombre
{
    tipo1 miembro1;
    tipo2 miembro2;
    ...
};

```

Una razón para utilizar una unión es ahorrar memoria. En muchos programas se debe 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 8.10. *Definiciones de uniones, y acceso con variables diferentes.*

```

#include <cstdlib>
#include <iostream>
using namespace std;

union ejemplo
{
    int valor;
    char car[2];
} Ejeunion;

```

```

int main(int argc, char *argv[])
{
    cout << " Introduzca un numero entero: ";
    cin >> Ejeunion.valor;
    cout << " La mitad mas significativa es : ";
    cout << (int)Ejeunion.car[1]<< endl;
    cout << " La mitad menos significativa es : ">;
    cout << (int)Ejeunion.car[0] << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

```

Introduzca un numero entero: 2345
La mitad mas significativa es : 9
La mitad menos significativa es : 41
Presione una tecla para continuar . . .

```

EJEMPLO 8.11. *Array de estructuras que contiene una unión de estructuras. Un universitario puede ser un alumno o un profesor. Para representarlo se usa una unión universitario que puede ser la estructura alumno o la estructura profesor de los Ejemplos 8.6, 8.8 o bien 8.9. Se leen y escriben los datos de un grupo completo de universitarios.*

En la implementación, se declara una estructura `datos_grupo` que tiene un miembro interruptor `sw` para indicar si se almacena un alumno o un profesor, en el miembro `persona` de la unión `universitario`. Se usan las declaraciones y funciones de los Ejemplos 8.6, 8.8, 8.9. La función `leer_universitario`, lee un `universitario`, retornando el resultado en el parámetro por referencia `u`, recibiendo la información, de si es un profesor o un alumno en la variable lógica `sw`. La función `escribir_universitario`, escribe el `universitario`, que recibe como dato en el parámetro por valor `u`. La información de ser un profesor o un alumno está dada por la variable lógica `sw`. Las funciones `leer_datos_grupo` y `escribir_datos_grupo`, leen y escriben respectivamente los datos de un grupo completo de universitarios en una array que se pasa como parámetro.

```

#include <cstdlib>
#include <iostream>
#define max 100           // máximo número de universitarios de un grupo.
using namespace std;
...
...
union universitario
{
    ...
    alumno a;
    profesor p;
};

struct datos_universitario
{
    bool sw;           // si es true la persona universitaria es un alumno.
    universitario persona;
};

void leer_universitario(universitario &u, bool sw)
{
    if (sw)
        leer_alumno(u.a);
    else
        leer_profesor(u.p);
}

```

```

void escribir_universitario(universitario &u, bool sw)
{
    if (sw)
        escribir_alumno(u.a);
    else
        escribir_profesor(u.p);

}

void leer_datos_grupo (datos_universitario grupo[], int n)
{
    char ch;

    cout<< " lectura datos grupo \n";
    for (int i = 0; i < n; i++)
    {
        do
        {
            cout << " elija a alumno p profesor del universitario " << i << endl;
            cin >> ch;
            ch = toupper( ch); // mayúscula
        } while ( ch != 'A' || ch != 'P');
        grupo[i].sw = ch == 'A';
        leer_universitario(grupo[i].persona, grupo[i].sw);
    }
}

void escribir_datos_grupo (datos_universitario grupo[], int n)
{
    cout<< " visualización datos grupo \n";

    for (int i = 0; i < n; i++)
    {
        cout << " universitario " << i << endl;
        escribir_universitario(grupo[i].persona, grupo[i].sw);
    }
}

int main(int argc, char *argv[])
{
    datos_universitario grupo[max];
    leer_datos_grupo(grupo, max);
    escribir_datos_grupo(grupo, max);
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

8.3. Enumeraciones

Un tipo enum 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, ... Formatos de declaración.

```

enum
{enumerador1, enumerador2, ...enumeradorn};

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:

```
enum nombre
{   enumerador1 = expresión_constante1,
    enumerador2 = expresión_constante2,
    ...
    enumeradorn = expresión_constanten
};
```

EJEMPLO 8.12. *Usos típicos de enum: Interruptor, colores, semana y menu.*

```
enum Interruptor
{ // valor por defecto de On 0, y de Off 1
    On;                                // On, activado
    Off;                               // Off desactivado
};

enum colores
{ // cambio de valores por defecto
    Rojo = 7, Anaranjado = 6, Verde =4, Azul = 4 Anyl =3, Violeta = 2;
};

enum semana
{
    lunes, martes, miércoles, jueves, viernes, sabado domingo
}

// uso de enum para la creación de un menu

#include <cstdlib>
#include <iostream>
using namespace std;

enum tipo_operacion {anadir, borrar, modificar, visualizar};
.....
int main(int argc, char *argv[])
{
    tipo_operación op;
    ...
    switch (op)
    {
        case anadir:                anade_estructura(argumentos);
                                    break;
        case borrar:                 borra_estructura(argumentos);
                                    break;
        case modificar:              modifica_estructura(argumentos);
                                    break;
        case visulizar:              visualiza_estructura(argumentos);
                                    break;
        default:                     escribe_error(argumentos);
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

El tamaño en bytes de una estructura, de una unión o de un tipo enumerado se puede determinar con el operador `sizeof`.

8.4. Sinónimo de un tipo de dato `typedef`

Un tipo de dato `typedef` permite a un programador crear un sinónimo de un tipo de dato definido por el usuario o integral ya existente. No es nuevo tipo de dato sino un sinónimo de un tipo existente. La ventaja de `typedef` es que permite dar nombres de tipos de datos más acordes con lo que representan en una determinada aplicación, y ayuda a la transportabilidad.

EJEMPLO 8.13. *Usos típicos de `typedef`.*

```
typedef double Longitud;      //el tipo double se denomina también Longitud
typedef char caracter;        //tipo char se denomina además caracter

typedef char* String;         // String almacena puntero a carácter, cadena

typedef struct datos1
{
    char nombre[20];
    float salario;
} nuevosdatos;
```

La estructura tiene dos nombres sinónimos. Pueden declararse variables como:

```
datos1 dat;
nuevosdatos dat1;

struct registro_operacion
{
    long numero_cuenta;
    float cantidad;
    TipOperacion operacion;
    Fecha f;
    Tiempo t;
};

typedef struct registro_operacion RegistrOperacion;
//Se tienen dos nombres sinónimos.
```

8.5. Campos de bit

El lenguaje C++ permite realizar operaciones con los bits de una palabra. Un campo de bits es un conjunto de bits adyacentes dentro de una palabra entera. Con los *campos de bit*, C++ permite acceder a un número de bits de una palabra entera. La sintaxis para declarar campos de bits se basa en la declaración de estructuras. El formato general es:

```
struct identificador_campo {
    tipo nombre1:longitud1;
    tipo nombre2:longitud2;
    tipo nombre3:longitud3;
    .
    .
    .
    tipo nombrnen:longitudn;
};
```

donde 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.

Al declarar 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, utilizar para ello una union con variable entera y campos de bits).

EJEMPLO 8.14. Definición de estructuras con campos de bit.

```
struct ejemplo
{
    char nombre[50], dirección[30], telefono[10];
    int edad:4;                                // edad menor o igual que 127 años
    int sexo:1;                                 // 1= hombre 0= mujer
    int codigo:3;                               // código de departamento < 8
    int contrato:2;                            // hay cuatro tipos de contrato
};
```

EJEMPLO 8.15. Campo de bits y unión para visualizar la decodificación en bits de cualquier carácter leído por teclado.

Se declara un tipo estructura campo de bits, con tantos campos como bits tiene un byte, que a su vez es el almacenamiento de un carácter. Se decodifica declarando una union entre una variable carácter y una variable campo de bits del tipo indicado.

```
#include <cstdlib>
#include <iostream>
using namespace std;
struct byte {
    unsigned int a: 1;
    unsigned int b: 1;
    unsigned int c: 1;
    unsigned int d: 1;
    unsigned int e: 1;
    unsigned int f: 1;
    unsigned int g: 1;
    unsigned int h: 1;
};

union charbits
{
    char ch;
    byte bits;
} caracter;

void decodifica(byte b)
{
    // Los campos de bits se alinean de derecha a izquierda, por esa
    // razón se escriben los campos en orden inverso
    cout << b.h << b.g << b.f << b.e << b.d << b.c << b.b << b.a << endl;
}
```

```

int main(int argc, char *argv[])
{
    cout << "Teclea caracteres. Para salir carácter X \n";
    cout << "carácter ascii binario\n";
    do
    {
        cin >> carácter.ch;
        cout << carácter.ch << "      " << (int)carácter.ch << "      : ";
        decodifica(carácter.bits);
    }while (toupper(carácter.ch)!='X');                                // mayúsculas
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJERCICIOS

- 8.1.** Declarar un tipo de datos para representar las estaciones del año.
- 8.2.** Escribir una función que devuelva la estación del año que se ha leído del teclado. La función debe de ser del tipo declarado en el Ejercicio 8.1.
- 8.3.** Escribir una función que reciba el tipo enumerado del Ejercicio 8.1 y lo visualice.
- 8.4.** Declarar un tipo de dato enumerado para representar los meses del año, el mes enero debe de estar asociado al dato entero 1, y así sucesivamente los demás meses.
- 8.5.** Encuentre 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;
}

```

- 8.6.** ¿Con `typedef` se declaran nuevos tipos de datos, o bien permite cambiar el nombre de tipos de datos ya declarados?

PROBLEMAS

- 8.1. Escribir un programa que lea y escriba la información de 100 clientes de una determinada empresa. Los clientes tienen un nombre, el número de unidades solicitadas, el precio de cada unidad y el estado en que se encuentran: moroso, atrasado, pagado.
- 8.2. Añadir al programa anterior una función que permita mostrar la factura de todos los clientes de cada una de las categorías moroso, atrasado o pagado.
- 8.3. Modificar el programa de los Problemas 8.1 y 8.2 para obtener los siguientes listados:
 - Clientes en estado moroso, con factura inferior a una cantidad.
 - Clientes en estado pagado con factura mayor de una determinada cantidad.
- 8.4. Se desea registrar una estructura PersonaEmpleado que contenga como miembros los datos de una persona: el salario y el número de horas trabajadas por semana. Una persona, a su vez es otra estructura que tiene como miembros el nombre, la edad, la altura, el peso, y la fecha de nacimiento. Por su parte, la fecha de nacimiento es otra estructura que contiene el día, el mes y el año. Escribir funciones para leer y escribir un empleado de tipo PersonaEmpleado.
- 8.5. Escribir funciones que permitan hacer las operaciones de suma, resta y multiplicación y cociente de números complejos en forma binómica, $a+bi$. El tipo complejo ha de definirse como una estructura.
- 8.6. Añadir al Problema 8.5 funciones para leer y escribir números complejos, y un programa que permita interactivamente realizar operaciones con números complejos.
- 8.7. Escribir un tipo de datos para representar números complejos en forma polar (módulo, argumento), y codificar dos funciones, para pasar un número complejo de forma polar a binómica y de forma binómica a polar respectivamente.
- 8.8. Escribir funciones que permitan hacer las operaciones de multiplicación y cociente y potencia de números complejos en forma polar.
- 8.9. Se quiere informatizar los resultados obtenidos por los equipos de baloncesto y de fútbol de la localidad alcarricense 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.

8.10. Modificar el programa del Ejercicio 8.9 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.

8.11. 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 .

8.12. Los protocolos IP de direccionamiento de red Internet definen la dirección de cada nodo de una red como un entero largo, pero dicho formato no es muy adecuado para los usuarios. Para la visualización a los usuarios se suelen separar los cuatro bytes separándolos por puntos. Escribir una función que visualice un entero largo como una dirección de red de los protocolos de Internet (cuatro bytes del entero largo separados por puntos).

8.13. Una librería desea tener el inventario de libros. Para ello quiere crear una base de datos en la que se almacenan la siguiente información por libro: título del libro; la fecha de publicación; el autor; el número total de libros existentes; el número total de libros existentes en pedidos; el precio de venta. Escribir funciones que permitan leer los datos de la base de datos y visualizar la base de datos.

SOLUCIÓN DE LOS EJERCICIOS

8.1. Una forma de realizarlo es mediante un tipo enumulado.

```
enum Estaciones {Primavera = 1, Verano = 2, Otonyo = 3, Invierno = 4};
```

8.2. El tipo enumerado asocia enteros a nombres simbólicos, pero estos nombres simbólicos no pueden ser leídos desde una función estándar como `cin`. Por consiguiente, la función que se codifica `leer_Estacion` lee los valores enteros y los traduce a los nombres simbólicos que se corresponden según la definición del tipo enumerado.

```
Estaciones Leer_Estacion (void)
{
    int est;

    cout << " Introduzca el número de la estación del año: ";
    cout << " 1 - Primavera\n";
    cout << " 2 - Verano\n";
    cout << " 3 - Otoño\n";
    cout << " 4 - Invierno\n";
    cin >> est;

    switch (est)
    {
        case 1: return Primavera;
        case 2: return Verano;
        case 3: return Otonyo;
        case 4: return Invierno;
    }
}
```

```

        default: cout << "Entrada errónea \n";
    }
}

```

- 8.3.** La función *Escribir_Estacion* transforma la salida del tipo enumerado en información legible para el usuario.

```

void Escribir_Estacion (Estaciones est )
{
    switch (est)
    {
        case Primavera: cout << " Primavera\n";
        case Verano: cout << " Verano\n";
        case Otonyo: cout << " Otonyo\n";
        case Invierno: cout << " Invierno\n";
    }
}

```

- 8.4.** Una solución es:

```

enum meses {Enero = 1, Fefrero = 2, Marzo = 3, Abril = 4, Mayo = 5,
Junio = 6, Julio = 7, Agosto = 8, Septiembre = 9,
Octubre = 10, Noviembre = 11, Diciembre = 12};

```

- 8.5.** La inicialización de una estructura puede hacerse solamente cuando es una variable estática o global. No se puede definir un array de caracteres sin especificar el tamaño y, además, hay coincidencia de nombres. La mayor parte de los compiladores tampoco permite asignar valores a estructuras tal y como queda reflejado; hay que hacerlo miembro a miembro. Sería conveniente hacerlo mejor en la inicialización. Por último, la función *escribe* tiene como parámetro la estructura *fecha*, que no es global, ya que se declara localmente en el programa principal. Una codificación podría ser la siguiente:

```

#include <stdio.h>
struct fecha                                // declaración global
{
    int dia;
    int mes;
    int anyo;
    char mesc[10];                            // se cambia el nombre
} ff = {1,1,2000,"ENERO"};                  // inicialización

void escribe(fecha f)                         // fecha ya está declarada
{
    ....
}

int main()
{ ....
    escribe(ff);
    return 1;
}

```

- 8.6.** La sentencia no añade ningún tipo de datos nuevos 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* o *float*.

SOLUCIÓN DE LOS PROBLEMAS

- 8.1.** Se define una estructura siguiendo la especificación del problema y con la misma se ha declarado un array que de max clientes que almacena la información de cada uno de los clientes. La función leer_clientes solicita al usuario la información de cada uno de los clientes, mediante un bucle for que recorre los valores 0, 1, 2, ..., max-1. La función escribir_clientes, visualiza la información almacenada en la estructura de datos. El programa principal llama a las dos funciones anteriores.

La codificación de este problema se encuentra en la página Web del libro.

- 8.2.** Se programa la función factura_clientes que recibe como parámetros la lista de clientes y la información del estado de los clientes que deben ser visualizadas. Esta función visualiza la información de todos los clientes cuyo estado coincide con el contenido del parámetro por valor est, así como el importe de la factura. Para hacerlo, se recorre el array con un bucle for comprobando la condición. También se programa la función facturacion_clientes, que mediante un menú solicita al usuario el tipo de clientes que quiere y que se visualice la factura, y realiza la llamada correspondiente a la función factura_clientes.

La codificación de este problema se encuentra en la página Web del libro.

- 8.3.** Para resolver ambos apartados es necesario recorrer mediante un bucle for todos los clientes almacenados en la estructura de datos, y visualizar aquellas estructuras y las facturas asociadas que cumplan las condiciones indicadas. Se programan las funciones clientes_morosos_menor y clientes_pagado_mayor, que resuelven cada uno los correspondientes apartados.

La codificación de este problema se encuentra en la página Web del libro.

- 8.4.** Se trata de un programa de lectura y visualización de estructuras anidadas mediante dos funciones. Primeramente, se declara la estructura fecha, con los atributos indicados. Posteriormente, se declara la estructura persona, con sus atributos, y, por último, la estructura persona_empleado. La función entrada tiene como parámetro por referencia una persona empleado pe, en la que se almacena y retorna toda la información que introduce el usuario. Por su parte la función muestra, presenta los valores que recibe en el parámetro por valor en la consola. La estructura de los registros anidados puede visualizarse en la siguiente figura:

Persona empleado:

Nombre	
Altura	
Edad	
Peso	
Fecha:	Dia
	Mes
	año
Salario	
horas_por_semana	

La codificación de este problema se encuentra en la página Web del libro.

- 8.5.** 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: real + i * imaginario (forma binómica). Donde el símbolo i se denomina "unidad imaginaria" y simboliza la raíz cuadrada de -1 ($i = \sqrt{-1}$). Debido a su naturaleza compuesta, un número complejo se representa de forma natural por una estructura con dos campos de tipo real que contienen la parte real y la imaginaria del número concreto. Las funciones que se codifican traducen las operaciones matemáticas tal y como se definen en la aritmética de números complejos representados mediante una estru-

tura complejo que tiene su parte real y su parte imaginaria. Las funciones matemáticas definidas sobre complejos en forma binómica son las siguientes:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$$

$$\frac{(a + bi)}{(c + di)} = \frac{a*c + b*d}{c*b + d*d} + \frac{-a*d + c*b}{c*b + d*d} i$$

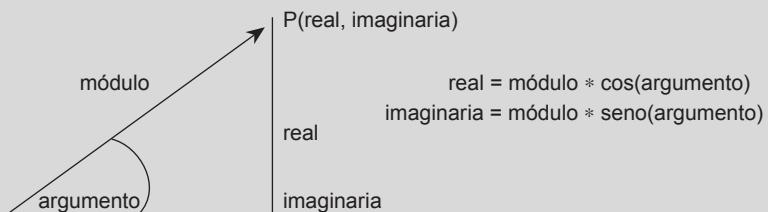
La codificación de este problema se encuentra en la página Web del libro.

- 8.6.** La función `leer_complejo` solicita y lee un número complejo. La información la retorna en un parámetro por referencia. La función `escribir_complejo`, visualiza la parte real y la parte imaginaria del número complejo que recibe como parámetro por valor. `elige opcion_entre`, recibe como parámetro dos números enteros, `x` e `y`, y retorna otro número entero que necesariamente debe estar entre ellos. Esta función es llamada por la función `menu` y garantiza que el usuario elige un número dentro del rango 0, 5 para poder realizar las correspondientes funciones de tratamiento de números complejos en forma binómica tal y como pide el enunciado.

La codificación de este problema se encuentra en la página Web del libro.

- 8.7.** Un numero complejo dado en forma polar, se representa por su módulo y su argumento. Para pasar un número complejo de forma binómica a forma polar se usa la siguiente transformación: la parte real = modulo * seno(argumento); la parte imaginaria = modulo * cos(argumento). La función `pasar_a_binomica`, realiza la transformación indicada.

Para pasar de forma binómica a polar se usa la siguiente transformación: el modulo es igual a la raíz cuadrada de la suma de los cuadrados de la parte real e imaginaria, y el argumento es el arco tangente de la parte real dividido entre la parte imaginaria. Debido a que el arco tangente no distingue entre primer y tercer cuadrante así como entre el segundo y cuarto cuadrante, es necesario tener en cuenta que si la parte imaginaria es negativa, entonces el argumento debe ser incrementado en el valor de $\pi = \text{atan}(-1.0)$. La función `pasar_a_polar`, realiza la transformación.



La codificación de este problema se encuentra en la página Web del libro.

- 8.8.** De acuerdo con las fórmulas matemáticas de tratamiento de números complejos en forma polar se tiene que: para multiplicar dos números complejos en forma polar, basta con multiplicar sus módulos y sumar sus argumentos; para dividirlos basta con calcular el cociente de los módulos, y restar los argumentos; para elevar a una potencia un número complejo en forma polar, basta con elevar el módulo a la potencia indicada, y multiplicar el argumento por la potencia correspondiente.

La codificación de este problema se encuentra en la página Web del libro.

- 8.9.** Se define la estructura `equipo` con la información correspondiente y común a las dos estructuras, tanto de baloncesto, como de fútbol. Se trata pues de definir dos estructuras que aniden en su interior la estructura `equipo`. Se codifica una función `leer_equipo`, que solicita al usuario la información correspondiente a un equipo y la retorna en un parámetro por referencia. Las funciones `leer_Equipo_Baloncesto` y `leer_Equipo_Futbol`, realizan la operación análoga a la de `leer_equipo`, pero ahora solicitan la información de un equipo de baloncesto, o de un equipo de fútbol respectivamente.

Se definen dos array de estructuras de tamaño `max` para almacenar la información correspondiente de los equipos de fútbol y los equipos de baloncesto. La información correspondiente a cada equipo es solicitada en sendos bucles `for` del programa principal: uno para los equipos de fútbol y otro para los equipos de baloncesto.

La codificación de este problema se encuentra en la página Web del libro.

- 8.10.** La función `mejores_Triplistas` recorre el array de equipos de baloncesto y visualizando el nombre e del mejor triplista. Es la función `maximo_Goleador` la que mediante un bucle voraz encuentra el índice del máximo goleador del array de equipos de fútbol. Para calcular el equipo ganador de fútbol `equipo_Ganador_Futbol` se obtienen los puntos de cada equipo y se encuentra el equipo con mayor número o de puntos almacenándolo en la variable `maxpuntos`. Este mismo proceso se realiza para la función `equipo_Ganador_Baloncesto`. Se muestra la codificación de las funciones. Las llamadas a las mismas hay que realizarlas mediante un parámetro que debe ser el nombre e del array que contiene los registros de todos los equipos tanto de fútbol, como de baloncesto.

La codificación de este problema se encuentra en la página Web del libro.

- 8.11.** Las coordenadas de un punto en el plano son dos números reales que se representa en este problema como una estructura punto, con sus dos coordenadas, `x` e `y`. Se codifican las funciones:

- `leer` no tiene ningún parámetro. Realiza la lectura de los datos de un punto mediante la función `cin`, que lee los datos de cada uno de sus miembros para posteriormente retornar el resultado en la propia función definida de tipo punto.
- La distancia entre dos puntos se calcula por la fórmula correspondiente de matemáticas que es la siguiente

$$dis = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

La función `distancia`, recibe como parámetros por valor los dos puntos, y calcula la distancia correspondiente mediante la expresión anteriormente expresada.

- El punto medio se obtiene sumando las dos coordenadas `x` de cada uno de los puntos y las dos coordenadas `y` de los puntos y dividiendo el resultado entre dos. Más concretamente la expresión matemática es la siguiente:

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right)$$

Es la función `medio` la encargada de retornar el punto medio de los dos puntos almacenados en sendas estructuras que recibe como parámetro la encargada de codificar el resultado.

La codificación de este problema se encuentra en la página Web del libro.

- 8.12.** La conversión pedida es sencilla si se usa la característica de las uniones de C++ para acceder a una información de forma diferente. Es la función `direccion_IP`, la que recibe como parámetro el entero largo `d` y visualiza la dirección IP, en la forma clásica en la que normalmente se presenta al usuario. Este entero largo `d` recibido como parámetro, lo almacena en una unión a la que se puede acceder con una función `miembro` o `enterolargo`, o bien con un array de 4 caracteres sin signo (`unsigned char` el bit de signo alto activado no se considera negativo). Cada uno de estos 4 caracteres, que internamente se corresponden con un byte representan un número natural que puede ser escrito como tal. Si entre dos bytes consecutivos se escribe, además, un punto, se tiene la escritura del protocolo de Internet. Observación: esta función puede retornar una cadena de caracteres donde se almacene la dirección IP, y utilizarla posteriormente en otra función, y hacer más eficiente el código de ejecución.

La codificación de este problema se encuentra en la página Web del libro.

- 8.13.** Se declara la estructura `libro` y las funciones miembro definidas en el enunciado. El inventario se almacena en el array de registros `a` de tipo `libro`. El número máximo de libros se define como 100 en la constante `max`. La variable `total` indica en cada momento el número total de libros que hay en el inventario. La función `leer_inventario`, recibe como parámetros por referencia el array de libros `a`, así como la variable `total`. Esta función inicializa `total` a 0, e incrementa la va-

riable total, cada vez que lee los datos de un libro. El final de la entrada de datos, viene indicado por tomar total el valor de max, o bien que la respuesta del usuario sea S, una vez que se solicita la posibilidad de continuar. El bucle for que controla la entrada de datos termina cuando se hace falsa la condición (resp == 'S') && (total < max). La función escribir_inventario, se encarga de visualizar la información almacenada en el array a que recibe como parámetro. El número de datos que contiene el inventario lo recibe la función en el parámetro por valor total.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 8.1. Diseñar 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.
- 8.2. En la base de datos anterior crear un campo de tipo enumerado que permita determinar el departamento al que pertenece un empleado, utilizando un nombre simbólico.
- 8.3. Diseñar una estructura que contenga información de operaciones financieras. Esta estructura debe constar de un número de cuenta, una cantidad de dinero, el tipo de operación (depósito o retirada de fondos) y la fecha y hora en que la operación se ha realizado.
- 8.4. Escribir un enumerado para representar los signos del zodiacal. Escribir funciones que permitan asignar valores al tipo enumerado y extraer valores (lectura y escritura de valores del tipo enumerado).

PROBLEMAS PROPUESTOS

- 8.1. Escribir un programa para calcular el número de días que hay entre dos fechas; declarar fecha como una estructura.
- 8.2. Un número racional se caracteriza por el numerador y denominador. Escribir 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.
- 8.3. 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.
- 8.4. Suponer que se tienen dos arrays del tipo descrito en el problema propuesto 8.1. Codificar un programa en C++ que los une en uno solo, eliminando los duplicados que puedan existir entre los dos.
- 8.5. Suponer que se tiene un array que almacena 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. Escribir 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. Nota ver capítulo de ordenación.
- 8.6. Utilizando el array del Problema 8.5 escribir 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.
- 8.7. Escribir un programa auxiliar que permita añadir nuevos campos a la tabla de empleados, como por ejemplo, sueldo anual y porcentaje de retenciones de impuestos. Una vez modificado el array de estructuras, escribir 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.

- 8.8.** 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 indica 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.

CAPÍTULO 9

Cadenas

Introducción

C++ no tienen 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 estudian 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.

9.1. Concepto de cadena

Una *cadena* es un tipo de dato compuesto, un array de caracteres (`char`), terminado por un carácter *nulo* ('\0'), NULL (Figura 9.1). Una cadena es "ABCD" (también llamada *constante de cadena* o *literal de cadena*). En memoria esta cadena consta de cinco elementos: 'A', 'B', 'C', 'D' y '\0', o de otra manera, se considera que la cadena "ABCD" es un array de cinco 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`.

a	E	j	e	m	p	l	o		d	e		c	a	d	e	n	a	
b	E	j	e	m	p	l	o		d	e		c	a	d	e	n	a	\0

a	A	B	C	D	
b	A	B	C	D	\0

Figura 9.1. a) array de caracteres; b) cadena de caracteres.

El número total de caracteres de un array es siempre igual a la longitud de la cadena más 1. Las funciones declaradas en el archivo de cabecera `<iostream>` se utilizan para manipular cadenas.

Declaración de cadenas. Las cadenas se declaran como los restantes tipos de arrays. El operador postfijo [] contiene el tamaño máximo del objeto. El tipo base, naturalmente, es char, o bien unsigned char:

```
char texto[80];           //una línea de caracteres de texto
char orden[40];           //cadena para recibir una orden del teclado
unsigned char datos;      // activación de bit de orden alto
```

El tipo unsigned char puede ser de interés en aquellos casos en que los caracteres especiales presentes puedan tener el bit de orden alto activado.

A veces, se puede encontrar una declaración como ésta: char *s;.. ¿Es realmente una cadena s? No, no es. Es un puntero a un carácter (el primer carácter de una cadena). Pero las cadenas pueden gestionarse como punteros, ya que un puntero a char, puede ser un array de caracteres.

Inicialización de variables tipo cadena. Las cadenas de caracteres pueden inicializarse en el momento de la declaración tal y como indican los siguientes ejemplos.

```
char alfa[81] = "Esto es una cadena de caracteres";
char cadenatest[] = "¿Cuál es la longitud de esta cadena?";
```

La cadena alfa puede contener un máximo de 80 caracteres más el carácter nulo. La inicialización no puede realizarse con una cadena de longitud superior. La segunda 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. Una cadena no se puede inicializar fuera de la declaración. 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.

EJEMPLO 9.1. Las cadenas terminan con el carácter nulo. El siguiente programa muestra que el carácter NULL ('\0') se añade a la cadena y que es no imprimible.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char S[]="Lucas"; // cadena estática inicializada a 5 caracteres y nulo
    char S1[12] = " nombre:";

    cout << S1 << endl << endl;

    for (int i = 0; i < 6; i++)
        cout << " S[" << i << "] = " << S[i] << "\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

En el resultado de ejecución aparece S[5] = carácter nulo que es no imprimible:

```
nombre:
S[0] = L
S[1] = u
S[2] = c
S[3] = a
S[4] = s
S[5] =
Presione una tecla para continuar
```

9.2. Lectura de cadenas

La lectura usual de datos con el objeto `cin` y el operador `>>`, si se aplica a datos de cadena produce anomalías, ya que el objeto `cin` termina la operación de lectura siempre que se encuentra un espacio en blanco. El método recomendado para la lectura de cadenas es utilizar una función denominada `getline()`, en unión con `cin`, en lugar del operador `>>`. La función `getline` permite a `cin` leer la cadena completa, incluyendo cualquier espacio en blanco.

EJEMPLO 9.2. *Este programa muestra como `cin` lee datos de tipo cadena, y cómo lo hace `cin.getline`.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char Ejemplo[31]; // Definir array de caracteres

    cout << " introduzca frase \n ";
    cin.getline(Ejemplo,30); // lectura de cadena completa
    cout << "\t\" << Ejemplo << "\"\n";
    cout << " introduzca otra frase \n ";
    cin >> Ejemplo; // lectura de palabra hasta encontrar blanco
    cout << "\t\" << Ejemplo << "\"\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Resultado de ejecución:

```
nombre:
S[0] = L
S[1] = u
S[2] = i
S[3] = s
S[4] =
S[5] =
Presione una tecla para continuar
```

El objeto `cin` del flujo de entrada incluye entre otras las funciones de entrada:

```
cin.getline( ), cin.get( ), cin.ignore( ), cin.putback( ).
```

`cin.getline()`. `cin` es un objeto de la clase `istream` y `getline()` es una función miembro de la clase `iostream`. La sintaxis de la función `getline()` es:

```
istream& getline(signed char* buffer, int long, char separador = '\n');
```

La lectura de cadenas con `cin` se realiza con el siguiente formato:

```
cin.getline(<var_cad>, <max_long_cadena+2>, <'separador'>);
```

El primer argumento de `getline` es el identificador de la variable cadena. El segundo argumento es el número máximo de caracteres que se leerán y debe ser al menos dos caracteres mayor que la cadena real, para permitir el carácter nulo '`\0`' y el '`\n`'. Por último, el carácter separador se lee y almacena como el siguiente al último carácter de la cadena y es el último carácter que se lee. La función `getline()` inserta automáticamente el carácter nulo como el último carácter de la cadena.

Reglas:

- La llamada `cin.getline(cad, n, car)` lee todas las entradas hasta la primera ocurrencia del carácter separador `car` en `cad`.
- Si el carácter especificado `car` es el carácter de nueva línea '`\n`', la llamada anterior es equivalente a `cin.getline(cad, n)`.

Problemas con la función `getline()`. La función `getline()` presenta problemas cuando se intente utilizar una variable cadena, después de que se ha utilizado `cin` para leer una variable carácter o numérico. La razón es que no se lee el retorno de carro `<INTRO>`, por lo que permanece en el buffer de entrada. Cuando se da la orden `getline()`, se lee y almacena este retorno que por defecto es el carácter separador, por lo que no se lee la cadena de caracteres que se pretende sea el dato.

EJEMPLO 9.3. *Lectura y escritura de cadenas de caracteres. Se codifican dos funciones, una para leer un registro y otra para escribirlo.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

struct registro
{
    char Nombre_y_apellidos[51];
    char Calle[33], Poblacion[28];
    char Provincia[28], Pais[21], Codigo_Postal[8];
}r;

void Leer_registro(registro &r)
{
    cout << " Lectura de datos \n";
    cout << " Nombre y apellidos: "; cin.getline(r.Nombre_y_apellidos, 50);
    cout << " Calle: "; cin.getline(r.Calle, 32);
    cout << " Poblacion: "; cin.getline(r.Poblacion, 27);
    cout << " Provincia: "; cin.getline(r.Provincia, 27);
    cout << " Pais: "; cin.getline(r.Pais, 20);
    cout << " Codigo Postal: "; cin.getline(r.Codigo_Postal, 7);
}

void Escribir_registro(registro r)
{
    cout << "\n Visualizacion de datos \n \n";
    cout << " " << r.Nombre_y_apellidos << endl;
    cout << " " << r.Calle << endl;;
    cout << " " << r.Poblacion << endl;
    cout << " " << r.Provincia << endl;
    cout << " " << r.Pais << endl;
    cout << " " << r.Codigo_Postal << endl;
}

int main(int argc, char *argv[])
{
    Leer_registro(r);
    Escribir_registro(r);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

`cin.get()`. La función `cin.get()` se utiliza para leer carácter a carácter. La llamada `cin.get(car)` copia el carácter siguiente del flujo de entrada `cin` en la variable `car` y devuelve 1, a menos que se detecte el final del archivo, en cuyo caso se devuelve 0.

EJEMPLO 9.4. (*Uso de cingetc*). La función `esvocal` decide si el carácter que recibe como parámetro es una vocal. El programa principal itera hasta que se teclee el final del flujo de entrada, dado por control + Z.

```
#include <cstdlib>
#include <iostream>
using namespace std;

bool esvocal(char ch)
{
    ch= toupper(ch);
    return ( ch == 'A' || ch == 'E' ||ch == 'I' ||ch == 'O' ||ch == 'U');
}

int main(int argc, char *argv[])
{
    char ch;
    int cuenta = 0;

    while (cin.get(ch))
        if (esvocal(ch))
            cuenta++;
    cout << " numero de vocales leidas " << cuenta << "\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

`cout.put`. La función opuesta de `get` es `put`. La función `cout.put()` se utiliza para escribir en el flujo de salida `cout` carácter a carácter.

EJEMPLO 9.5. El siguiente programa lee un texto y lo visualiza, escribiendo todas las vocales en minúscula y las consonantes en mayúsculas.

```

#include <cstdlib>
#include <iostream>
using namespace std;

bool esvocal(char ch)
{
    ch= toupper(ch);
    return ( ch == 'A' || ch == 'E' ||ch == 'I' ||ch == 'O' ||ch == 'U');
}

int main(int argc, char *argv[])
{
    char ch;

    while (cin.get(ch))
    {
        if (esvocal(ch))                                // es una vocal
            cout.put(tolower(ch));                     // escribe en minúscula
        else if(isalpha(ch))                           // es letra

```

```

        cout.put(toupper(ch));           // escribe minúscula
    else cout.put(ch);                // escribe como se lee
}
system("PAUSE");
return EXIT_SUCCESS;
}

```

La función `cin.putback()` restaura el último carácter leído por `cin.get()` de nuevo al flujo de entrada `cin`. La función `cin.ignore()` lee uno o más caracteres del flujo de entrada `cin` sin procesar.

EJEMPLO 9.6. *El siguiente programa lee líneas del teclado hasta que se introduce control más Z. En cada línea debe haber dos números reales. El programa presenta por cada línea leída otra línea con la suma de los dos números.*

La expresión `cin.get(ch)` copia el siguiente carácter en `ch` y devuelve 1 si tiene éxito. A continuación, si `ch` es un dígito, el número real completo se lee en `r` y se devuelve. En caso contrario, el carácter se elimina de `cin` y continúa el bucle. Si se encuentra el final del archivo, la expresión `cin.get(ch)` devuelve 0, y se detiene el bucle.

```

#include <cstdlib>
#include <iostream>
using namespace std;

float siguienteReal()
{ // busca el primer carácter dígito y lee el número real
char ch;
float r;

while (cin.get(ch))
if (ch >= '0' && ch <= '9')
{
    cin.putback(ch);                      // restaura
    cin >> r;                            // lee dato real
    break;
}
return r;
}

int main(int argc, char *argv[])
{
    float r1, r2;
    char ch;

cout << "introduzca linea con dos numeros a sumar control+z fin \n";
while(cin.get(ch))
{
    cin.putback(ch);                      // restaura para leer reales
    r1 = siguienteReal(), r2 = siguienteReal();
    cin.ignore(80,'\'\n');                 // salta hasta fin línea
    cout << r1 << " + " << r2 << " = " << r1 + r2 << endl;
}
system("PAUSE");
return EXIT_SUCCESS;
}

```

EJEMPLO 9.7. Función que obtiene una cadena del dispositivo de entrada, de igual forma que la función miembro `getline` de `cin` utilizando la función miembro `getc` de `cin`.

La función `cin.getline()` lee caracteres de la entrada estándar, normalmente el teclado, hasta que se le introduce un salto de línea, o se llega al máximo de caracteres a leer o bien ha leído un carácter que se le pasa como parámetro. Los caracteres que recibe son colocados en la dirección donde indique el argumento de la función, en caso de que disponga de uno. Se utiliza la sobrecarga de funciones explicada en el Capítulo 6 para definir la función `getline2`. La primera codificación `void getline2 (char* cadena, int n)` admite como parámetros la cadena y la longitud máxima, por lo que la lectura y almacenamiento de caracteres debe realizarse hasta que sea fín de fichero, se haya leído el fin de línea '`\n`', y, además, no se haya leído el número máximo de caracteres. La segunda codificación `void getline2 (char* cadena, int n, char ch)` es análoga a la primera, pero en este caso uno de los finales de la lectura de datos, en lugar de ser el fin de línea '`\n`', es el contenido de `ch`.

```
void getline2 (char* cadena, int n)
{
    char c, *p = cadena;
    int i = 0;

    while (( cin.get(c)) && (c != '\n') && (i < n))
    {
        *p++ = c;
        i++;
    }
    *p = '\0';
}

void getline2 (char* cadena, int n, char ch)
{
    char c, *p = cadena;
    int i = 0;

    while (( cin.get(c)) && (c != ch) && (i < n))
    {
        *p++ = c;
        i++;
    }
    *p = '\0';
}
```

9.3. La biblioteca `string.h`

La biblioteca estándar de C++ contiene la biblioteca de cadena `iostream`, que contiene las funciones de manipulación de cadenas utilizadas más frecuentemente. El uso de las funciones de cadena tienen una variable parámetro declarada similar a: `char *s1;`. Cuando se utiliza la función, se puede usar un puntero a una cadena o se puede especificar el nombre de una variable array `char`. Cuando se pasa un array a una función, C++ pasa automáticamente la dirección del array `char`. Las funciones que incluyen la palabra reservada `const`, lo que permite ver rápidamente la diferencia entre los parámetros de entrada (`const`) y salida. La Tabla 9.1 resume algunas de las funciones de cadena más usuales.

EJEMPLO 9.8. Ejemplo de programación de la función que realiza la misma operación que `strcpy`.

```
char * strcpya( char * destino, const char* origen)
{
    int lon = strlen(origen); // se calcula longitud
```

```

destino = new char [lon + 1];           // se reserva memoria
for (int i = 0; i < lon; i++)          //se copia carácter a carácter
    destino[i]= origen[i];
destino[lon] = '\0';                  // se añade fin decadena
return destino;                      // retorno de la copia
}

```

Tabla 9.1. Funciones de <string.h>

Función	Cabecera de la función y prototipo
memcpy()	void* memcpy(void* s1, const void* s2, size_t n). Reemplaza los primeros <i>n</i> bytes de *s1 con los primeros <i>n</i> bytes de *s2. Devuelve s.
strcat	char *strcat(char *destino, const char *fuente). Añade la cadena fuente al final de destino.
strchr()	char* strchr(char* s1, const char* s2). Devuelve un puntero a la primera ocurrencia de c en s. Devuelve NULL si c no está en s.
strcmp	int strcmp(const char *s1, const char *s2). Compara la cadena s1 a s2 y devuelve: 0 si s1 == s2, <0 si s1 < s2, >0 si s1 > s2.
strcmpi	int strcmpi(const char *s1, const char *s2). Igual que strcmp(), pero trata los caracteres como si fueran todos del mismo tamaño.
strcpy	char *strcpy(char *dest, const char *fuente). Copia la cadena fuente a la cadena destino.
strcspn()	size_t strcspn(char* s1, const char* s2). Devuelve la longitud de la subcadena más larga de s1 que comienza con s1[10] y no contiene ninguno de los caracteres encontrados en s2.
strlen	size_t strlen (const char *s). Devuelve la longitud de la cadena s.
strncat()	char* strncat(char* s1, const char*s2, size_t n). Añade los primeros n caracteres de s2 a s1. Devuelve s1. Si n >= strlen(s2), entonces strncat(s1, s2, n) tiene el mismo efecto que strncat(s1, s2).
strncmp()	int strncmp(const char* s1, const char* s2, size_t n). Compara s1 con la subcadena s de los primeros n caracteres de s2. Devuelve un entero negativo, cero o un entero positivo, según que s1 lexicográficamente sea menor, igual o mayor que s. Si n ≈ strlen(s2), entonces strncmp(s1, s2, n) y strcmp(s1, s2) tienen el mismo efecto.
strnset	char *strnset(char *s, int ch, size_t n). Utiliza strcmp() sobre una cadena existente para fijar n bytes de la cadena al carácter ch.
strupr()	char*strupr(const char* s1, const char* s2). Devuelve la dirección de la primera ocurrencia de s1 de cualquiera de los caracteres de s2. Devuelve NULL si ninguno de los caracteres de s2 aparece en s1.
strrchr()	char* strrchr(const char* s, int c). Devuelve un puntero a la última ocurrencia de c en s. Devuelve NULL si c no está en s.
strspn()	size_t strspn(char* s1, const char* s2). Devuelve la longitud de la subcadena más larga de s1 que comienza con s2[0] y contiene únicamente caracteres encontrados en s2.
strstr	char *strstr(const char *s1, const char *s2). Busca la cadena s2 en s1 y devuelve un puntero a los caracteres donde se encuentra s2.
strtok()	char* strtok(char* s1, const char* s2). Analiza la cadena s1 en tokens (componentes léxicos) delimitados por los caracteres encontrados en la cadena s2. Después de la llamada inicial strtok(s1, s2), cada llamada sucesiva a strtok(NULL, s2) devuelve un puntero al siguiente token encontrado en s1. Estas llamadas cambian la cadena s1, reemplazando cada separador con el carácter NUL ('\0').

EJEMPLO 9.9. Leer un texto cuyo máximo número de líneas sea 60 de una longitud máxima de 80 caracteres por línea, y escribir el mismo texto, pero intercambiando entre sí las líneas de mayor con la de menor longitud.

El texto se maneja a partir de una matriz de caracteres de 60 filas por 81 columnas. En cada una de las filas se almacena una línea de entrada. Mediante un bucle controlado por la variable entera i, se van leyendo las líneas del texto y almacenándolas en la matriz por filas. A la vez que se almacena se va calculando la longitud y posición de las líneas más corta y más larga, para que cuando se termina el bucle, se puedan intercambiar entre sí las líneas más cor-

ta con la más larga. El bucle `while` termina cuando se han leído las 60 líneas, o bien cuando se introduce una línea vacía, ya que en ese momento se rompe la ejecución de bucle.

```
#include <cstdlib>
#include <iostream>
using namespace std;
#define max 60

int main(int argc, char *argv[])
{
    char texto[max][81];
    int i = 0, j, lon,
        int longmax = 0, posmax, longmin = 81, posmin;
    char linea [80];

    while (i < max)
    {
        cin.getline(linea, 80);
        lon = strlen(linea);
        if (lon == 0)
            break;
        if (lon < longmin)
        {
            posmin = i;
            longmin = strlen(linea);
        }
        if (lon > longmax)
        {
            posmax = i;
            longmax = strlen (linea);
        }
        strcpy (texto [i], linea);
        i++;
    }

    strcpy (linea, texto[posmin]);
    strcpy (texto[posmin], texto[posmax]);
    strcpy (texto[posmax], linea);
    cout << " texto intercambiado \n";
    for (j = 0; j < i; j++)
        cout << texto[j] << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

9.4. 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. 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.

La función `atol()` convierte una cadena a un valor largo (`long`). Su prototipo es:

```
long atol(const char*cad);
```

EJEMPLO 9.10. *Lectura de números en líneas, y acumulación de los números. Se sabe que los valores numéricos de cada línea están separados por blancos o el carácter f in de línea, y que representan kilogramos de un producto. En las líneas puede haber, además, cualquier otro carácter. El siguiente programa lee el texto y obtiene la suma de los valores numéricos de cada una de las líneas así como el total de la suma de todos los valores numéricos. El fin de texto viene dado por una línea en blanco.*

La función `strtok(s1,s2)` analiza la cadena `s1` en *tokens* (componentes léxicos) delimitados por los caracteres encontrados en la cadena `s2`. Después de la llamada inicial `strtok(s1, " ")`, cada llamada sucesiva a `strtok(NULL, " ")` devuelve un puntero al siguiente *token* encontrado en `s1`. Como la función estándar de concatenación 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 que se necesita saber. Se programa un bucle infinito, cuya salida viene determinada con la orden `break`, que se ejecuta cuando se introduce una línea vacía. Los acumuladores `sumalinea` y `suma`, se encargan de sumar los números introducidos en una línea y en el total de las líneas respectivamente.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char buffer [80], *ptr;
    int kilos, suma = 0, sumalinea;

    cout << " Introduzca el texto linea a linea. \n";
    for (; ; )
    {
        sumalinea = 0;
        cin.getline(buffer, 80);
        if (strlen(buffer) == 0)
            break;                                // salida del bucle
        ptr = strtok(buffer, " ");
        while (ptr)
        {
            if ((kilos = atoi (ptr)) != 0)
                sumalinea += kilos;
            ptr = strtok(NULL, " ");
        }
        cout << " La suma de kg en linea es: " << sumalinea << endl;
        suma += sumalinea;
    }
    cout << " La suma de kg de todas las lineas es: " << suma << endl; system("PAUSE");
}
```

Las funciones `strtol` y `strtoul` convierten los dígitos de una cadena, escrita en cualquier base de numeración a un entero largo o a un entero largo sin signo. El prototipo de ambas funciones es:

```
long strtol(const char * c, char**pc, int base) y
unsigned long strtoul(const char c, char**pc, int base)
```

transforma el número escrito en la cadena `c` en una cierta base en números enteros o enteros largos, moviendo `*pc` al final del número leído dentro de la cadena `c`. Si el número está incorrectamente escrito en la base.

EJEMPLO 9.11. *Convertir una cadena numérica a números enteros largos y entero sin signo en una base.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{   char *cadena= " 32 111";
    char **pcadena = new(char*) ;
    long numero1;
    unsigned long numero2;

    numero1 = strtol (cadena,pcadena,4);           // extrae numero en base 4
    cout << " numero = "<< numero1 << endl;
    cout << " cadena actual "<< *pcadena << endl;
    cadena = *pcadena;
    numero2 = strtoul (cadena, pcadena,2);         // extrae numero en base 2
    cout << " n2 = "<< numero2 << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJERCICIOS

9.1. ¿Cuál de las siguientes declaraciones son equivalentes?

```
char var_cad0[10] ="Hola";
char var_cad1[10] = { 'H','o','l','a'};
char var_cad2[10]= { 'H','o','l','a','\0'};
char var_cad3[5]= "Hola";
char var_cad4[]="Hola";
```

9.2. ¿Qué hay de incorrecto (si existe) en el siguiente código?

```
char var_cad[]="Hola";
strcat(var_cad, " y adios");
cout << var_cad<< endl;
```

9.3. Suponga que no existe el código de la función `strlen`. Escriba el código de una función que realice la tarea indicada.

9.4. ¿Qué diferencias y analogías existen entre las variables `c1`, `c2`, `c3`? La declaración es:

```
char** c1;
char* c2[10];
char* c3[10][21];
```

- 9.5. Escribir un programa que lea dos cadenas de caracteres, las visualice junto con su longitud, las concatene y visualice la concatenación y su longitud.
- 9.6. La carrera de sociología tiene un total de N asignaturas. Escribir una función que lea del dispositivo estándar de entrada las N asignaturas con sus correspondientes códigos.
- 9.7. Añadir al Ejercicio 9.6 funciones para visualizar las N asignaturas y modificar una asignatura determinada.
- 9.8. Escriba un programa que lea una cadena de caracteres de la entrada y la invierta.

PROBLEMAS

- 9.1. La función `atoi()` transforma una cadena formada por dígitos decimales en el equivalente número o entero. Escribir una función que transforme una cadena formada por dígitos hexadecimales en un entero largo.
- 9.2. Definir un array de cadenas de caracteres para poder leer un texto compuesto por un máximo de 80 caracteres por líneas. Escribir una función para leer el texto, y otra para escribirlo; las funciones deben tener dos argumentos, uno el texto y el segundo el número de líneas.
- 9.3. Se sabe que en las 100 líneas que forman un texto hay valores numéricos enteros, que 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 todos los valores numéricos.
- 9.4. Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.
- 9.5. 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.
- 9.6. Escribir un programa para las líneas de un texto sabiendo que el máximo de caracteres por línea es 80 caracteres. Contar el número de palabras que tiene cada línea, así como el número total de palabras leídas.
- 9.7. Se tiene un texto formado por un máximo de 30 líneas, del cual se quiere saber el número de apariciones de una palabra clave. Escribir un programa que lea la palabra clave determine el número de apariciones en el texto.
- 9.8. 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 ordenadas creciente de su longitud.
- 9.9. Escribir un programa que lea una cadena clave y un texto de, como máximo, 50 líneas. El programa debe eliminar las líneas que contengan la clave.
- 9.10. Escribir una función que reciba como parámetro una cadena de caracteres y la invierta, sin usar la función `strrev`.
- 9.11. Escribir una función que reciba una cadena de caracteres, una longitud lon , y un carácter ch . La función debe retornar otra cadena de longitud lon , que contenga la cadena de caracteres y si es necesario, el carácter ch repetido al final de la cadena las veces que sea necesario.

- 9.12.** Se quiere sumar números grandes que no puedan almacenarse en variables de tipo `long`. Por esta razón 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.
- 9.13.** Escribir una función que reciba como parámetros un número grande como cadena de caracteres, y lo multiplique por un dígito, que reciba como parámetro de tipo carácter.
- 9.14.** Escribir una función que multiplique dos números grandes, recibidos como cadenas de caracteres.
- 9.15.** 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 llenar 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.

SOLUCIÓN DE LOS EJERCICIOS

- 9.1.** Todas las declaraciones realizan inicializaciones de array de caracteres a cadenas, excepto la declaración correspondiente a `var_cad1`, que inicializa un array de caracteres, simplemente, ya que no termina en el carácter nulo '`\0`'. Son equivalentes las inicializaciones de las variables `var_cad0` y `var_cad2` entre sí, ya que la primera se inicializa a una constante cadena, y la segunda se inicializa cada posición del array a los mismos caracteres terminando en el carácter nulo. También son equivalentes las inicializaciones de `var_cad3` y `var_cad4`, a pesar de que la segunda inicialización es de un array indeterminado.
- 9.2.** La primera sentencia inicializa la variable `var_cad` de longitud variable a la cadena de caracteres "Hola". La siguiente sentencia concatena la cadena de caracteres, "y adios", y la siguiente sentencia visualiza el resultado. Por tanto, no existe error en el código. Si se ejecuta el siguiente programa se puede comprobar el resultado.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    char var_cad[] = "Hola";

    strcat(var_cad, " y adios");
    cout << var_cad << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- 9.3.** Se utiliza `typedef` para definir `cadena80`, como un array de 80 caracteres. La función `Longitud` recibe como parámetro por referencia una cadena de caracteres. El cálculo de la longitud se realiza con un bucle `while` que itera hasta que el contenido de la posición del array a la que se accede con contador sea `NULL`. El programa principal realiza una llamada a la función para comprobar resultados.

```
#include <cstdlib>
#include <iostream>
```

```

using namespace std;
typedef char cadena80[80];

int Longitud(cadena80 &cad)
{
    int contador = 0;

    while (cad[contador] != '\0')
        contador++;
    return contador;
}

int main(int argc, char *argv[])
{
    cadena80 cad = "C++ es mejor que C";

    cout << "longitud de " << cad << " = " << Longitud(cad) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 9.4.** La variable *c1* es una variable puntero que apunta a un puntero de caracteres. La variable *c2* es un array de 10 punteros a caracteres. La variable *c3* es una matriz con espacio para 210 punteros a caracteres accesibles según un arreglo de 10 filas de 21 elementos cada una de ellas.
- 9.5.** Se presenta la codificación programa así como un resultado de ejecución. Se utiliza la función *strcat* que concatena dos cadenas de caracteres.

```

int main(int argc, char *argv[])
{
    char s1[81], s2[81];

    cout << " introduzca cadena :";
    cin.getline(s1,80);
    cout << " introduzca cadena :";
    cin.getline(s2,80);
    cout << " Antes de strcat(s1, s2): \n";
    cout << " s1 = \" " << s1 << "\", longitud = " << strlen(s1) << endl;
    cout << " s2 = \" " << s2 << "\", longitud = " << strlen(s2) << endl;
    strcat(s1, s2);
    cout << " Despues de strcat(s1, s2):\n";
    cout << " s1 = \" " << s1 << "\", longitud = " << strlen(s1) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 9.6.** Se supone que el número de asignaturas *N* es constante y vale 50. Este valor de *N* se define en el macro `#define N 50`. Se supone que la asignatura tiene un nombre de menos de 20 caracteres y que su código no sobrepasa los cinco caracteres alfanuméricos. Se declara dos array *asignatura* y *codigo* que almacenan los nombres y códigos de las *N* asignatura. Mediante un bucle *for*, se solicitan al usuario los datos, realizando la lectura con *cin.getline*. La función *leer*, la que realiza la tarea solicitada, devolviendo las asignaturas y los códigos en dos parámetros.

```

#include <cstdlib>
#include <iostream>
using namespace std;

char asignatura [N][21], codigo [N][7];

void leer(char asignatura[][21], char codigo[][7], int n)
{
    for (int i = 0; i <= n; i++)
    {
        cout << " Escriba el nombre de la asignatura: " << i + 1 << " ";
        cin.getline(asignatura[i], 20);
        cout << " Escriba el código de la asignatura: " << i + 1 << " ";
        cin.getline(codigo[i], 6);
    }
}

#define N 50
int main(int argc, char *argv[])
{
    .....
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 9.7.** La visualización de las asignaturas y los códigos se realizaron por la función `escribir`, que presenta los datos que recibe en sus parámetros. El número total de asignaturas a visualizar es `n`, que debe estar en el rango $0, \dots, N-1$. Para modificar los datos de una asignatura determinada `i`, basta con solicitarlos al usuario, y rotarlos, en la posición correspondiente de los array que se reciben como parámetros.

```

void escribir(char asignatura[][21], char codigo[][7], int n)
{
    for (int i = 0; i <= n; i++)
    {
        cout << " nombre de la asignatura: " << i + 1 << asignatura[i] << endl;
        cout << " código de la asignatura: " << i << codigo[i] << endl;
    }
}

void modificar(char asignatura[][21], char codigo[][7], int i)
{
    cout << " Escriba nuevo nombre de asignatura: " << i + 1 << " ";
    cin.getline(asignatura[i], 20);
    cout << " Escriba nuevo código de asignatura: " << i + 1 << " ";
    cin.getline(codigo[i], 6);
}

```

- 9.8.** Se usa la función `strrev` que recibe como entrada una cadena y la invierte:

```

#include <cstdlib>
#include <iostream>
using namespace std;

```

```

int main(int argc, char *argv[])
{
    char cadena[41] ;

    cout << "deme cadena ";
    cin.getline(cadena,40);
    strrev(cadena);
    cout << "Cadena invertida \n" << cadena << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

SOLUCIÓN DE LOS PROBLEMAS

- 9.1.** La función `Hexadecimal_a_enterolargo` recibe como parámetro una cadena de caracteres `Hexa` en la que se encuentran escritos los dígitos de un número en base hexadecimal, y lo transforma a un entero largo escrito en base 10 devolviéndolo en la propia función, que es declarada como de tipo entero largo. Para realizar la operación se usa, el método de Horner de evaluación de un polinomio escrito en una cierta base:

$$"2345"_{16} = ((0 * 16 + 2) * 16 + 3) * 16 + 4) * 16 + 5.$$

Se inicializa una variable `numero` a 0. Mediante un bucle `for` se recorren todas las posiciones de `Hexa`, hasta que no queden datos `cadena[c] == '\0'`. El carácter almacenado en la posición `c` se transforma en un dato numérico con la expresión `Valordígito = Hexa[c] - '0'`, si es un dígito, y mediante la expresión `Valordígito = Hexa[c] - 'A' + 10`, en el caso de que sea un carácter de entre 'A', 'B', 'C', 'D', 'E', 'F'. Una vez obtenido el valor del dígito se realiza la transformación de Horner `numero = numero * base + Valordígito`.

La codificación de este problema se encuentra en la página Web del libro.

- 9.2.** Se declara una `lnea` como un sinónimo de una cadena de caracteres de longitud 80. El texto `text` es un puntero a las líneas, que es el array donde se almacenará el texto. A este texto se le asigna memoria en el programa principal, y se llaman a las funciones `Leer_Texto` y `Escribir_Texto`, que son las encargadas de leer los datos y visualizar los resultados.

La codificación de este problema se encuentra en la página Web del libro.

- 9.3.** Se declara la variable `TextoEntrada` como un array de 100 posiciones de punteros a `char`, para poder almacenar las 100 líneas del texto de entrada. A cada una de estas líneas del texto se le asigna memoria dinámicamente mediante la sentencia `new`, una vez que se sabe la longitud de la línea. Esta longitud viene determinada por la función `strlen` aplicada a la línea de entrada previamente leída con la orden `cin.getline()`. La entrada de información de cada línea se realiza con un bucle `for` que itera 100 veces. Es mediante otro bucle `for` como se visualiza al final del programa todo el texto leído del dispositivo estándar de entrada. Para encontrar separar, y acumular los números del texto, se usan las funciones: `atoi` que devuelve el número almacenado en la cadena o cero en caso que no comience por un número entero, y la función `strtok`, que se encarga de ir desplazando un puntero `ptr` a lo largo de la línea de entrada, en cada blanco que encuentra.

La codificación de este problema se encuentra en la página Web del libro.

- 9.4.** La función que se pide debe retornar tres valores, por ello se definen tres parámetros por referencia para retornarlos. Para averiguar el tipo del carácter que almacena en cada posición la cadena, se recorre la cadena con un puntero auxiliar y se compara su código ASCII con el de los números y las letras, usando las funciones `isupper`, `tolower` y `isdigit`. El bucle

while que controla el carácter que se está comparando termina cuando llega al final de la cadena, detectado por coincidir el número ASCII del contenido del puntero con el número ASCII del carácter nulo.

La codificación de este problema se encuentra en la página Web del libro.

- 9.5.** *Se codifica en primer lugar la función tolowercad que recibe como parámetro una cadena de longitud máxima 80 y devuelve la cadena en la que todas las letras mayúsculas son convertidas en minúsculas. Para realizarlo itera sobre todas las posiciones de la cadena de caracteres usando la función tolower. Para decidir si dos cadenas de caracteres son anagramas basta convertir todos los caracteres de las cadenas que sean letras mayúsculas en letras minúsculas y contar el número de apariciones de cada carácter que sea una letra mayúscula en letras minúsculas. Para contar el número de veces que aparece cada letra minúscula en las respectivas cadenas, se usan dos arrays de enteros letras1 y letras2, que almacenan en sus 28 posiciones, las apariciones de 'a' en la posición 0, 'b' en la posición 1, etc.*

La codificación de este problema se encuentra en la página Web del libro.

- 9.6.** *Cada línea se lee llamando a la función cin.getline, con un argumento que pueda almacenar el máximo de caracteres de una línea. Por consiguiente, se declara la variable: char cad[81], argumento de cin.getline. La longitud de la cadena se determina con una llamada a strlen. Se definen dos funciones, la primera saltablancos sirve para saltarse los blancos que hay antes de cualquier palabra, y la segunda saltapalabra se utiliza para saltarse una palabra completa. Ambas funciones reciben como parámetro la cadena de caracteres cad, así como la posición i por la que debe comenzar la búsqueda, y retornan en el propio parámetro por referencia i la posición donde ha concluido su búsqueda. El número de palabras de cada línea se determina, buscando el comienzo de la primera palabra con la función saltablancos y, posteriormente, llamando en un bucle a las funciones saltapalabra y saltablancos. La ejecución termina cuando se introduce una línea sin datos o bien el fin de archivo. El contador n cuenta el número de palabras de cada línea, y el acumulador total, acumula las palabras totales leídas.*

La codificación de este problema se encuentra en la página Web del libro.

- 9.7.** *Se trata de realizar una búsqueda con strstr() en cada una de las líneas del texto. En primer lugar, se lee la palabra clave que se supone de longitud máxima, menor o igual que 14. El texto se representa por un array de un máximo de 30 posiciones de punteros a caracteres. En cada una de esas posiciones del array, se almacena una línea a la que se le asigna memoria mediante la orden new de reserva de espacio. Cada vez que se lee una línea y se almacena en el texto, se determina el número de veces que la palabra clave está en la línea. No hay que olvidar que la función strstr retorna un puntero L a la posición de la primera aparición de la palabra clave, por lo que si se quiere avanzar en la búsqueda hay que incrementarlo.*

La codificación de este problema se encuentra en la página Web del libro.

- 9.8.** *Se trata de leer un texto y de manipular los punteros del texto, para no tener que trasladar las propias líneas. Como en C++ las cadenas de caracteres no guardan información acerca de su propia longitud, y puesto que la marca de final de línea, '\0', es suficiente para determinar su extensión, se usa un array auxiliar longitudlineas para almacenar la longitud de cada línea y la posición inicial que tienen en el texto (indexación para poder ordenar y no mover líneas del texto). El texto se almacena en un array de 40 posiciones con punteros a caracteres (en cada una de estas posiciones se almacenará una línea).*

- La función leer_texto recibe como parámetro el array del texto, el array para la indexación y un parámetro por referencia i, que indica cuántas líneas del texto reales se han leído. Esta función lee el texto línea a línea, almacenado en el array de indexación su longitud y la posición que ocupa en el texto. El bucle que controla la lectura termina cuando se lee una línea en blanco.
- La función ordenar, ordena por longitud de línea el array de indexación por el conocido método de la burbuja (véase métodos de ordenación del Capítulo 12).
- La función escribir_texto se encarga de visualizar el texto con las líneas del original en las posiciones que indica el array de indexación que contiene las longitudes ordenadas y las posiciones que ocupa.
- El programa principal realiza las llamadas a las funciones para resolver el problema planteado.

La codificación de este problema se encuentra en la página Web del libro.

- 9.9.** Se declara *texto* como un array de cadenas de caracteres de longitud variable. Asimismo, se declara un puntero a cadenas de caracteres *ptr* usado en la función *strstr* para determinar si la clave leída se encuentra en la cadena. Esta clave se almacena en una cadena de caracteres de longitud máxima 15. El bucle se repite mediante una variable entera *j* un total de 50 veces leyendo líneas y almacenándolas en el texto si no aparece la clave. Para realizar el almacenamiento, se dispone de una variable entera *j* que indica la posición del texto donde debe ser almacenada la línea actual que se lee en caso que haya que hacerlo. Una línea se almacena si no se encuentra a la clave en ella. En caso de que así sea (debe quedar en el texto), se reserva espacio en memoria para la línea, se copia con la orden *strcpy* y se incrementa el contador *j* en una unidad. Al final del programa se presenta el texto sin las líneas que contienen la clave en ella.

La codificación de este problema se encuentra en la página Web del libro.

- 9.10.** La función *reverse* resuelve el problema. Para ello se calcula la longitud de la cadena, se crea espacio en memoria para ella, y mediante un bucle *for* se realiza la asignación *rcadena[lon - i - 1] = cadena[i]*, que va cambiando las posiciones de los caracteres. Recuerde que toda cadena termina siempre en el carácter nulo, por lo que es necesario añadirlo después del bucle *for*. El resultado de la cadena invertida se retorna en la función.

La codificación de este problema se encuentra en la página Web del libro.

- 9.11.** La función recibe como parámetro una cadena y retorna una cadena de longitud *lon* que contiene lo siguiente: si la cadena del parámetro de entrada tiene longitud mayor que *lon* toma los *lon* primeros caracteres de la cadena; en otro caso a la cadena del parámetro de entrada se le añade el carácter *ch* hasta que tenga longitud *lon*. Para realizarlo basta con copiar la cadena y añadir el carácter *ch* las veces que sean necesarias. Antes de retornar la cadena, hay que añadir el carácter nulo en la posición correspondiente.

La codificación de este problema se encuentra en la página Web del libro.

- 9.12.** Para resolver el problema, se sigue el siguiente esquema:

- Invertir las cadenas de caracteres y convertirlas en cadenas de igual longitud, para poder sumar al revés y así poder realizar la suma de izquierda a derecha. Para invertir las cadenas y convertirlas en cadenas de la misma longitud, rellenas con ceros a la derecha, se usan las funciones: *reverse* y *ampliacaracter* de los Problemas 9.10 y 9.11 respectivamente.

487954558	invertido	855459784
235869	invertido y relleno de ceros	000968532

- Sumar las cadenas de caracteres convirtiendo como paso intermedio los caracteres en dígitos y convirtiendo el resultado de nuevo en cadenas.

487954558	855459784
+ 235869	+ 000968532
488190427	724091884

Antes de comenzar la suma de las cadenas, se reserva memoria para el resultado de la suma. Se toman las cadenas carácter a carácter, se convierten en dígitos y se suma el resultado, teniendo cuidado, claro está, que si el resultado es mayor que 9 ha de 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. Es la función *SumarGrandes*, la que realiza el trabajo de sumar los números grandes que recibe como parámetro en sendas cadenas de caracteres, devolviendo el resultado en una cadena de caracteres. Obsérvese que una vez sumados los números se debe invertir de nuevo la cadena de caracteres resultado.

Se incluye la codificación de las funciones *reverse* *ampliacaracter*, así como un programa principal que realiza dos llamadas a la función *sumarGrandes* para comprobar los resultados.

La codificación de este problema se encuentra en la página Web del libro.

- 9.13.** Para multiplicar una cadena que almacena un número o entero por un dígito, se invierte la cadena de caracteres (34567 se convierte en 76543), para posteriormente usar el algoritmo clásico de multiplicación por un dígito, almacenando el resultado en otra cadena de caracteres. Una vez obtenido el resultado, hay que retornar la cadena invertida. Para multiplicar por un dígito, se usa como paso intermedio la conversión de caracteres en dígitos, para volver a reconvertir el resultado de nuevo en caracteres. Es la función `multiplicadigito`, la que multiplica la cadena de caracteres número por el dígito `dig`, retornando el resultado como cadena de caracteres. Se usa la función `reverse` del ejercicio anterior.

La codificación de este problema se encuentra en la página Web del libro.

- 9.14.** El algoritmo clásico de multiplicación realiza la operación de la siguiente forma:

$ \begin{array}{r} 2457 \\ \times 956 \\ \hline 22113 \\ 12285 \\ 14742 \\ \hline 1619163 \end{array} $	$ \begin{array}{r} 2457 \\ \times 956 \\ \hline 22113 \\ 122850 \\ 1474200 \\ \hline 1619163 \end{array} $
--	---

Usando las funciones definidas en los dos problemas anteriores, resulta sencillo realizar la multiplicación, ya que sólo se necesita inicializar el resultado a la cadena vacía, para, posteriormente, ir multiplicando el primer número por cada uno de los dígitos del segundo número en sentido inverso, desplazándolo una posición hacia la izquierda, e ir acumulando el valor del sumando en el acumulador resultado. Para realizar este desplazamiento basta con observar que a la derecha de los números desplazados es muy útil poner un cero para utilizar la función `SumarGrandes`. Este desplazamiento puede realizarse con la función `ampliacaracter`, que pone los ceros necesarios para el desplazamiento. Una vez terminado el proceso, se retorna el valor de `resultado`. En la codificación que se presenta, se incluye un programa principal que realiza una llamada a la función que multiplica los números grandes.

La codificación de este problema se encuentra en la página Web del libro.

- 9.15.** Se lee el texto línea a línea, almacenado en la variable `mayor`, la longitud máxima de las líneas leídas. Una vez terminada la lectura se rellenan de blancos las líneas para convertirlas todas en cadenas de caracteres de la misma longitud, para lo cual se realizan las llamadas correspondientes a la función `ampliacaracter`, pero con el carácter blanco en la llamada. Por último se visualiza el texto leído ya tratado.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 9.1.** Escribir una función para transformar un número entero en una cadena de caracteres formada por los dígitos del número entero.
- 9.2.** Escribir una función para transformar un número real en una cadena de caracteres que sea la representación decimal del número real.
- 9.3.** Escribir una función de C++ que realice la misma tarea que la función `strncat` y con los mismo parámetros. Es decir, concatenar dos cadenas.
- 9.4.** Escribir una función que reciba como parámetro una cadena e indique el número de letras mayúsculas que tiene.

PROBLEMAS PROPUESTOS

- 9.1. 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.
- 9.2. 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().
- 9.3. Un sistema de encriptación 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.
- 9.4. Escribir un programa que lea un texto, y dos palabras claves, clave1 y clave2. El programa debe sustituir todas las apariciones de clave1 por clave2, mostrar el nuevo texto, e indicar el número de intercambios realizados.
- 9.5. 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.
- 9.6. Se trata de encriptar un texto mediante una función de cifrado. El cifrado se realiza cambiando todas las letras minúsculas por otras nuevas letras que recibe en un array cifrado de caracteres; es decir de cambiar 'a' por cifrado[0], 'b' por cifrado[1], etc. Escribir una función que reciba un texto y lo encripte.
- 9.7. Escribir una función que reciba el texto que retorna la función del ejercicio anterior, debidamente encriptado, y lo transforme en el texto original. El array de cifrado es un dato.
- 9.8. Escribir una función que genere un código alfanumérico diferente cada vez que sea llamada devolviéndolo en forma de cadena. El argumento de dicha función es el número de caracteres que va a tener el código generado.
- 9.9. Escribir un programa que tome como entrada un programa escrito en lenguaje C++ de un archivo 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 '*/', pero teniendo presente que no se pueden anidar comentarios.
- 9.10. Escribir una función que reciba una palabra y genere todas las palabras que se pueden construir con sus letras.
- 9.11. Modificar la función anterior para que sólo genere las palabras que comienzan por una consonante.

CAPÍTULO 10

Punteros (apuntadores)

Introducción

El **puntero (apuntador)**, es una herramienta que puede utilizar en sus programas para hacerlos más eficientes y flexibles. Los punteros son, 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 punteros contienen valores que son direcciones de memoria donde se almacenan datos. Utilizando punteros su programa puede realizar muchas tareas que no sería posible utilizando tipos de datos estándar.

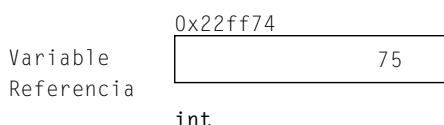
10.1. Concepto de puntero (apuntador)¹

Cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su *nombre*, su *tipo* y su *dirección* en memoria. Al valor, o contenido de una variable se accede por medio de su nombre. A la dirección de la variable se accede por medio del operador de dirección &.

Una *referencia* es un alias de otra variable. Se declara utilizando el operador de referencia (&) que se añade al tipo de la referencia.

EJEMPLO 10.1. *Obtener el valor y la dirección de una variable y una referencia.*

Los dos identificadores Variable y Referencia son nombres diferentes para la misma variable, cuyo contenido y dirección son respectivamente 75 y 0x22ff74.



```
#include <cstdlib>
```

¹ En Latinoamérica es usual emplear el término *apuntador*.

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int Variable= 75;                      // Declaración de variable
    int& Referencia = Variable;             //Referencia e inicialización

    cout << " Contenido de Variable = " << Variable << endl;
    cout << " Direccio &Variable = " << &Variable << endl;
    cout << " Contenido de Referencia = " << Referencia << endl;
    cout << " Direccio &Referencia = " << &Referencia << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Reglas

& se refiere a la dirección en que se almacena el valor. El carácter & tiene diferentes usos en C++:

1. Cuando se utiliza como prefijo de un nombre de una variable, devuelve la dirección de esa variable.
2. Cuando se utiliza como un sufijo de un tipo en una declaración de una variable, declara la variable como sinónimo de la variable que se ha inicializado.
3. Cuando se utiliza como sufijo de un tipo en una declaración de parámetros de una función, declara el parámetro referencia de la variable que se pasa a la función.

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>*

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.
- Asignar un valor a la dirección de memoria.

Existe un segundo método para inicializar un puntero: *asignación dinámica de memoria*. Este método utiliza los operadores new y delete, y se analizará en el siguiente capítulo, aunque el operador new de reserva de memoria se usa ya en este capítulo.

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 10.1 resume los operadores de punteros.

Tabla 10.1. Operadores de punteros

Operador	Propósito
&	Obtiene la dirección de una variable.
*	Define una variable como puntero.
*	Obtiene el contenido de una variable puntero.

Siempre que aparezca un asterisco (*) en una definición de variable, ésta es una variable puntero.

Siempre que aparezca un asterisco (*) delante de una variable puntero se accede a la variable contenido del puntero.

El operador & devuelve la dirección de la variable a la cual se aplica.

C++ requiere que las variables puntero direccionen realmente variables del mismo tipo de dato que está ligado a los punteros en sus declaraciones.

Un puntero puede apuntar a otra variable puntero. Este concepto se utiliza con mucha frecuencia en programas largos y complejos de C++. Para declarar un puntero a un puntero se hace preceder a la variable con dos asteriscos (**). En el siguiente código, ptr5 es un puntero a un puntero.

```
int valor_e = 100;
int *ptr1 = &valor_e;
int **ptr5 = &ptr1;
```

EJEMPLO 10.2. Asignar a una variable puntero una dirección, y a su contenido un valor.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int var;                                // define una variable entera var
    int *pun;                               //define un puntero a un entero pun

    pun = &var;                            //asigna la dirección de var a pun
    *pun = 60;                             // asigna al contenido de pun 60
    cout << " &var. Dirección de var = " << &var << endl;
    cout << " pun. Contenido de pun es la misma dirección de var ";
    cout << pun << endl;
    cout << " var. Contenido de var = " << var << endl;
    cout << " *pun. El contenido de *pun es el mismo que el de var: ";
    cout << *pun << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

NOTA: Son variables punteros aquellas que apuntan a la posición en donde otra/s variable/s de programa se almacenan.

EJEMPLO 10.3. Escriba los números ASCII de los caracteres A, B, C, D, E, F.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char *punteroc;                      // un puntero a una variable carácter
    char caracter;
```

```

punteroc = &caracter;
cout << " ASCII caracter" << endl;
for (caracter = 'A'; caracter <= 'F'; caracter++)
    cout << " " << (int) caracter << " " << *punteroc << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

```

10.2. Punteros *NULL* y *void*

Un *puntero nulo* no apunta a ningún dato válido, se utiliza para proporcionar a un programa un medio de conocer cuando una variable puntero no direcciona a un dato válido. Para declarar un puntero nulo se utiliza la macro `NULL`.

En C++ se puede declarar un puntero de modo que apunte a cualquier tipo de dato, es decir, no se asigna a un tipo de dato específico. El método es: declarar el puntero como un puntero `void`. Un puntero de tipo `void` puede dirigir a cualquier posición en memoria, pero el puntero no está unido a un tipo de dato específico.

NOTA: Un puntero nulo no dirige a ningún dato válido. Un puntero `void` dirige datos de un tipo no especificado. Un puntero `void` se puede igualar a nulo si no se dirige a ningún dato válido. `NULL` es un valor; `void` es un tipo de dato.

EJEMPLO 10.4. Los punteros `void` pueden apuntar a cualquier tipo de dato.

```

int x, *px = &x, &rx = x;
char* c = "Cadena larga";
float *z = NULL;
void *r = px, *s = c, *t = z;

```

`x` es una variable entera; `px` es un puntero a una variable entera inicializada a la dirección de `x`; `rx` es una referencia a un entero inicializada a `x`.

`c` (puntero a carácter) es una cadena de caracteres de longitud 10.

`z` es un puntero a un real inicializado a `NULL`.

`r` es un puntero `void` inicializado a un puntero a entero; `s` es un puntero `void`, inicializado a un puntero a `char`; `t` es un puntero `void` inicializado a un puntero a `float`.

10.3. Punteros y arrays

Los arrays y los punteros están fuertemente relacionados en el lenguaje C++. El nombre de un array es un puntero que contiene la dirección en memoria de comienzo de la secuencia de elementos que forman el array. Este nombre del array es un puntero constante ya que no se puede modificar, sólo se puede acceder para indexar a los elementos del array. Para visualizar, almacenar o calcular un elemento de un array, se puede utilizar notación de subíndices o notación de punteros, ya que a un puntero `p` se le puede sumar un entero `n`, desplazándose el puntero tantos bytes como ocupe el tipo de dato.

Si se tiene la siguiente declaración de array `int V[6] = {1, 11, 21, 31, 41, 51};`, su almacenamiento en memoria será el siguiente:

	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]
memoria	1	11	21	31	41	51
	*V	*(V + 1)	*(V + 2)	*(V + 3)	*(V + 4)	*(V + 5)

EJEMPLO 10.5. Inicialización y visualización de un array con punteros.

El programa inicializa un array de reales y visualiza las direcciones de cada una de las posiciones así como sus contenidos. Las direcciones consecutivas se diferencian en 4 unidades correspondientes a los 4 bytes que ocupa un float.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float V[6];
    for (int j = 0; j < 6; j++)
        *(V+j) = (j + 1) * 10 + 1;
    cout << "    Dirección      Contenido" << endl;
    for (int j= 0; j < 6; j++)
    {
        cout << " V+" << j << " = " << V + j;
        cout << "      V[" << j << "] = " << *(V+j)<< "\n";
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Se puede declarar un *array de punteros*, como un array que contiene punteros como elementos, cada uno de los cuales apuntará a otro dato específico.

EJEMPLO 10.6. Inicialización y visualización de un array de punteros.

El siguiente programa inicializa el array de reales *V*, así como el array de punteros a reales *P*, con las direcciones de las sucesivas posiciones del array *V*. Posteriormente, visualiza las direcciones y los contenidos de *V* usando el array de punteros *P*.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float V[6], *P[6];

    for (int j = 0; j < 6; j++)
    {
        *(V+j) = (5-j) * 10 + 1;
        *(P+j) = V+j; // inicialización de array de punteros
    }
    cout << "    Dirección      Contenido" << endl;
    for (int j = 0; j<6; j++)
    {
        cout << " V+" << j << " = " << *(P+j) << " = *(P+" << j << ")";
        cout << "      V[" << j << "] = " << **(P+j) << "\n";
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

10.4. Punteros de cadenas

Considérese la siguiente declaración de un array de caracteres que contiene las veintisésis 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; // o bien p = &alfabeto[0];
```

Es posible, entonces, considerar dos tipos de definiciones de cadena

```
char cadena1[]="Las estaciones"; //array contiene una cadena
char *pCadena = "del año son:"; //puntero a cadena
```

También es posible declarar un array de cadenas de caracteres:

```
char* Estaciones[4]={"Primavera", "Verano", "Otonyo", "Invierno"};
// array de punteros a cadena
```

10.5. 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 `--`. Esta operación hace que el operador contenga la dirección del siguiente, o anterior elemento. Se pueden sumar o restar una constante puntero a o desde un puntero y sumar o restar un entero. Sin embargo, no tiene sentido 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.

EJEMPLO 10.7. Modificación de una cadena con un puntero.

El programa lee una cadena de caracteres, y mediante una variable puntero, inicializada a la primera posición del array de caracteres, se van cambiando las letras mayúsculas por minúsculas y recíprocamente. El bucle `while` itera hasta que se llegue al final de la cadena de caracteres. La sentencia `*puntero++ = *puntero - 32`. Asigna al contenido del puntero el contenido del puntero menos el número ASCII 32 para que el carácter pase a letra minúscula. Posteriormente, el puntero avanza una posición (un byte por ser de tipo `char`).

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char *puntero;
    char Cadena[81];

    cout << "Introduzca cadena a convertir:\n\n";
    cin.getline(Cadena, 80);
    puntero = Cadena; // puntero apunta al primer carácter de la cadena
```

```

while (*puntero) // mientras puntero no apunte a \0
    if ((*puntero >= 'A') && (*puntero <= 'Z'))
        *puntero++ = *puntero+32; // sumar 32, para convertir en minúscula
    else if ((*puntero >= 'a') && (*puntero <= 'z'))
        *puntero++ = *puntero-32; // restar 32, para convertir en mayúscula
    else
        puntero++;

cout << "La cadena convertida es:\n" << endl;
cout << Cadena << endl;
system("PAUSE");
return EXIT_SUCCESS;
}

```

Un puntero constante es un puntero que no se puede cambiar , pero que los datos apuntados por el puntero pueden ser cambiados. Para crear un puntero que no pueda ser modificado o *puntero constante* se debe utilizar el siguiente formato:

```
<tipo de dato> *const <nombre puntero> = <dirección de variable>;
```

No puede cambiarse el valor del puntero, pero puede cambiarse el contenido almacenado en la posición de memoria a donde apunta.

Un puntero a una constante se puede modificar para apuntar a una constante diferente, pero los datos apuntados por el puntero no se pueden cambiar. 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, pero puede cambiarse el valor del puntero.

Nota: Una definición de un puntero constante tiene la palabra reservada `const` delante del nombre del puntero, mientras que el puntero a una def inición 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 sabe que un puntero siempre apuntará a la misma posición y nunca necesita ser reubicado (recolocado), defíalo como un puntero constante.
- Si sabe que el dato apuntado por el puntero nunca necesitará cambiar , defina el puntero como un puntero a una constante.

EJEMPLO 10.8. Puntero constante y puntero a constante.

Muestra las operaciones válidas y no válidas de un puntero constante `puntero1`, y un puntero a constante `puntero2`.

```

int x, y;
const int z = 25; // constante entera
const int t = 50; // constante entera
int *const puntero1 = &x; //puntero1 es un puntero constante
const int *puntero2 = &z; // puntero2 es un puntero a constante

```

```

*puntero1 = y;           // sentencia válida puede cambiarse su contenido
p1 = &y;                 //sentencia ilegal no puede cambiarse el puntero
puntero2 = &t;            // sentencia válida puede cambiarse puntero2
*puntero2 = 15;          // sentencia ilegal no puede cambiarse su contenido

```

Punteros en los array de dos dimensiones. Para apuntar a un 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 f filas del array bidimensional o matriz. Si `a` se ha definido como un array bidimensional, el nombre del array `a` es *un puntero constante* que apunta a la primera fila `a[0]`. El puntero `a+1` apunta a la segunda fila `a[1]`, etc. A su vez `a[0]` es un puntero que apunta al primer elemento de la fila 0 que es `a[0][0]`. El puntero `a[1]` es un puntero que apunta al primer elemento de la fila 1 que es `a[1][0]`, etc.

EJEMPLO 10.9. Array bidimensional, punteros y posiciones de memoria.

Dada la declaración `float A[5][3]` que define un array bidimensional de cinco filas y tres columnas, se tiene la siguiente estructura:

Puntero a puntero fila	Puntero a fila	ARRAY BIDIMENSIONAL float A[4][3]		
A →	A[0] →	A[0][0]	A[0][1]	A[0][2]
A+1 →	A[1] →	A[1][0]	A[1][1]	A[1][2]
A+2 →	A[2] →	A[2][0]	A[2][1]	A[2][2]
A+3 →	A[3] →	A[3][0]	A[3][1]	A[3][2]
A+4 →	A[5] →	A[4][0]	A[4][1]	A[4][2]

`A` es un puntero que apunta a un array de 5 punteros `A[0]`, `A[1]`, `A[2]`, `A[3]`, `A[4]`.

`A[0]` es un puntero que apunta a un array de tres elementos `A[0][0]`, `A[0][1]`, `A[0][2]`.

`A[1]` es un puntero que apunta a un array de tres elementos `A[1][0]`, `A[1][1]`, `A[1][2]`.

`A[2]` es un puntero que apunta a un array de tres elementos `A[2][0]`, `A[2][1]`, `A[2][2]`.

`A[3]` es un puntero que apunta a un array de tres elementos `A[3][0]`, `A[3][1]`, `A[3][2]`.

`A[4]` es un puntero que apunta a un array de tres elementos `A[4][0]`, `A[4][1]`, `A[4][2]`.

`A[i][j]` es equivalente a las siguientes expresiones:

- `*(A[i]+j)` el contenido del puntero a la fila `i` más el número de columna.
- `*((A+i))+j)`. Si se cambia `A[i]` por `*(A+i)` se tiene la siguiente expresión anterior.
- `*(&A[0][0]+ 3*i+j)`.

`A` es un puntero que apunta a `A[0]`.

`A[0]` es un puntero que apunta a `A[0][0]`.

Si `A[0][0]` se encuentra en la dirección de memoria 100 y teniendo en cuenta que un `float` ocupa 4 bytes, la siguiente tabla muestra un esquema de la memoria:

Contenido de puntero a puntero fila	Contenido de puntero a fila	Direcciones del array bidimensional float A[4][3]		
<code>*A = A[0]</code>	<code>A[0] = 100</code>	<code>&A[0][0] = 100</code>	<code>&A[0][1] = 104</code>	<code>&A[0][2] = 108</code>
<code>*(A+1) = A[1]</code>	<code>A[1] = 112</code>	<code>&A[1][0] = 112</code>	<code>&A[1][1] = 116</code>	<code>&A[1][2] = 120</code>
<code>*(A+2) = A[2]</code>	<code>A[2] = 124</code>	<code>&A[2][0] = 124</code>	<code>&A[2][1] = 128</code>	<code>&A[2][2] = 132</code>
<code>*(A+3) = A[3]</code>	<code>A[3] = 136</code>	<code>&A[3][0] = 136</code>	<code>&A[3][1] = 140</code>	<code>&A[3][2] = 144</code>
<code>*(A+4) = A[4]</code>	<code>A[5] = 148</code>	<code>&A[4][0] = 148</code>	<code>&A[4][1] = 152</code>	<code>&A[4][2] = 156</code>

EJEMPLO 10.10. *Matriz de reales y vector de punteros a reales. Diferencias que pueden encontrarse entre las declaraciones: float A[10][10]; float *V[10];. ¿Pueden realizarse las siguientes asignaciones?: A = V; V[1] = A[1]; V[2] = &A[2][0];.*

A es una matriz de reales y V es un vector de punteros a reales, por lo que las declaraciones son esencialmente diferentes.

En cuanto a la primera asignación A = V, ambas variables A y V son punteros que apuntan a punteros a reales (aunque ligeramente diferente) por lo que en principio podría realizarse la asignación, pero debido a que ambas variables son punteros constantes (al ser nombres de variables de arrays), no puede realizarse, ya que no puede modificarse su contenido.

En la segunda asignación V[1] = A[1], el compilador interpreta la expresión A[1] como conteniendo la dirección de la segunda fila de la matriz, que es un puntero a un vector de 10 reales, por lo que es un puntero a reales. Por otro lado, para el compilador V[1], es un puntero a reales y, por tanto, su valor es del mismo tipo que A[1]. Todo lo anteriormente indicado permite realizar la asignación indicada.

La expresión de la derecha &A[2][0], de la tercera asignación V[2] = &A[2][0], proporciona la dirección del primer elemento de la tercera fila de la matriz, por consiguiente es también de tipo puntero a real al igual que el lado izquierdo de la asignación V[2], por lo que la asignación es correcta.

EJEMPLO 10.11. *Direcciones ocupadas por punteros asociados a una matriz.*

El programa muestra las direcciones ocupadas por todos los elementos de una matriz de reales dobles de 5 filas y 4 columnas, así como las direcciones de los primeros elementos de cada una de las filas, accedidos por un puntero a fila. Observe que la salida se produce en hexadecimal. La dirección de un elemento de la matriz se obtiene del anterior sumándole 8 en hexadecimal.

```
include <cstdlib>
#include <iostream>
using namespace std;
#define N 5
#define M 4
double A[N][M];

int main(int argc, char *argv[])
{
    int i,j;
    cout << " direcciones de todos los elementos de la matriz\n\n";
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < M; j++)
            cout << " &A[" << i << "][" << j << "]=" << &A[i][j];
        cout << "\n";
    }
    cout << " direcciones de comienzo de las filas de la matriz\n\n";
    for (i = 0; i < N; i++)
        cout << " A[" << i << "] = " << A[i]
            << " contiene dirección de &A[" << i << "][0]" << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJEMPLO 10.12. *Lectura y escritura de matrices mediante punteros. Escribir un programa que lea y escriba matrices genéricas mediante punteros y funciones.*

Para poder tratar la lectura y escritura de matrices mediante funciones que reciban punteros como parámetros, basta con trasmitir un puntero a puntero. Además, se debe informar a cada una de las funciones el número de columnas y fi-

las que tiene (aunque sólo es necesario el número de columnas), por lo que las funciones pueden ser declaradas de la siguiente forma: `void escribir_matriz(int ** A, int f, int c)` y `void leer_matriz(int ** A, int f, int c)` donde `f` y `c` son respectivamente el número de filas y el número de columnas de la matriz. Para tratar posteriormente la lectura y escritura de datos en cada una de las funciones hay que usar `*(*A + c * i + j)`. Las llamadas a ambas funciones, deben ser con un tipo de dato compatible tal y como se hace en el programa principal. En el programa, además se declaran el número de filas `F`, y el número de columnas `C` como macros constantes.

```
#include <cstdlib>
#include <iostream>
#define F 3
#define C 2
using namespace std;

int A[F][C];

void escribir_matriz(int ** A, int f, int c)
{
    int i, j;

    for (i = 0; i < f; i++)
    {
        for(j = 0; j < c ; j++)
            cout << " " << *(*A + c*i+j);
        cout << endl;
    }
}

void leer_matriz(int** A, int f, int c)
{
    int i, j;

    for (i = 0; i < f; i++)
        for(j = 0; j < c; j++)
            cin >> *(*A + c*i+j);
}

int main(int argc, char *argv[])
{
    int * a = &A[0][0];
    leer_matriz(&a,F,C);
    escribir_matriz(&a,F,C);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

10.6. Punteros como argumentos de funciones

Cuando se pasa una variable a una función (*paso por valor*) no se puede cambiar el contenido de esa variable. Sin embargo, si se pasa un puntero a una función (*paso por dirección*) se puede cambiar el contenido 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. Cuando se pasa un elemento a una función, sin embargo, el elemento se pasa por valor.

Los parámetros dirección son más comunes en C, dado que en C++ existen los parámetros por referencia que resuelven mejor la modificación de los parámetros dentro de funciones.

EJEMPLO 10.13. *Paso por referencia en C y en C++.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

struct datos
{
    float mayor, menor;
};

void leer_registrodatosCmasmas(datos &t) // parámetro por referencia en C++
{
    float actual;
    cin >> actual;
    if (actual > t.mayor)
        t.mayor = actual;
    else if (actual < t.menor)
        t.menor = actual;
}

void leer_registrodatosC(datos *t) // parámetro dirección C, referencia
{
    float dat;
    cin >> dat;
    if (dat > t->mayor)
        t->mayor = dat;
    else if (dat < t->menor)
        t->menor = dat;
}

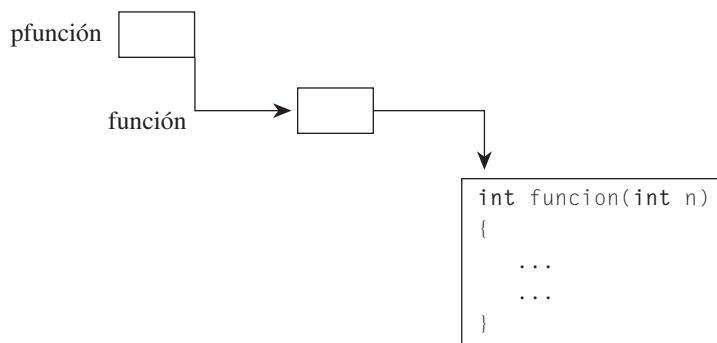
int main(int argc, char *argv[])
{   datos dat;
    ....
    leer_registrodatosCmasmas(dat);      // llamada por referencia en C++
    leer_registrodatosC(&dat);          // llamada por dirección en C
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

10.7. Punteros a funciones

Es posible crear punteros que apunten a funciones. Estos punteros de funciones apuntan a código ejecutable, en lugar de dirigir datos. 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. Mediante un puntero cuyo valor sea igual a la dirección inicial de una función se puede llamar a una función de una forma indirecta. 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 de vuelve el tipo *Tipo_de_retorno* y tiene una lista de parámetros.



La función asignada a un puntero a función 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. La sintaxis general de inicialización de un puntero a función y el de llamada son respectivamente:

```
PunteroFuncion = una_funcion
PunteroFuncion(lista de parametros);
```

Recuerde

- func, nombre de un elemento.
- func[] es un array.
- (*func[]) es un array de punteros.
- (*func[])() es un array de punteros a funciones.
- int (*func[])() es un array de punteros a funciones que devuelven valores int.

EJEMPLO 10.13. *Array de punteros a funciones. Se quiere evaluar las funciones $f_1(x)$, $f_2(x)$ y $f_3(x)$ para todos los valores de x en el intervalo $0.3 \leq x \leq 4.1$ con incremento de 0.5. Escribir un programa que evalúe dichas funciones utilizando un array de punteros a función. Las funciones son las siguientes:*

$$f_1(x) = 3\sin(x) + 2.5\cos(x); f_2(x) = -x\sin(x) + x^2; f_3(x) = x^2 - x + 1$$

Se trata, como se hace en la codificación, de definir las tres funciones f_1 , f_2 y f_3 al estilo de C++ y después definir un array de punteros a las funciones, a los que se asignan cada una de las funciones previamente definidas. El acceso a dichas funciones para su ejecución es similar al acceso que se realiza con cualquier otro tipo de dato cuando se accede con punteros.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#define maxf 3
#define minx 0.3
#define maxx 4.1
#define incremento 0.5
using namespace std;

float f1 (float x)
{
    return (3 * sin(x)+ 2.5*cos(x));
}

float f2(float x)
```

```

{
    return (-x * sin (x) + x*x);
}

float f3(float x)
{
    return ( x * x - x + 1);
}

int main(int argc, char *argv[])
{
    float (*Array_de_Funciones[maxf]) (float);
    // array de punteros a funciones que retornan reales

    Array_de_Funciones [0] = f1;
    Array_de_Funciones [1] = f2;
    Array_de_Funciones [2] = f3;

    for (int i = 0; i < maxf; i++)
    {
        cout << " funcion " << i +1 << endl;
        for ( float x = minx; x < maxx; x += incremento)
            cout << " x = "<<x<<" f = " << Array_de_Funciones [i](x) << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

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.

EJEMPLO 10.14. *Paso de funciones como parámetros a otras funciones. Cálculo de la suma de los n primeros términos de una serie genérica.*

Se trata de sumar los n primeros términos de una serie genérica $suma = \sum_{i=1}^n t_i$. La función `sumaterminos`, recibe como parámetro el número de términos n a sumar y una función `fun` que calcula el término genérico. Para probar la función se usan los términos de dos series

$$serie1 = \sum_{i=1}^n \frac{3}{i * i} \quad serie2 = \sum_{i=1}^n \frac{1}{i}$$

Las funciones `terminoserie1` y `terminoserie2`, calculan los términos de cada una de las series.

```

#include <cstdlib>
#include <iostream>
using namespace std;

double Sumaterminos(int n, double (*fun) (int ))
{
    double acu = 0;

    for (int i = 1; i <= n; i++)
        acu += fun(i);
    return acu;
}

```

```

double terminoseriel(int i)
{
    return (double)3.0 / (i * i);
}

double terminoserie2(int i)
{
    return (double)1.0 / i;
}

int main(int argc, char *argv[])
{
    cout << "Suma de cinco elemento de serie1: "
        << Sumaterminos(5, terminoseriel) << endl;
    cout << "Suma de cuatro terminos de serie2: "
        << Sumaterminos(3, terminoserie2) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

10.8. Punteros a estructuras

Se puede declarar un puntero a una estructura tal como se declara un puntero a cualquier otro objeto. Cuando se referencia un miembro de una estructura usando el nombre de la estructura, se emplea el operador punto (.). En cambio, cuando se referencia una estructura utilizando el puntero estructura, se emplea el operador flecha (->) para acceder a un miembro de ella.

EJEMPLO 10.15. *Acceso a miembros de estructuras. Dadas las siguientes declaraciones de estructuras, se trata de escribir cómo acceder a los atributos dat, dia y mes de la variable estructura est de tipo datos. ¿Qué problemas habría en la siguiente sentencia? cin.getline(est.nombre, 40).*

```

struct fechas
{
    int dia, mes, anyo;
    float dat;
};

struct datos
{
    char* nombre;
    fechas * fec;
} est;

```

La variable `est` es una variable de tipo estructura, y por esta razón se usa el operador punto para acceder al miembro `fec`. Desde el dato miembro `fec` es necesario usar el operador flecha `->` para acceder a los campos `dat`, `día`, `mes` y `anyo`, al ser un puntero a la estructura `fechas` a la que apunta.

```

est.fec->dat;
est.fec->dia;
est.fec->mes;
est.fec->anyo;

```

El campo `nombre` de la estructura `fechas` es un puntero a `char`, por ello si no se reserva memoria previamente, no apunta a ningún sitio, y dará problemas cuando la función `cin.getline(est.nombre, 40)` intente colocar el resultado en el pun-

tero que se le pasa como argumento. Para evitar esto sería necesario reservar memoria antes de llamar a `cin.getline(est.nombre, 40)`, de la siguiente forma `este.nombre = new char[41];`

EJEMPLO 10.16. *Lectura de una estructura con parámetro de tipo puntero a estructura, y visualización con parámetro de tipo estructura.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

struct articulo
{
    char nombre[81];
    int codigo;
};

void leerarticulo( articulo * particulo)
{
    cout << "nombre del articulo :";
    cin.getline(particulo->nombre, 80);
    cout << "Codigo del articulo :";
    cin >> particulo->codigo;
}

void mostrararticulo ( articulo articul)
{
    cout << "nombre del articulo :" << articul.nombre << endl;
    cout << "Codigo asociado al articulo : " << articul.codigo << endl;
}

int main(int argc, char *argv[])
{
    articulo a;
    leerarticulo(&a);
    mostrararticulo(a);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJERCICIOS

10.1. *Encontrar los errores de las siguientes declaraciones de punteros:*

```
int x, *p, &y;
char* b= “Cadena larga”;
char* c= ‘C’;
float x;
void* r = &x;
```

10.2. *¿Qué diferencias se pueden encontrar entre un puntero a constante y una constante puntero?*

10.3. Un array unidimensional se puede indexar con la aritmética de punteros. ¿Qué tipo de puntero habría que definir para indexar un array bidimensional?

10.4. El código siguiente 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++)
        cout << mt[f][c];
    cout << "\n";
}
```

10.5. Escribir una función con un argumento de tipo puntero a double y otro argumento de tipo int. El primer argumento se debe de 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.

10.6. Dada las siguientes definiciones y la función gorta:

```
double W[15], x, z;
void *r;

double* gorta(double* v, int m, double k)
{
    int j;
    for (j = 0; j < m; j++)
        if (*v == k)
            return v;
    return 0,
}
```

- ¿Hay errores en la codificación? ¿De qué tipo?
- ¿Es correcta la siguiente llamada a la función?:

r = gorta(W,10,12.3);

- ¿Y estas otras llamadas?:

```
cout << (*gorta(W,15,10.5));
z = gorta(w,15,12.3);
```

PROBLEMAS

- 10.1. Se quiere almacenar la siguiente información de una persona: nombre, edad, altura y peso. Escribir una función que lea los datos de una persona, recibiendo como parámetro un puntero y otra función que los visualice.
- 10.2. Escribir un programa que decida si una matriz de números reales es simétrica. Utilizar: 1, una función de tipo `bool` que reciba como entrada una matriz de reales, así como el número de filas y de columnas, y decida si la matriz es simétrica; 2, otra función que genere la matriz de 10 filas y 10 columnas de números aleatorios de 1 a 100; 3, un programa principal que realice llamadas a las dos funciones. **Nota:** usar la transmisión de matrices como parámetros punteros y la aritmética de punteros para la codificación.
- 10.3. Escribir un programa para generar una matriz de 4 x 5 números reales, y multiplique la primera columna por otra cualquiera de la matriz y mostrar la suma de los productos. El programa debe descomponerse en subprogramas y utilizar punteros para acceder a los elementos de la matriz.
- 10.4. Codificar funciones para sumar a una fila de una matriz otra fila; intercambiar dos filas de una matriz, y sumar a una fila una combinación lineal de otras. **Nota:** debe realizarse con punteros y aritmética de punteros.
- 10.5. Codificar funciones que realicen las siguientes operaciones: multiplicar una matriz por un número, y llenar de ceros una matriz. **Nota:** usar la aritmética de punteros.
- 10.6. Escribir un programa en el que se lea 20 líneas de texto, cada línea con un máximo de 80 caracteres. Mostrar por pantalla el número de vocales que tiene cada línea, el texto leído, y el número total de vocales leídas.
- 10.7. 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 típica respecto al tiempo.
- 10.8. Escribir un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapecio y un círculo. **Nota:** utilizar un array de punteros de funciones, siendo las funciones las que permiten calcular el área.
- 10.9. 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 la 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.
- 10.10. 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).
- 10.11. Escribir un programa para simular una pequeña calculadora que permita leer expresiones enteras terminadas en el símbolo = y las evalúe de izquierda a derecha sin tener en cuenta la prioridad de los operadores. Ejemplo 4*5-8=12. **Nota:** usar un array de funciones para realizar las distintas operaciones (suma, resta, producto, cociente resto).

SOLUCIÓN DE LOS EJERCICIOS

- 10.1. Es incorrecta sintácticamente la declaración `int &y`, ya que es una referencia y debe ser inicializada a una variable. (Una referencia es un alias de otra variable.) No tiene ningún sentido en C++, debería ser, por ejemplo, `int &y = x`. Un carácter que está entre 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"`. La declaración `void* r = &x;` es correcta.

- 10.2.** Por medio de un puntero a constante se puede acceder a la constante apuntada, pero no está permitido cambiar o modificar su valor por medio del puntero. Un puntero declarado como constante no puede modificar el valor del puntero, es decir, la dirección que contiene y a la que apunta; pero puede modificarse el contenido almacenado en la posición de memoria a la que apunta.
- 10.3.** El tipo del puntero que se utilice para recorrer un array unidimensional tiene que ser del 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. El acceso y recorrido puede realizarse de la siguiente forma:

```
int *p, v[5];
p = v;
for (int i = 0; i < 5; i++, p++) // p apunta a v[i]
```

El compilador de C++ considera que un array bidimensional es en realidad un array de punteros a los arrays que forman sus filas, por lo que para apuntar al inicio de un array bidimensional es necesario un puntero o doble o puntero a puntero, que contiene la dirección del primer puntero del array de punteros a cada una de las filas del array bidimensional o matriz. Como los arrays bidimensionales se almacenan en memoria linealmente fila a fila, para acceder a un elemento concreto de una fila y columna determinadas hay que calcular primer o en que fila está y dentro de esa fila en qué columna para poder calcular 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 (conocido el tipo de dato). Por ejemplo dada la declaración:

int v[10][5]; son expresiones equivalentes:

```
v[i][j];
*(v[i] + j); // puntero de filas
*((v + i) + j) // puntero de columnas
(&v[0][0] + Numero_de_columnas * i + j)
```

- 10.4.** Se define un puntero a puntero que apunte a la primera posición de la matriz y se calculan las posiciones de memoria donde se encuentran 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 i , habrá que saltar i filas enteras de elementos del tipo de la matriz, para situarse al comienzo de su fila en la memoria. Si el elemento que se busca está en la columna j , hay que avanzar tres posiciones desde el comienzo de su fila calculado anteriormente. Así, se accede a la dirección donde se encuentra a el elemento buscado. $V[i][j]$ está $*(&V[0][0] + i * (numero_de_columnas) + j)$ (Para realizar los cálculos es imprescindible conocer el número de columnas de la matriz, que es igual al tamaño de cada fila). El elemento buscado también se encuentra en $*(V[i]+j)$, o bien en $*((*(mt+f))+c)$. Una de estas expresiones (normalmente la primera) es la misma que sustituye el compilador de C++, cuando compila la indirección que representan los operadores corchetes de los arrays. En la codificación que se presenta, se incluyen cada una de las expresiones anteriores para visualizar la matriz.

```
#include <cstdlib>
#include <iostream>
using namespace std;
#define N 2
#define M 3
double mt[N][M];

int main(int argc, char *argv[])
{
    int f, c;
```

```

for (f = 0; f < N; f++)
    for (c = 0; c < M; c++)
        cin >> mt[f][c];

cout << " mt[f][c]\n";
for (f = 0; f < N; f++)
{
    for (c = 0; c < M; c++)
        cout << mt[f][c] << " ";
    cout << endl;
}

cout << " * ( ( * (mt + f) ) + c)\n";
for (f = 0; f < N; f++)
{
    for (c = 0; c < M; c++)
        cout << (* ( ( * (mt + f) ) + c)) << " ";
    cout << endl;
}

cout << " * (mt[f]+c)\n";
for (f = 0; f < N; f++)
{
    for (c = 0; c < M; c++)
        cout << ( * (mt[f] + c)) << " ";
    cout << endl;
}

cout << " * (&mt[0][0] + M * f + c)\n";
for (f = 0; f < N; f++)
{
    for (c = 0; c < M; c++)
        cout << (* (&mt[0][0] + M * f + c)) << " ";
    cout << endl;
}
system("PAUSE");
return EXIT_SUCCESS;
}

```

- 10.5.** Al hacer la llamada a la función se debe pasar el nombre del array que se va a recorrer o un puntero con la dirección del comienzo del array. El puntero que recibe la función `double *A` puede ser tratado dentro de su código como si fuer a un array, usando el operador corchete `[]`, `A[i]`, o como un simple puntero que se va a mover por la memoria. En este caso se codifica con el operador corchete. Para resolver el problema planteado se usa un algoritmo voraz, que toma como `minimo` el primer elemento `A[0]`, y como dirección del mínimo `dirminimo` la dirección del primer elemento del array `&A[0]`. Usando el segundo parámetro de la función `int n`, que es necesario para no moverse fuera de la región de la memoria donde están los datos válidos del array original almacenados, se itera con un bucle `for`, eligiendo como nuevo `minimo` el que ya se tenía o el que se está tratando en caso de que lo sea.

```

double *minimo_de_Array (double *A, int n)
{
    int i;

```

```

double minimo = A[0];
double *dirminimo = & A[0];

for (i = 0; i < n; i++)
    if (A[i] < minimo)
    {
        dirminimo = &A[i];
        minimo = A[i];
    }
return dirminimo;
}

```

- 10.6.** La función tiene, en primer lugar, un error de sintaxis en la sentencia `return`, ya que debe terminar necesariamente en punto y coma (`:`) y no en coma (`,`) como así lo hace. Por otra parte, se observa que el bucle `for` itera pero siempre compara el mismo valor `*v == k`, y parece que hay otro error semántico y sintáctico, ya que es poco probable que se programe un bucle para comparar un solo elemento. Tal vez, lo que el programador, haya querido programar, sea una función que retorne la dirección de la primera ocurrencia de `k` en el vector `v`, por ello la codificación debería ser la siguiente:

```

double* gorta(double* v, int m, double k)
{
    int j;

    for (j = 0; j < m; j++)
        if (*v == k)
            return v;
        else
            v++;
    return 0; // si no se encuentra retorna el valor de 0
}

```

- La primera llamada `r = gorta(W,10,12.3);` es correcta ya que a un puntero `void` se le puede asignar cualquier otro tipo de punteros, y el valor de `m` es 12 que es menor o igual que 15 que es el tamaño declarado del array.
- La segunda llamada `cout << (*gorta(W,15,10.5)` es sintácticamente correcta. La tercera llamada `z = gorta(W,15,12.3)` es incorrecta ya que a una variable de tipo `double` no puede asignársele un puntero a un tipo `double`.

SOLUCIÓN DE LOS PROBLEMAS

- 10.1.** Se declara la estructura `Persona` con los atributos indicados, y se programa las dos funciones pedidas, ambas con parámetros tipo puntero a `Persona`, con lo que el acceso a los miembros debe hacerse con `->`. Además, se programa una llamada a la función `mostrar_Persona` usando un `Array_de_Personas`, previamente inicializado con la información de dos personas, realizando las llamadas con un puntero `p` que toma antes de comenzar el bucle el valor de `p = Array_de_Personas`.

La codificación de este problema se encuentra en la página Web del libro.

- 10.2.** La función `simetrica` decide si una matriz cuadrada que recibe como parámetro es simétrica, por lo que es de tipo `bool`. Esta función se codifica con tres parámetros `bool simetrica (float **mat, int f, int c)`. El primero es un puntero a puntero de reales, `m` en el que se recibe la matriz, el segundo, `f`, y el tercero, `c`, son respectivamente el número de filas

y de columnas, que en el caso de una matriz cuadrada debe coincidir. Es bien conocido que una matriz es simétrica si cada uno de sus elementos es igual a su simétrico con respecto a la diagonal principal. Para resolver el problema de decidir si una matriz es simétrica, se define una variable de tipo bool `sime`, inicializada a `true`. Esta variable se pone a `false` cuando se está seguro de que la matriz no cumple la condición de simetría. El recorrido de la matriz se hace mediante dos bucles `for` anidados que avanzan por las posibles filas y columnas, siempre que la condición de simetría siga cumpliéndose (se sale de cualquiera de los dos bucles si `sime` toma el valor de `false`). Si en una de las iteraciones se comprueba que no se cumple la condición “cada elemento es idéntico al que se obtiene cambiando su fila por su columna” la variable, o interruptor `sime` cambia de valor poniéndose a `false`.

La función `crearMatriz`, genera aleatoriamente los datos de la matriz cuadrada de `f` filas y `c` columnas. Para hacerlo usa las funciones `randomize` y `random` definidas en sendos macros, y que se encargan de generar la semilla y los sucesivos números aleatorios en el rango de 1 y `max = 100`.

El programa principal realiza las llamadas a las funciones explicadas anteriormente, para mostrar cómo debe realizarse una invocación de las mismas. Se hace al propio tiempo una llamada a la función `escribir_matriz`, cuyo prototipo es incluido en el código, y cuya codificación, puede consultarse en el Ejemplo 10.11. A nivel global se referencian en sentencias `define` el número de filas, el de columnas y el valor aleatorio máximo que se puede generar en los correspondientes identificadores `F`, `C`, `max`.

La codificación de este problema se encuentra en la página Web del libro.

- 10.3.** La función `multiplicacolumna` recibe como parámetro en forma de puntero a puntero de reales la matriz `m`, el número de filas `el` de columnas, así como la columna `ncol` que se multiplica por la columna `0`, un vector de punteros `vp` como puntero a reales, donde se retorna el resultado. La codificación usa un bucle `for` que recorre las sucesivas filas de la matriz, para ir almacenando los resultados de los productos parciales de la columna `0` por la columna `ncol` en las posiciones correspondientes del vector `vp`.

La función `muestravector`, se encarga de mostrar el vector `v` que recibe como parámetro (puntero a reales).

La función `muestra_suma` realiza la suma de las correspondientes componentes del vector que recibe como parámetro visualizando el resultado.

El programa principal realiza las sucesivas llamadas a las funciones, usando, además, las funciones `escribir_matriz` y `crearMatriz` cuyos prototipos son incluidos, y cuya codificación puede consultarse en el Problema 10.2.

La codificación de este problema se encuentra en la página Web del libro.

- 10.4.** La función `Intercambiafilas`, recibe los números de las filas que debe intercambiar, `nf1`, `nf2`, y mediante un bucle itera por cada una de las columnas intercambiando los elementos uno a uno. La función `Sumafilas` es la encargada de sumar a la fila `nf1`, la fila `nf2`, realizándolo mediante su correspondiente bucle de columnas controlado por la variable `j`. Para sumar a una fila `nf1`, una combinación lineal de filas, la función `Sumacombinacion`, recibe los coeficientes de la combinación en el vector `vf` (puntero a reales) y, posteriormente, por cada elemento `j` de la fila `nf1`, se la suma la combinación lineal

$$m(nf1, j) = \sum_{i=0}^{f-1} m(i, j) * vf(i)$$

La codificación de este problema se encuentra en la página Web del libro.

- 10.5.** La codificación de ambas funciones es casi trivial. Basta con recorrer todos los elementos de la matriz por `f` filas y `c` columnas, efectuando los cálculos correspondientes

La codificación de este problema se encuentra en la página Web del libro.

- 10.6.** Para almacenar el texto de entrada se declara un sinónimo de `char*` que es `linea`, para que al declarar `entrada` como un puntero a `linea`, se tenga un puntero que apunta a un puntero de caracteres. Ésta es una forma de declarar una matriz bidimensional de caracteres, de tal manera que cada una de las `f` filas contenga una línea de la entrada. Al no declarar una matriz de caracteres como tal en el programa, es necesario disponer de memoria para la matriz, reservando primeramente memoria para un vector de punteros a `char` con la orden `entrada = new linea [20]` y, posteriormente, reservando

memoria para cada una de las líneas mediante la orden `entrada[i] = new char[strlen(cad)+1]`. Esta variable `cad`, es una cadena de caracteres (puntero a carácter) a la que se le reserva memoria para almacenar una línea completa de 80 caracteres. Ahora bien, cuando se reserva memoria a `entrada[i]`, sólo se hace con la longitud real que tenga la línea leída, sin desperdiciar espacio de memoria sin ser usada como ocurriría si hubiera definido una matriz con todas las filas de la misma longitud. El programa que se presenta, lee las líneas del texto en un bucle externo controlado por la variable `i`, almacenándolas en el array bidimensional `entrada`, para, posteriormente, en otro bucle interno controlado por la variable `j`, ir tratando cada uno de los caracteres de la línea, para acumular las vocales y poder visualizar el resultado del número de vocales leídas al final del bucle interno. Una vez terminada la lectura de datos, se presenta el texto leído, así como el número total de vocales que tiene el texto.

La codificación de este problema se encuentra en la página Web del libro.

- 10.7.** El primer paso consiste en definir una estructura `tiempos` para poder almacenar los minutos y segundos. El segundo paso consiste en definir la estructura `regnadador` para almacenar los datos de cada nadador. Como la estructura `regnadador` tiene miembros que son cadenas, se declaran punteros a carácter, para reservar sólo la memoria que se necesite, una vez que se conozca la longitud de la cadena que se introduce en el campo. Al tener que almacenar un máximo de 16 nadadores se define un array `VectorNadadores` que almacena la información de cada uno de los nadadores, como punteros a estructuras de `regnadador` por lo que al principio del programa se crea un array de punteros sin inicializar, lo que ocupa poco espacio en memoria. Se gún se lea cada uno de los nadadores se reservará espacio en memoria ajustado a la información que se almacena. El ahorro en espacio de memoria es significativo cuando el número de registros, como en este caso, es grande. La función `leerregnadador` se encarga de la reserva de memoria para un registro de un nadador, y leer los datos introducidos por el usuario. Recibe como parámetro por referencia un puntero `reg` a `regnadador`. Para visualizar los datos de un nadador se emplea la función `escribirregnadador` que recibe como parámetro un puntero a una estructura `regnadador`. Por último, la función `DesviaciónMedia`, calcula la desviación media en segundos del array de nadadores, para lo que necesita una función `MediaVector`, encargada de calcular la media en segundos de los nadadores. Las expresiones matemáticas de estas dos funciones son:

$$m = \left(\sum_{i=0}^{n-1} a(i) \right) / n \quad dm = \frac{\sum_{i=0}^{n-1} abs(a(i) - m)}{n}$$

La codificación de este problema se encuentra en la página Web del libro.

- 10.8.** Se programan en primer lugar las funciones que calculan el área de cada una de las figuras de acuerdo con las correspondientes fórmulas:

$$\begin{aligned} Tri\ Rec &= (c_1 * c_2) / 2 & TriIsos &= l1 * \sqrt{\frac{l1^2 - \left(\frac{l2}{2}\right)^2}{2}} & Cuad &= l^2 & Circ &= p * r^2 \end{aligned}$$

El array `areafigura[]` contiene los nombres de cada una de las funciones. Este array de nombres de las funciones es llamado en el programa principal, dentro de un menú de opciones.

La codificación de este problema se encuentra en la página Web del libro.

- 10.9.** El programa se ha estructurado de la siguiente forma: La función `leer_registro`, solicita al usuario todos los datos de un registro, cuyos atributos han sido declarados en la estructura `registro`, retornando los resultados en el parámetro por referencia `reg`. Esta función permite al usuario la posibilidad de no continuar con la entrada de nuevos registros de pacientes. La lectura de la información de todos los pacientes es realizada por la función `leervector`. Esta función retorna la información en vector, que es un parámetro recibido como puntero a `registro`. La función `mostrarlasconsultas`, está encargada de visualizar los nombres de los pacientes que cumplen la condición de la fecha. Por su parte el programa principal realiza las llamadas a las distintas funciones.

La codificación de este problema se encuentra en la página Web del libro.

10.10. Para resolver el problema se programa la función `Cadena_A_Numero` que recibe la cadena de caracteres y lo retorna como número real. El algoritmo tiene las siguientes partes: saltarse los blancos; decidir el signo; extraer la parte entera; añadir la parte decimal. Para hacerlo, basta con recorrer la cadena de caracteres con un puntero carácter a carácter y convertir los caracteres a dígitos, usando el número ASCII asociado $9 = '9' - '0'$. Se incluye, además, un programa principal que efectúa llamadas a la función.

La codificación de este problema se encuentra en la página Web del libro.

10.11. Para programar una pequeña calculadora, se definen en primer lugar las funciones que se pueden usar en ella. En este caso las de sumar, restar, multiplicar, dividir y resto de números enteros. Se define un array de punteros a función, `arrayfunc[]`, inicializado con cada una de las operaciones. Asimismo, se tratan los símbolos de cada una de las funciones en otro array operadores, inicializado a cada uno de ellos. En este caso estos símbolos son: +, -, *, /, %. Una vez definidos dos operandos como variables de tipo `char` denominados `operando1`, `operando2`, y un operador, se leen en primer lugar un operando y un operador, y se itera hasta fin de datos (=), leyendo el siguiente operador, evaluando la expresión, realizando la correspondiente llamada a la función almacenada en `arrayfunc`, y volviendo a solicitar un nuevo operador.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 10.1.** 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;
};
```

- 10.2.** Escribir una función que tenga como argumento un puntero al tipo de dato `boton` del ejercicio anterior y lea de la entrada los distintos datos.

- 10.3.** 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

por columnas. 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.

- 10.4.** Escribir una función cuyo prototipo sea similar al de la función del ejercicio anterior, cuyo propósito sea leer de la entrada las filas pares. Escribir la implementación de la función aplicando la aritmética de punteros.

- 10.5.** Escribir funciones que usando la aritmética de punteros y punteros como parámetros realicen la suma, resta y producto de matrices.

- 10.6.** Escribir una función que reciba como parámetro un vector y un dato, y retorne la dirección de la posición del vector donde se encuentre el dato.

PROBLEMAS PROPUESTOS

- 10.1.** Dado un array que contiene una serie de registros con los datos de clientes de un establecimiento, realizar una función en el 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.

- 10.2.** Se quiere evaluar las funciones $f(x)$, $g(x)$, $h(x)$ 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, utilizando un array de punteros a función. Las funciones son las siguientes:

$$\begin{aligned}f(x) &= 3e^x - 2x; g(x) = -x\sin(x) + 1.5 \\h(x) &= \cos(x) + \tan(x); z(x) = x^3 - 2x + 1\end{aligned}$$

- 10.3.** 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.

- 10.4.** 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.

- 10.5.** Escribir una función que tenga como entrada una cadena de caracteres en la que se introducen distintos núme-

ros reales separados por blancos y devuelva un array de números reales (por ejemplo la cadena “25.56 23.4” se ha de convertir en los correspondientes valores reales y retornarlos en un vector).

- 10.6.** 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 puntero de función aplicar uno de estos métodos para encontrar una raíz de dicha ecuación.

- 10.7.** Se pretende gestionar un vídeo club; se dispone de 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 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.

- 10.8.** 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 también preguntar qué cliente tiene una determinada cinta.

- 10.9.** Escribir una función genérica que reciba como parámetro un vector y el nombre del método de ordenación, y realice la ordenación por el método indicado. **Nota:** úsense los métodos de ordenación codificados en el Capítulo 12.

CAPÍTULO 11

Gestión dinámica de la memoria

Introducción

Los programas pueden crear variables globales o locales. Estas clases de variables se definen cuando se compila el programa por lo que el compilador reserva (define) espacio para almacenar valores de los tipos de datos declarados 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.

C++ asigna memoria en el momento de la ejecución en el *montículo* o *montón* (**heap**), mediante las funciones `malloc()` y `free()`. C++ ofrece un método mejor y más eficiente que C para realizar la gestión dinámica (en tiempo de ejecución) de la memoria. En lugar de llamar a una función para asignar o liberar memoria, C++ proporciona dos operadores para la gestión de la memoria: `new()` y `delete()`, que asignan y liberan la memoria de una zona de memoria denominada *almacén libre* (`free store`).

11.1. 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*.

Un 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.

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.

Error típico de programación en C++

La declaración de un array exige especificar su longitud como una expresión constante, así se declara un array de 100 elementos: `char cadena[100];`

Si se utiliza una variable o una expresión que contiene variables se producirá un error.

El mapa de memoria del modelo de un programa grande es muy similar al mostrado en la Figura 11.1. El diseño exacto dependerá del modelo de programa que se utilice. El almacén libre es, esencialmente, toda la memoria que queda libre después de que se carga el programa.

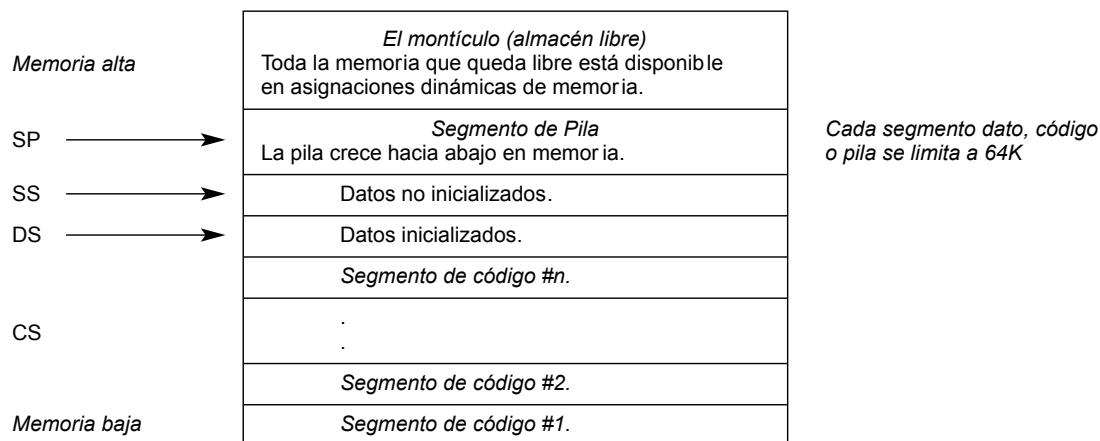


Figura 11.1. Mapa de memoria de un programa

En C las funciones `malloc()` y `free()` asignan y liberan memoria de un bloque de memoria denominado *el montículo del sistema*. C++ ofrece un nuevo y mejor método para gestionar la asignación dinámica de memoria, los operadores `new` y `delete`, que asignan y liberan la memoria de una zona de memoria llamada *almacén libre*.

Los operadores `new` y `delete`, son versátiles y fiables ya que el compilador realiza verificación de tipos cada vez que un programa asigna memoria con `new`. Se implementan como operadores por lo que los programas pueden utilizarlos sin incluir ningún archivo de cabecera. Nunca requieren *moldeado* (conversión forzosa de tipos) de tipos y eso hace a `new` y `delete` sean fáciles de usar. Los objetos creados con `new` residen en el *almacenamiento libre*. Los objetos del almacenamiento libre se eliminan con el operador `delete`.

10.2. El operador new

El operador `new` asigna un bloque de memoria que es el tamaño del tipo de dato. El dato u objeto dato puede ser un `int`, un `float`, una estructura, un array o cualquier otro tipo de dato. El operador `new` devuelve un puntero, que es la dirección del bloque asignado de memoria. El formato del operador `new` es:

```
puntero = new nombreTipo (inicializador opcional)
```

o bien

```
1. tipo *puntero = new tipo           // No arrays
2. tipo *puntero = new tipo[dimensiones] // Arrays
```

El formato 1 es para tipos de datos básicos y estructuras, y el segundo es para arrays. Cada vez que se ejecuta una expresión que invoca al operador `new`, el compilador realiza una verificación de tipo para asegurar que el tipo del puntero especifi-

cado en el lado izquierdo del operador es el tipo correcto de la memoria que se asigna en la derecha. Si los tipos no coinciden, el compilador produce un mensaje de error de volviendo NULL.

Sintaxis de los operadores new y delete

```
puntero = new tipo;
delete puntero ;
```

El operador `new` devuelve la dirección de la variable asignada dinámicamente. El operador `delete` elimina la memoria asignada dinámicamente a la que se accede por un puntero.

Sintaxis de asignación-desasignación de un array dinámico.

```
Puntero Array = new[tamaño Array]
delete[] Puntero Array ;
```

EJEMPLO 11.1. New y delete de tipo básico y cadena.

Se asigna memoria dinámica a una cadena de caracteres y a un entero y se libera la memoria a asignada.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char *Cadena = " Montes de Toledo en Castilla la Mancha";
    int *pEntero, lonCadena, Entero = 15, ;
    char *pCadena;

    lonCadena = strlen(Cadena);
    pCadena = new char[lonCadena+1]; //Memoria con una posición fin cadena
    strcpy(pCadena, Cadena); //copia Cadena a nueva área de memoria
    pEntero = new int; //se reserva memoria para un entero
    *pEntero = Entero; // se almacena en contenido de pEntero 15
    cout << " pCadena =" << pCadena << " longitud = "<< lonCadena<<endl;
    delete pCadena; // libera memoria de pCadena
    cout << " *pEntero = " << *pEntero << endl ;
    delete pEntero; // libera memoria de pEntero
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Se puede invocar al operador `new` para obtener memoria para un array, incluso si no se conoce con antelación cuánta memoria requieren los elementos del array. Todo lo que se ha de hacer es invocar al operador `new` utilizando un puntero al array. Si `new` no puede calcular la cantidad de memoria que se necesita cuando se compila su sentencia, `new` espera hasta el momento de la ejecución, cuando se conoce el tamaño del array, y asigna, a continuación, la memoria necesaria.

Cuando se asigna memoria con el operador `new`, se puede situar un valor o expresión de inicialización dentro de paréntesis al final de la expresión que invoca el operador `new`. C++ inicializa entonces la memoria que se ha asignado al valor que se ha especificado. Sin embargo, no se puede inicializar cada elemento de un array con la orden `new`.

EJEMPLO 11.2. Reserva dinámica de memoria e inicialización.

Se reserva e inicializa memoria dinámica de un entero y de un vector de enteros dinámico, visualizando los contenidos.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i, *v, n, *pEntero;

    pEntero = new int(20);           //reserva e inicialización de memoria
    cout << " *pEntero inicializado: " << *pEntero << endl;
    cout << " introduzca dimension de v : ";
    cin >> n;
    v = new int[n];                // reserva dinámica no se puede inicializar

    for( i = 0; i < n; i++)
        v[i] = 10 * (i+1);          // relleno del vector
    cout <<" vector :\n";

    for( i = 0; i < n; i++)
        cout << " v[" << i << "]=" << *v++;//salida de datos con puntero
    cout << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Para asignar memoria dinámica a un array multidimensional, se indica cada dimensión del array de igual forma que se declara un array multidimensional, haciendo uso de tipos sinónimos, pero no es exactamente el mismo tipo de dato.

```
typedef float * col;
float ** p;                      // también col *p;
float m[5][5];                    // m apunta a un array de punteros de 5 reales

p = new col[5];                   // p apunta a un array de punteros reales
                                    // de cualquier longitud
for(int i = 0; i < 5; i++)
    p[i] = new int[5];            // p apunta a un array de punteros de 5 reales
```

pero hay que tener en cuenta que ahora son expresiones equivalentes y, por tanto, pueden mezclarse los códigos:

```
p[i][j]
*(*(p+i)+j)
*(p[i]+j)
```

pero la expresión $\ast(\&p[0][0] + i * 5 + j)$ es distinta a las anteriores y no debe mezclarse en los códigos (la expresión anterior con las otras), ya que el compilador debe almacenar la información “ $p[i]$ es un puntero que apunta a un array de punteros a reales de longitud variable”, a diferencia de $m[i]$ que apunta a un array de exactamente 5 reales.

EJEMPLO 11.3. Reserva dinámica de memoria interactiva para matrices. Tratamiento como puntero a puntero de un tipo dado.

Se asigna memoria dinámica a una matriz cuadrada de dimensión $n \times n$, llenando la matriz con datos fijos y visualizando la matriz de la forma estándar y con punteros.

```
#include <cstdlib>
#include <iostream>
```

```

using namespace std;

int main(int argc, char *argv[])
{
    int i, j, n;
    typedef int* col;           // col es un sinónimo de puntero a entero
    int **m;                   // m es puntero que apuntan a punteros a enteros

    cout << " introduzca dimension de m : ";
    cin >> n;
    m = new col[n];           // reserva de memoria para n punteros enteros

    for( i = 0; i < n; i++)
    {
        m[i] = new int[n];      // m[i] puntero a vector de n enteros
                                // /m puntero a puntero de enteros
        for (j = 0; j < n; j++)
            m[i][j] = 10 * (i + 1) + j;
    }
    cout << "\n visualizacion matriz con indexacion: \n";

    for( i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            cout << " " << m[i][j];          // indexación tipo matriz
            cout << endl;
    }
    cout << "\n visualizacion matriz con punteros: \n" ;

    for( i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            cout << " " << *(*(m+i)+j);    // también válido con *(m[i]+j)
            cout << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 11.4. *Lectura y visualización de matrices cuadradas con funciones cuya memoria se reserva dinámicamente e interactivamente.*

Se escriben las funciones de lectura y escritura de matrices cuadradas, cuya memoria se ha reservado dinámicamente en un programa principal.

```

#include <cstdlib>
#include <iostream>
using namespace std;
typedef int* col;

void leermatriz (int** m, int n)
{
    int i, j;

```

```

        for( i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                cin >> m[i][j] ;
    }

void escribirmatriz (int** m, int n)
{
    int i, j;

    for( i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            cout << " " << m[i][j] ;
        cout << endl;
    }
}

int main(int argc, char *argv[])
{
    int n;
    int **m;                                // válido con col *m;

    cout << " introduzca dimension de m : ";
    cin >> n;
    m = new col[n];
    for (int i = 0; i < n; i++)
        m[i] = new int[n];
    leermatriz(m,n);
    escribirmatriz(m,n);
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 11.5. *Lectura y visualización de un texto almacenado en memoria dinámica.*

Lee un texto de 10 líneas, y se visualizan indicando la longitud de cada línea. La memoria se ajusta a la longitud mínima de la entrada de datos. Para ello, se reserva memoria interactivamente a un array de punteros a caracteres. Se lee una línea en un *buffer* de longitud constante 81, y se ajusta la longitud del texto de entrada al valor leído.

```

int main(int argc, char *argv[])
{
    typedef char * columna;
    columna * textoentrada;
    int i, n = 10;
    char buffer[81];

    textoentrada = new columna[n];                      // reserva de memoria
    for (i = 0; i < n; i++)
    {
        cout << " Introduzca linea: ";
        cin.getline(buffer,80);
        textoentrada[i] = new char [strlen (buffer)+1];   //ajuste
        strcpy (textoentrada[i], buffer);
    }
}

```

```

cout << "longitud      linea\n";
for (i = 0; i < n; i++)           // visualización con punteros
    cout << " " << strlen(*(textoentrada + i)) << " :"
        << *(textoentrada + i) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

11.3. El operador delete

Cuando se ha terminado de utilizar un bloque de memoria previamente asignado por `new`, se puede liberar el espacio de memoria y dejarlo disponible para otros usos, mediante el operador `delete`. 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 formato es:

```

delete puntero; si la reserva es puntero = new tipo;
delete[] Puntero Array; si la reserva es Puntero Array = new[tamaño Array]

```

EJEMPLO 11.6. Reserva espacio en memoria para un array de 5 elementos, y la libera.

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    float *DatosI = new float[5];
    int i;

    for (i = 0; i < 5; i++)
        DatosI[i] = 2 * i + 1;
    cout << " los 5 primeros impares son \n" ;

    for (i = 0; i < 5; i++)
        cout << " " << DatosI[i];
    cout << endl;
    delete [] DatosI;                                // Libera memoria
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 11.7. Reserva espacio en memoria para un array dinámico de n elementos de estructuras.

Se leen y visualizan mediante funciones, las coordenadas de n puntos del plano, almacenando los resultados en un array dinámico.

```

#include <cstdlib>
#include <iostream>
using namespace std;
struct Punto
{

```

```
float CoordX;
float CoordY;
};

Punto *Vector;
void Leer_Punto( Punto &p)
{
    cout << " coordenada x e y :";
    cin >> p.CoordX >> p.CoordY;
}

void Escribir_Punto( Punto p)
{
    cout << " " << p.CoordX << " " << p.CoordY << endl;
}

void LeerVectorPuntos( Punto *Vector, int n)
{
    int i;

    for (int i = 0; i < n; i++)
        Leer_Punto(Vector[i]);
}

void EscribirVectorPuntos( Punto *Vector, int n)
{
    int i;

    for (int i = 0; i < n; i++)
        Escribir_Punto(Vector[i]);
}

int main(int argc, char *argv[])
{
    int n;

    cout << " ¿cuantos puntos? : ";
    do
        cin >> n;
    while (n <= 0);
    Vector = new Punto[n];
    LeerVectorPuntos(Vector, n);
    EscribirVectorPuntos(Vector, n);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJERCICIOS

11.1. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int * p1, *p2;

    p1= new int;
    *p1 = 45;
    p2 = p1;
    cout << " *p1 == " << *p1 << endl;
    cout << " *p2 == " << *p2 << endl;
    *p2 = 54;
    cout << " *p1 == " << *p1 << endl;
    cout << " *p2 == " << *p2 << endl;
    cout << " viven los punteros \n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

11.2. ¿Cuál es la salida del siguiente código?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int * p1, *p2;

    p1= new int;
    p2= new int;
    *p1 = 10;
    *p2 = 20;
    cout << *p1 << " " << *p2 << endl;
    *p1 = *p2;
    cout << *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout << *p1 << " " << *p2 << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

11.3. Suponga que un programa contiene el siguiente código para crear un array dinámico.

```
int * entrada;
entrada = new int[10];
```

Si no existe suficiente memoria en el montículo, la llamada anterior a new fallará. Escribir un código que verifique si esa llamada a new falla por falta de almacenamiento suficiente en el montículo (heap). Visualice un mensaje de error en pantalla que lo advierta expresamente.

- 11.4.** Suponga que un programa contiene código para crear un array dinámico como en el Ejercicio 11.3, de modo que la variable entrada está apuntado a este array dinámico. Escribir el código que rellene este array de 10 números escritos por teclado.

- 11.5.** 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;
};
```

- 11.6.** Una vez asignada memoria al puntero b del Ejercicio 10.5 escribir sentencias para leer los campos rotulo y codigo.

- 11.7.** Declarar una estructura para representar un punto en el espacio tridimensional con un nombre. Declarar un puntero a la estructura que tenga la dirección de un array dinámico de n estructuras punto. Asignar memoria al array y comprobar que se ha podido asignar la memoria requerida.

PROBLEMAS

- 11.1.** Dada la declaración de la estructura punto del Ejercicio 11.7, escribir 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).

- 11.2.** Añadir al problema anterior una función que muestre los puntos de un vector dinámico cuya tercera coordenada sea mayor que un parámetro que recibe como dato.

- 11.3.** Añadir al problema anterior el punto más alejado del origen de coordenadas.

- 11.4.** Se pretende representar en el espacio tridimensional triángulos, pero minimizando el espacio necesario para almacenarlos. Teniendo en cuenta que un triángulo en el espacio tridimensional viene determinado por las coordenadas de sus tres vértices, escribir una estructura para representar un triángulo que usando el vector dinámico de los problemas anteriores, almacene el índice de los tres vértices así como su perímetro o y área (de esta forma se almacena los tres vértices con tres enteros en lugar de con 9 reales, ahorrando memoria). El perímetro y el área deben ser calculados mediante una función a partir del índice de los tres vértices, y el vector dinámico de puntos del espacio tridimensional.

- 11.5.** Añadir al problema anterior una función que lea y almacene en un vector los m triángulos del espacio tridimensional cuyos vértices se encuentren en el array dinámico de puntos. Usar las funciones y estructuras definidas en los problemas anteriores.

- 11.6.** Añadir al problema anterior una función que visualice los triángulos del vector de triángulos.

- 11.7.** Dada la declaración del array de punteros:

```
#define N 4
char *linea[N];
```

Escribir las sentencias de código para leer N líneas de caracteres y asignar cada línea a un elemento del array.

- 11.8.** Escribir un código de programa que elimine del array de N líneas del Problema 11.7 todas aquellas líneas de longitud menor que 20.
- 11.9.** Escribir una función que utilice punteros para copiar un array de elementos *double*.
- 11.10.** Escribir un programa que lea un número o determinado de enteros (1000 como máximo) del terminal y los visualice en el mismo orden y con la condición de que cada entero o sólo se escriba una vez. Si el entero ya se ha impreso, no se debe visualizar de nuevo. Por ejemplo, si los números siguientes se leen desde el terminal 55 78 -25 55 24 -3 7 el programa debe visualizar lo siguiente: 55 78 -25 55 24 -3 7. Se debe tratar con punteros para tratar con el array en el que se hayan almacenado los números.
- 11.11.** Escribir 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.
- 11.12.** Escribir el código de la siguiente función que devuelva la suma de los elementos *float* apuntados por los n primeros punteros del array *p*.
- 11.13.** Escriba una matriz que reciba como parámetro un puntero doble a *float*, así como su dimensión, y retorne memoria dinámica para una matriz cuadrada.
- 11.14.** Escriba una función que genere aleatoriamente una matriz cuadrada simétrica, y que pueda ser llamada desde un programa principal que genere aleatoriamente la matriz con la función del Problema 11.13. El tratamiento de la matriz debe hacerse con punteros.
- 11.15.** Escribir una función que decida si una matriz cuadrada de orden n es mayoritaria. “una matriz cuadrada de orden n se dice que es mayoritaria, si existe un elemento en la matriz cuyo número de repeticiones sobrepasa a la mitad de $n \times n$. El tratamiento debe hacerse con punteros.
- 11.16.** 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 otros 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. Diseñar 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.

SOLUCIÓN DE LOS EJERCICIOS

- 11.1.** La primera sentencia `p1 = new int;` reserva memoria para el puntero *p1*. La segunda sentencia asigna a *p2* el valor de *p1*, por lo que los dos punteros apuntan a la misma dirección de memoria y las dos sentencias de salida, mostrarán el contenido de la dirección de memoria que es 45, asignado por la sentencia `*p1 = 45;`. Cuando se ejecuta la siguiente sentencia `*p2 = 54;` se cambia el contenido de la memoria apuntada por *p1* y *p2* al valor de 54, por lo que ésa será la salida que muestren las dos sentencias siguientes.
- 11.2.** Las dos primeras sentencias `p1 = new int;` `p2 = new int;` reservan memoria para los punteros *p1* y *p2*. Las dos sentencias siguientes asignan valores a los contenidos de los dos punteros, por lo que la salida de la sentencia `cout` será los valores asignados que son 20 y 30. La sentencia `*p1 = *p2;` asigna al contenido del puntero *p1* el contenido del puntero *p2*, por lo que la salida será 20 20. La siguiente sentencia cambia el contenido del puntero *p1* a 30, por lo que la nueva salida será 30 20.

- 11.3.** Teniendo en cuenta que la orden `new` devuelve `NULL` si no se puede asignar memoria, para resolver el problema basta con interrogar por el valor `r` retornando por la orden `new` y dar el mensaje correspondiente. El siguiente fragmento de programa, además de dar el mensaje de error pedido, informa aproximadamente de la memoria disponible en el montículo (heap).

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int *entrada;

    for (int i = 0; ; i++)          // bucle infinito
    {
        entrada = new int[5012];
        if (entrada == NULL)
        {   cout << "memoria agotada ";
            break;
        }
        cout << "asignado " << i* 10 << "K bytes" << endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS
}
```

- 11.4.** Una vez reservada memoria dinámica al vector, la variable `entrada` se comporta como el identificador de una variable que es un array unidimensional, por lo que la lectura se realiza de la forma estándar.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int *entrada;

    entrada = new int[10];
    for (int i = 0; i < 10; i++)
        cin >> entrada[i];
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- 11.5.** La solución viene dada por la siguiente sentencia.

```
boton * b;
b = new boton;
```

- 11.6.** Las sentencias solicitadas usan el operador de indirección flecha `->` para acceder desde el puntero `b` a cada uno de los campos de la estructura. Además, como el atributo `rotulo` es un puntero a una cadena, es necesario asignar, previamente, memoria a la cadena, o bien que apunte a una cadena de caracteres leída con anterioridad.

```
boton * b;
b = new boton;
cin >> b->codigo;
```

```
b->rotulo = new char [21];           // por ejemplo cadena de 20 caracteres
cin.getline(b->rotulo,20);
```

- 11.7.** Se asigna memoria dinámicamente mediante el uso de la variable *enter a n*, cuyo valor ha sido solicitado al usuario. Después, se comprueba si ha habido error en la asignación, analizando si *Vector3D* contiene *NULL*.

```
struct Punto3D
{
    char *Nombre;
    float CoordX;
    float CoordY;
    float CoordZ;
};

Punto3D *Vector3D;
int n;

cout << "introduzca n :";
cin >> n;

if ((Vector3D = new Punto3D[n]) == NULL)
{
    cout << " Error de asignación de memoria dinámica \n";
    exit(1);
}
```

SOLUCIÓN DE LOS PROBLEMAS

- 11.1.** La función *Leer_Punto3D* devuelve como parámetro por referencia las coordenadas de un punto de tres coordenadas, así como el *Nombre*. El *nombre*, al ser un puntero a *char*, se le asigna memoria dinámicamente, mediante la longitud de la cadena de entrada real que introduzca el usuario, para optimizar la memoria. La función *Leer_Vector_Puntos3D* retorna en el vector que recibe como parámetro los datos solicitados, realizando un total de *n* llamadas a la función que lee la estructura que define un punto en dimensión 3.

La codificación de este problema se encuentra en la página Web del libro.

- 11.2.** La programación de la función pedida es similar a la realizada en el problema anterior, excepto que en este caso se llama a una función que visualiza los datos de las coordenadas de un punto en dimensión 3 cuya ordenada sea mayor que la que recibe en su parámetro.

La codificación de este problema se encuentra en la página Web del libro.

- 11.3.** Para encontrar el punto más cercano al origen de coordenadas, es necesario definir la función *distancia_Origen* de un punto cualquiera *x* que, en este caso, será la distancia euclídea $d = \sqrt{x^2 + y^2 + z^2}$. Mediante el uso de esta función puede calcularse el punto *Mas_Cercano_Origen* mediante un algoritmo voraz que toma inicialmente como punto más cercano el primero, y posteriormente mediante un bucle *for*, elige como más cercano el que ya tenía o bien el que se está tratando. La función *Escribe_Mas_Cercano_Origen* visualiza el contenido del puntero del punto más cercano que recibe de la función que lo calcula.

La codificación de este problema se encuentra en la página Web del libro.

- 11.4.** Un triángulo está completamente determinado mediante el índice de los tres puntos del espacio tridimensional que lo definen, por lo que para almacenarlos, basta con incluir los índices de los tres puntos en un array de enteros.

El perímetro de un triángulo viene dado por la suma de las longitudes de sus tres lados, a , b , c . El área de un triángulo de lados a , b y c es $= \sqrt{p(p - a)(p - b)(p - c)}$ siendo p el semiperímetro $p = \frac{a + b + c}{2}$. Las longitudes de cada uno de los lados del triángulo, están determinadas por los módulos de los vectores formados por los puntos que determinan los vértices del triángulo. En concreto $\text{vector1} = p_1 - p_0$; $\text{vector2} = p_2 - p_0$ y $\text{vector3} = p_2 - p_1$, siendo p_0 , p_1 , y p_2 las coordenadas de sus tres puntos. Exactamente las coordenadas de un vector $v = p_1 - p_0$ son $v = (v_1, v_2, v_3) = (x_1, y_1, z_1) - (x_0, y_0, z_0)$, siendo $p_1 = (x_1, y_1, z_1)$ y $p_0 = (x_0, y_0, z_0)$. Y el módulo del vector v viene dado por la expresión: módulo(v) = $\sqrt{v_1^2 + v_2^2 + v_3^2}$. Para calcular la distancia entre dos puntos se define la función `distancia_puntos` que recibe como parámetros los dos puntos. El cálculo del perímetro y el área de un triángulo se realiza, encontrando la distancia entre cada par de puntos, y obteniendo los valores correspondientes mediante las expresiones que lo determinan. Para poder realizarlo hay que tener en cuenta que es necesario disponer en todo momento del vector de puntos que contiene las coordenadas del espacio tridimensional, ya que el triángulo, sólo almacena los índices de los puntos. Esto obliga a que la función `Calcula_perímetro_Area_Triangulo` tenga como parámetros el vector dinámico de puntos del espacio, así como el triángulo.

La codificación de este problema se encuentra en la página Web del libro.

- 11.5.** La lectura de los triángulos se realiza leyendo los tres vértices de cada uno de ellos, y almacenando el área y perímetro. La función `leer_triangulos` realiza la lectura de los vértices de los m triángulos, y los almacena en un vector de triángulos `VT`, mediante un bucle `for` que itera desde 0 hasta $m - 1$. En cada iteración, solicita los índices de los tres, puntos del triángulo comprobando que son distintos, y se encuentran dentro del rango de puntos del espacio disponible ($0, 1, \dots, n$) con ayuda de la función `corecto`.

La codificación de este problema se encuentra en la página Web del libro.

- 11.6.** La visualización de un triángulo se realiza presentando los vértices que lo forman con ayuda de la función `Escribir_Punto3D`, así como su perímetro y área. Hay que tener en cuenta, que en todo momento se necesita el vector dinámico de puntos del espacio para poder visualizarlo, ya que sólo se almacenan los índices de los vértices. La presentación de todos los puntos se realiza por la función `Escribir_triangulos`.

La codificación de este problema se encuentra en la página Web del libro.

- 11.7.** En este caso, la declaración dada no es más que un array de N punteros a caracteres `lnea`. Es, por tanto, un puntero a un `char`. Ésta es una forma bastante eficiente de minimizar el espacio de memoria reservado para todo el texto, ya que se puede ajustar la longitud de cada línea al tamaño real del dato de entrada. Para poder realizarlo, hay que usar un buffer temporal (array de longitud fija de caracteres) declarado de una cierta longitud para leer cada línea, y una vez se haya leído, reservar memoria dinámica con un carácter más que la longitud real de la cadena leída (este carácter más es para el fin de cadena '\0'). El programa que se codifica solicita el texto, y lo visualiza.

La codificación de este problema se encuentra en la página Web del libro.

- 11.8.** Si el array `lneas` contiene líneas válidas, se procede a la eliminación de las líneas de la siguiente forma: una variable `n1`, indica en cada momento las líneas válidas que hay en el texto. Cuando una línea es suprimida se decrementa `n1` en una unidad. Si una línea cualquiera `i` cumple la condición de ser eliminada, se libera la memoria de la línea, y se mueven las líneas que ocupan posiciones superiores en el texto una posición hacia atrás (en la posición actual, se pone la siguiente, realizado con el bucle `for j`). Si una línea cualquiera `i` no cumple la condición de ser eliminada se incrementa el contador de líneas `i` en una unidad. Este contador de líneas `i` se inicializa a 0; termina cuando `i` toma el valor de `n1`. Como además en cada iteración `i` se incrementa `i`, o se decremente `n1`, el bucle `while` siempre termina.

La codificación de este problema se encuentra en la página Web del libro.

- 11.9.** La función `copiar`, recibe el vector `v` como puntero a `double`, y `Copia` como puntero a `double`. Se reserva espacio de memoria, para el vector que contendrá la copia, para posteriormente iterar copiando los elementos uno a uno mediante los punteros que se van desplazando a lo largo de cada uno de los vectores.

La codificación de este problema se encuentra en la página Web del libro.

11.10. La función `leer` recibe un puntero a un `float` `v` por referencia para que pueda asignarse memoria dinámicamente dentro de la función al vector `v` y se retorne los valores a la función de llamada. También tiene por referencia el parámetro `n` que es el número de elementos leídos del vector menor o igual que 1000. Hay que observar que la lectura se realiza de datos del vector `v` y se realiza con un puntero auxiliar a `float`, `p`, previamente inicializado a `v` y que avanza en cada iteración del bucle de lectura una posición en el vector `v` (no puede realizarse esta misma operación con `v` ya que en ese caso cambia de posición el comienzo del array y los resultados serían incorrectos). La función `repe`, recibe como parámetro el vector `v` y una posición `i`, retornando `true` si el elemento que se encuentra en la posición `i` aparece almacenado en alguna posición anterior del array. De esta forma, en un bucle se puede recorrer todas las posiciones del vector de datos, y si el elemento que se está tratando, no está repetido en alguna posición anterior del array entonces se visualiza.

La codificación de este problema se encuentra en la página Web del libro.

11.11. Se declara, en primer lugar, un sinónimo de puntero a carácter que es `linea`. Posteriormente, las variables `texto` y `textol`, como punteros a `linea`, donde se leerá el texto original, y se hará la copia del texto sin blancos. Es decir, punteros dobles que son array de punteros a caracteres, o bien un array bidimensional de caracteres. Se reserva espacio en memoria para cada uno de los textos, para llamar a las funciones encargadas de leer el texto transformarlo y visualizarlo.

La función que realiza la lectura de las líneas del texto `Leer_Texto`, asigna dinámicamente la longitud de cada línea del texto, una vez conocida su longitud real dada por el usuario.

La función que transforma el texto, en otro texto sin blancos, `Transformar_Texto`, copia línea a línea los datos del texto original en una cadena de caracteres `buffer`, para ir eliminando los blancos uno a uno. Para realizarlo, se inicializan dos variables `j`, y `lon` apuntando al primer y último carácter del array de caracteres (carácter fin de cadena). Estas dos variables controlan el bucle que elimina los blancos. Si el carácter a tratar es un blanco, se elimina, haciendo retroceder todos los caracteres de la cadena una posición, y haciendo que la cadena termine en el carácter anterior (`lon--`; `buffer[lon]= '\0'`). Si el carácter a tratar no es un blanco se avanza la variable `j`.

La visualización del texto es realizada por la función `Mostrar_Texto`, que lo muestra carácter a carácter en lugar de utilizar las cadenas de caracteres, para que sirva de ejemplo de otra forma de visualizar las cadenas.

La codificación de este problema se encuentra en la página Web del libro.

11.12. La función se escribe con tratamiento de punteros. Como `p` es un vector de punteros a reales, se tiene que `* (p + i)` es un puntero a real, por lo que `* (* (p + i))` contiene el número real que hay que sumar.

La codificación de este problema se encuentra en la página Web del libro.

11.13. La función `generamatriz` tiene como parámetro por referencia un puntero doble a `float`, para retornar la memoria y un entero `n` que indica la dimensión de la matriz cuadrada, y genera dinámicamente en, primer lugar, los `n` punteros a las columnas y, posteriormente, los `n` punteros a cada uno de los reales de la columnas.

```
#include <cstdlib>
#include <iostream>
using namespace std;

typedef float* col; //col es un sinónimo de un puntero a float

void generamatriz( float ** &m, int n)
{
    m = new col[n];
    for (int i = 0; i < n; i++)
        m[i] = new float[n];
}

int main(int argc, char *argv[])
{
    int n;
```

```

float **m;
.....
generamatriz(m,n);
.....
}

```

- 11.14.** La función recibe como parámetro la matriz y asigna aleatoriamente el mismo valor a las posiciones $m[i][j]$ y $m[j][i]$. Usa dos macros para definir las funciones estándar de generación de números aleatorios.

La codificación de este problema se encuentra en la página Web del libro.

- 11.15.** Se programa en primer lugar la función `el_mas_repe` que encuentra el elemento que más veces se repite en una matriz así como el número de veces que lo hace. Esta operación es realizada mediante un bucle voraz. Se tiene en una variable `nmas-repe` el número de veces que se ha repetido el número `masrepe` hasta el momento. Una vez contado el número de veces que se encuentra el número actual en la matriz, el nuevo más repetido, es el que tenía o el nuevo. Para realizar el bucle voraz se exploran todos los elementos de la matriz mediante dos bucles `for` anidados controlados por las variables `i` y `j` respectivamente. Para cada posición actual $m[i][j]$, de la matriz se cuenta el número de veces que aparece repetido en la matriz el dato, y si es mayor que el que tenía el nuevo número más repetido se actualiza, así como su número de veces. En otro caso no hace nada.

La matriz será mayoritaria si el número de veces que se repite el elemento que más se encuentra a almacenado supera el número total de elementos de la matriz que es n^2 .

La codificación de este problema se encuentra en la página Web del libro.

- 11.16.** Se define una matriz dinámica de enteros de F filas y C columnas donde se leen los datos de la malla de números mediante una función `LeerMatriz`. La función `EscribeMatriz` por su parte visualiza los datos de la matriz mediante punteros.

Dado un punto cuya posición en la matriz es i, j , ($m[i][j]$) la media de los valores de los puntos que lo rodean se obtiene sumando todos los valores y dividiendo el resultado entre el número de puntos. Hay que tener en cuenta que no a todos los puntos lo rodean la misma cantidad. Casi todos tienen ocho puntos, pero, por ejemplo, los de las cuatro esquinas de la matriz ($m[0][0], m[0][C], m[F][0]$ y $m[F][C]$) sólo tienen 3 puntos que lo rodean. Tampoco tienen 8 puntos rodeándolos aquellos que se encuentran en la fila 0, columna 0, fila F o columna C. La función `Valormedio` es la encargada de realizar los cálculos necesarios para cada punto con dos bucles `for`, que recorren en principio los 9 candidatos, a vecinos y sólo trata los que realmente lo son.

Para decidir los puntos de la matriz que tienen ruido, basta con comprobar la condición para cada uno de ellos. Si un punto tiene ruido, entonces se pone en la posición i, j de la matriz de ruido ($ruido[i][j]$) el valor de 1 y 0 en otro caso. La función `EncontrarRuido` recorre todos los puntos de la matriz original con dos bucles `for` anidados asignando a cada posición de la matriz de ruido el valor correspondiente.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 11.1.** Un array unidimensional puede considerarse una constante puntero. ¿Cómo puede considerarse un array bidimensional?, ¿y un array de tres dimensiones?
- 11.2.** Dada la declaración del array de punteros:

```
#define N 4
char *l[N];
```

Escribir las sentencias de código para leer N líneas de caracteres y asignar cada línea a un elemento del array .

- 11.3.** Escribir una función que reciba el array dinámico de n elementos y amplíe el array en otros m puntos del espacio.

- 11.4.** ¿Qué diferencias existe entre las siguientes declaraciones?

```
char *c[15];
char **c,
char c[15][12];
```

PROBLEMAS PROPUESTOS

- 11.1.** En una competición de ciclismo se presentan n ciclistas. Cada participante se representa por el nombre, club, los puntos obtenidos y prueba en que participará en la competición. La competición es por eliminación. Hay pruebas de dos tipos, persecución y velocidad. En la de persecución participan tres ciclistas, el primero recibe 3 puntos y el tercero se elimina. En la de 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 aquellos que han participado en menos pruebas. El juego termina cuando no quedan ciclistas para alguna de las dos pruebas. Se han de mantener arrays dinámicos con los ciclistas participantes y los eliminados. El ciclista ganador será el que más puntos tenga.
- 11.2.** 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 de enteros 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.
- 11.3.** Un polinomio, $P(x)$, puede representarse con un array de tantos elementos como el grado del polinomio mas uno. Escribir un programa que tenga como entrada el grado n del polinomio, reserve memoria dinámicamente para un array $den+1$ elementos. En una función se introducen por teclado los coeficientes del polinomio, en orden decreciente. El programa tiene que permitir evaluar el polinomio para un valor dado de x .
- 11.4.** 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.
- 11.5.** Escribir una función que tome como entrada un número entero y produzca una cadena con los dígitos de su expresión en base dos.
- 11.6.** Escribir funciones para sumar restar y multiplicar números enteros positivos en binario.
- 11.7.** Construir un programa que simule el comportamiento de los clientes en la salida de un gran almacén. Se parte de un array de cajas a las que se van añadiendo al azar los clientes para pagar sus compras. Los clientes están representados por cadenas de asteriscos enlazadas con

las cajas, con tantos asteriscos como clientes. El programa debe poder permitir añadir clientes a las cajas según diferentes velocidades y simulando diversas velocidades en el servicio de cada caja. La evolución de todas las colas se comprueba visualmente imprimiendo las cadenas de asteriscos que representan los clientes de cada cola delante de cada caja.

- 11.8.** Dada la siguiente definición de registro:

```
struct reg
{
    char *autor;
    char *título;
    char *editorial;
    int anno;
    char *ISBN;
    char** comentarios;
};
```

Realizar un programa que genere dinámicamente una tabla a partir de un array de punteros a estructuras del tipo anterior apuntados por el siguiente puntero doble:

```
reg **tabla;
```

Una vez generada la tabla del tamaño indicado por el usuario, llenar adecuadamente los campos con la memoria exacta que cada registro necesite. El campo comentarios permite anotar varios comentarios respecto de cada obra.

- 11.9.** Añadir al programa anterior un menú que permita al usuario añadir elementos a la tabla, eliminar registros y modificar su contenido.
- 11.10.** Con los datos introducidos en los dos ejemplos anteriores en el array de punteros tabla, crear tres arrays del mismo tipo que apunten a los registros de tabla en el orden en el que estarían si estuviesen ordenados respectivamente por autor, título o editorial. Construir un programa que, además de realizar lo anterior, permita imprimir el contenido de los registros, según el orden original en que han sido introducidos los datos y seguir los tres órdenes definidos.
- 11.11.** De la misma forma que hace el sistema UNIX con las líneas de órdenes, realizar una función que tome como entrada una cadena y separe sus componentes en cadenas diferentes. Cada una de las cadenas resultantes ha de estar enlazada con un elemento de un array de punteros a carácter.

11.12. Se tiene una matriz de 20×20 elementos enteros. En la matriz hay un elemento repetido muchas veces. Se quiere generar otra matriz de 20×20 filas y que en cada fila estén sólo los elementos no repetidos. Escribir un programa que tenga como entrada la matriz de 20×20 , genere la matriz dinámica pedida y se muestre en pantalla.

11.13. Escribir un programa que lea una serie de códigos de artículos y muestre las frecuencias de aparición de cada uno de ellos construyendo un histograma de frecuencias. Para ello el programa establece cuántos códigos distintos se le han introducido y visualiza cada código en una línea de la pantalla seguida de tantos asteriscos como veces haya aparecido dicho código en los datos de entrada.

CAPÍTULO 12

Ordenación y búsqueda

Introducción

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 muy importante 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.

12.1. Búsqueda

El proceso de encontrar un elemento específico de un array se denominabúsquedas. Este proceso permite determinar si un array contiene un valor que coincide con un cierto *valor clave*. La búsqueda permite la recuperación de datos previamente almacenados. Si el almacenamiento se realiza en memoria, la búsqueda se denomina interna. Las búsquedas más usadas en listas son: secuencial (lineal) y binaria.

12.1.1. BÚSQUEDA SECUENCIAL

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado *clave*. En una búsqueda secuencial (a veces llamada *búsqueda lineal*), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro (de izquierda a derecha o vice versa). La búsqueda lineal consiste en recorrer cada uno de los elementos hasta alcanzar el final de la lista de datos. Si en algún lugar de la lista se encontrara el elemento buscado, el algoritmo deberá informar sobre la o las posiciones donde ha sido localizado.

EJEMPLO 12.1. *Búsqueda secuencial de una clave en un vector de n datos de izquierda a derecha comenzando en la posición 0.*

Retorna -1 si la clave no está en el vector V y la posición de la primera aparición sí se encuentra en el vector V.

```
int busquedaLinealID (int V [], int n, int clave)
{
    int i;
```

```

for (i = 0 ; i < n; i++)
    if (V[i] == clave)
        return i;
    return -1;
}

```

EJEMPLO 12.2. *Búsqueda secuencial de una clave en un vector de n datos de der echo a izquierda comenzando en la última posición.*

Retorna -1 si la clave no está en el vector V y la posición de la última aparición sí se encuentra en el vector V.

```

int busquedaLinealDI (int V [], int n, int clave)
{
    int i;

    for (i = n - 1; i >= 0; i--)
        if (V[i] == clave)
            return i;
    return -1;
}

```

Si la lista se encuentra ordenada y el dato a buscar no está en la lista, la búsqueda termina cuando el elemento a encontrar sea menor que el elemento en curso en una lista ordenada ascendentemente, o cuando sea mayor si se busca en una lista ordenada descendente.

EJEMPLO 12.3. *Búsqueda secuencial de una clave en un vector de n datos ordenado crecientemente de izquierda a derecha comenzando en la primera posición.*

La búsqueda secuencial se programa de forma ascendente (el bucle termina cuando ha encontrado el elemento, se está seguro que no se encuentra o cuando no hay más elementos en el vector) suponiendo que el vector está ordenado. La variable encontrada se pone a true cuando se termina la búsqueda de la clave, o bien porque no puede estar la clave en el vector, o bien porque ya se ha encontrado.

```

int BsecorAscendente(int V[], int n, int clave)
{
    int i = 0;
    bool encontrada = false;

    while ((!encontrada) && (i < n))          // Búsqueda secuencial ascendente
    {
        encontrada = (V[i] >= clave);         //encontrada true fin de búsqueda
        if (!encontrada)                      // avanzar en la búsqueda
            i++;
    }
    if (i < n)
        encontrada = V[i] == clave ;
    return encontrada ? i : -1;           //i si clave encontrada -1 otro caso
}

```

Estudio de la complejidad

El mejor caso, se produce cuando el elemento buscado sea el primero que se examina, de modo que sólo se necesita una comparación. En el peor caso, el elemento deseado es el último que se examina, de modo que se necesitan n comparaciones. En el caso medio, se encontrará el elemento deseado aproximadamente en el centro de la colección de datos, haciendo aproximadamente $n/2$ comparaciones. Por esta razón, se dice que la prestación media de la búsqueda secuencial es $O(n)$.

12.1.2. BÚSQUEDA BINARIA

Este método de búsqueda requiere que los elementos se encuentren almacenados en una estructura de acceso aleatorio de forma ordenada, es decir clasificados, con arreglo al valor de un determinado campo. En general, si la lista está ordenada se puede acortar el tiempo de búsqueda. La búsqueda binaria consiste en comparar el elemento buscado con el que ocupa en la lista la posición central y, según sea igual, mayor o menor que el central, parar la búsqueda con éxito, o bien, repetir la operación considerando una sublistas formada por los elementos situados entre el que ocupa la posición `central+1` y el último, ambos inclusive, o por los que se encuentran entre el primero y el colocado en `central-1`, también ambos inclusive. El proceso termina con búsqueda en fracaso cuando la sublistas de búsqueda se quede sin elementos.

EJEMPLO 12.4. Búsqueda binaria de una clave en un vector de n datos ordenados crecientemente.

La búsqueda binaria se programa, tomando en cada partición el elemento `central`. Decidiendo si el elemento buscado se encuentra en esa posición, o bien si hay que mover el índice izquierda o derecha de la partición. Se inicializa izquierda a 0, derecha a $n - 1$ y `central` es, en el comienzo de cualquier iteración del bucle, la posición del elemento que ocupa el centro. El bucle que realiza la búsqueda termina cuando se ha encontrado el elemento o bien cuando los índices de la partición se han cruzado.

```
int BusquedaBinaria(float V[], int n, float clave)
{
    int izquierda = 0, derecha = n - 1, central;
    bool encontrado = false;

    while((izquierda <= derecha) && (!encontrado))
    {
        central = (izquierda + derecha) / 2;
        if (V[central] == clave)                         // éxito en la búsqueda
            encontrado = true;
        else if (clave < V[central])                   // a la izquierda de central
            derecha = central - 1;
        else                                         // a la derecha de central
            izquierda = central + 1;
    }
    return encontrado ? central:-1;      //central si encontrado -1 otro caso
}
```

EJEMPLO 12.5. Seguimiento de la búsqueda binaria anterior de la clave 40 en el vector $V \{-8, 4, 5, 9, 12, 18, 25, 40, 60\}$.

Clave = C1 = 40, I = izquierda, C = central, D = derecha

V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	CL	I	C	D
-8	4	5	9	12	18	25	40	60	40	0	4	8
-8	4	5	9	12	18	25	40	60	40	5	6	8
-8	4	5	9	12	18	25	40	60	40	7	7	8

EJEMPLO 12.6. Búsqueda binaria de una clave en un vector de n datos ordenados decrecientemente.

Basta con cambiar la condición de búsqueda del Ejemplo 12.4

clave < V[central] por clave > V[central]

```

int BusquedaBinaria(float V[], int n, float clave)
{
    int izquierda = 0, derecha = n - 1, central;
    bool encontrado = false;

    while((izquierda <= derecha) && (!encontrado))
    {
        central = (izquierda + derecha) / 2;
        if (V[central] == clave)                         // éxito en la búsqueda
            encontrado = true;
        else if (clave > V[central])                   // a la izquierda de central
            derecha = central - 1 ;
        else                                              // a la derecha de central
            izquierda = central + 1;
    }
    return encontrado? central:-1;           //central si encontrado -1 otro caso
}

```

Estudio de la complejidad

El algoritmo determina en qué mitad está el elemento, y descarta la otra mitad. En cada visión, el algoritmo hace una comparación. El número de comparaciones es igual al número de veces que el algoritmo divide la lista en la mitad. El tamaño de las sublistas en las sucesivas iteraciones es: $n, n/2, n/4, n/8, \dots, 1$. Si se supone que n es aproximadamente igual a 2^k entonces k o $k+1$ es el número de veces que n se puede dividir hasta tener un elemento encuadrado ($k = \log_2 n$). Por consiguiente, el algoritmo es $O(\log_2 n)$ en el peor de los casos. En el caso medio $O(\log_2 n)$ y $O(1)$ en el mejor de los casos.

12.2. Ordenación

La *ordenación* o *clasificación* de datos es la operación consistente en disponer un conjunto de datos en algún orden determinado con respecto a uno de los *campos* de elementos del conjunto. La clasificación interna es la ordenación de datos que se encuentran en la memoria principal del ordenador. 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 $\text{lista}[i] \leq \text{lista}[j]$

y se dice que está en *orden descendente* si:

$i > j$ implica que $\text{lista}[i] \leq \text{lista}[j]$.

En todos los métodos de este capítulo, se utiliza el *orden ascendente* sobre vectores o listas (arrays unidimensionales). Los métodos de clasificación interna se dividen en dos tipos fundamentales al evaluar su complejidad en cuanto al tiempo de ejecución:

Directos. De complejidad $O(n^2)$ como burbuja (intercambio directo), selección e inserción.

Avanzados. De complejidad inferior $O(n^2)$ como Shell, y otros de complejidad $O(n \log n)$, como el método del montículo, ordenación por mezcla o *radix Sort*.

En este capítulo se estudian los métodos directos y el método Shell.

12.3. Ordenación por burbuja

La ordenación por burbuja (“bubble sort”) se basa en comparar elementos adyacentes de la lista (vector) e intercambiar sus valores si están desordenados. De este modo, se dice que los valores más grandes *burbujean* hacia la parte superior de la lista (hacia el último elemento), mientras que los valores más pequeños se *hunden* hacia el fondo de la lista. En el caso de un array (lista) con n elementos, la ordenación por burbuja requiere hasta $n - 1$ pasadas.

La primera pasada realiza los siguientes pasos: el vector tiene los elementos $A[0]$, $A[1]$, ..., $A[n - 1]$. El método comienza comparando $A[0]$ con $A[1]$; si están desordenados, se intercambian entre sí. A continuación, se compara $A[1]$ con $A[2]$, realizando el intercambio entre ellos si están desordenados. 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. Al terminar esta pasada el elemento mayor está en la parte superior de la lista.

La segunda pasada realiza un proceso parecido al anterior, pero la última comparación y, por tanto, el último posible intercambio es realizado con los elementos $A[n - 3]$ y $A[n - 2]$ y, por tanto, coloca el segundo elemento mayor en la posición $A[n - 2]$.

El proceso descrito se repite durante $n - 1$ pasadas teniendo en cuenta que en la pasada i 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 $n - 2$, el vector está ordenado.

EJEMPLO 12.7. Codificación del método de la burbuja.

Se realiza la ordenación poniendo en primer lugar el elemento mayor en la última posición del array. Posteriormente, se coloca el siguiente mayor en la penúltima posición, y así sucesivamente. Para realizar el proceso sólo se utilizan comparaciones de elementos consecutivos, intercambiándolos en el caso de que no estén colocados en orden.

```
void Burbuja(float A[], int n)
{
    int i, j;
    float auxiliar;

    for (i = 0; i < n - 1; i++)                                // n-1 pasadas
        for (j = 0; j < n - 1 - i; j++)                      // burbujeo de datos
            if (A[j] > A[j + 1])                            //comparación de elementos contiguos
            {                                                 // mal colocados intercambio
                auxiliar = A[j];
                A[j] = A[j + 1];
                A[j + 1] = auxiliar;
            }
    }
}
```

Estudio de la complejidad

En este método de ordenación, los dos bucles comienzan en cero y terminan en $n - 2$ y $n - i - 1$. El método de ordenación requiere como máximo $n - 1$ pasadas a través de la lista o el array. Durante la pasada 1, se hacen $n - 1$ comparaciones y como máximo $n - 1$ intercambios, durante la pasada 2, se hacen $n - 2$ comparaciones y como máximo $n - 2$ intercambios. En general, durante la pasada i , se hacen $n - i$ comparaciones y a lo más $n - i$ intercambios. Por consiguiente, en el peor caso, habrá un total de $(n - 1) + (n - 2) + \dots + 1 = n*(n - 1) / 2 = O(n^2)$ comparaciones y el mismo número de intercambios. Por consiguiente, la eficiencia del algoritmo de burbuja en el peor de los casos es $O(n^2)$.

EJEMPLO 12.8. Seguimiento del método de ordenación de burbuja para los datos 8, 3, 8, 7, 14, 6.

Si se ejecuta el código del Ejemplo 12.7 escribiendo el contenido del vector cada vez que se realiza un intercambio de datos, así como los datos intercambiados y el índice de la pasada se obtiene el siguiente resultado:

Vector desordenado		cambio	pasada
8 3 8 7 14 6			
comienza burbuja:			
3 8 8 7 14 6		8 3	0
3 8 7 8 14 6		8 7	0
3 8 7 8 6 14		14 6	0
3 7 8 8 6 14		8 7	1
3 7 8 6 8 14		8 6	1
3 7 6 8 8 14		8 6	2
3 6 7 8 8 14		7 6	3
Vector ordenado			
3 6 7 8 8 14			
Presione una tecla para continuar . . .			

12.4. Ordenación por selección

El método de ordenación se basa en seleccionar la posición del elemento más pequeño del array y su colocación 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. El algoritmo de ordenación por selección de una lista (vector) de n elementos se realiza de la siguiente forma: se encuentra el elemento menor de la lista y se intercambia el elemento menor con el elemento de subíndice 0. A continuación, se busca el elemento menor en la sublista de subíndices 1 .. $n - 1$, y se intercambia con el elemento de subíndice 1. Después, se busca el elemento menor en la sublista 2 .. $n - 1$ y se intercambia con la posición 2. El proceso continúa sucesivamente, durante $n-1$ pasadas. Una vez terminadas las pasadas la lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

EJEMPLO 12.9. Codificación del método de ordenación por selección.

```
void seleccion(float A[], int n)
{
    int i, j, indicemin;
    float auxiliar;

    for (i = 0; i < n - 1; i++)
    {
        indicemin = i; // posición del menor
        for(j = i + 1; j < n; j++)
            if(A[j] < A[indicemin])
                indicemin = j; // nueva posición del menor

        auxiliar = A[indicemin];
        A[indicemin] = A[i];
        A[i] = auxiliar;
    }
}
```

Estudio de la complejidad

Como primera etapa en el análisis del algoritmo se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. La función siempre realiza $n - 1$ intercambios (n , número de elementos del array), ya que hay $n - 1$ llamadas a intercambio). El bucle interno hace $n - 1 - i$ comparaciones cada vez; el bucle externo va desde 0 a $n - 2$ de modo que el número total de comparaciones C es $O(n^2)$, por lo que el número de comparaciones de clara ve es cuadrático. Ha de observarse que el algoritmo no depende de la disposición inicial de los datos, lo que supone una ventaja de un algoritmo de selección ya que el número de intercambios de datos en el array es lineal. $O(n)$.

EJEMPLO 12.10. Seguimiento del método de ordenación de selección para los datos 8, 6, 5, 7, 18, 7.

Al realizar un seguimiento del método de ordenación de selección codificado en el Ejemplo 12.9 y escribir el contenido del vector cada vez que se realiza un intercambio de datos, así como los datos intercambiados y el índice de la pasada se obtiene los siguiente:

Vector desordenado	cambio pasada
8 6 5 7 18 7	8 5 0
comienza selección:	
5 6 8 7 18 7	8 7 2
5 6 7 8 18 7	8 7 3
5 6 7 7 18 8	18 8 4
Vector ordenado	
5 6 7 7 8 18	
Presione una tecla para continuar . . .	

12.5. Ordenación por inserción

Este método consiste en tomar una sublistas inicial con un único elemento que se podrá considerar siempre ordenada. En la sublistas ordenada se irán insertando sucesivamente los restantes elementos de la lista inicial, en el lugar adecuado para que dicha sublistas no pierda la ordenación. El algoritmo considera que la sublistas $A[0], A[1], \dots, A[i-1]$ está ordenada, posteriormente inserta el elemento $A[i]$ en la posición que le corresponda para que la sublistas $A[0], A[1], \dots, A[i-1], A[i]$ esté ordenada, moviendo hacia la derecha los elementos que sean necesarios. La sublistas ordenada aumentará su tamaño cada vez que se inserta un elemento hasta llegar a alcanzar el de la lista original, momento en el que habrá terminado el proceso. Los métodos de ordenación por inserción pueden ser directo o binario, dependiendo de que la búsqueda se realice de forma secuencial o binaria.

EJEMPLO 12.11. Codificación del método de ordenación por inserción lineal.

La sublistas $0 \dots i - 1$ está ordenada, y se coloca el elemento que ocupa la posición i de la lista en la posición que le corresponde mediante una búsqueda lineal. De esta forma, la sublistas se ordena entre las posición $0 \dots i$. Si i varía desde 1 hasta $n - 1$ se ordena la lista de datos.

```
void Insercionlineal(float A[], int n)
{
    int i,j;
    bool encontrado;
    float auxiliar;

    for (i = 1; i < n; i++)
        {
            auxiliar = A[i]; // A[0], A[1], ..., A[i-1] está ordenado
            j = i - 1;
            encontrado = false;
            while ((j >= 0) && !encontrado)
                if (A[j] > auxiliar)
                    { // se mueve dato hacia la derecha para la inserción
                        A[j + 1] = A[j];
                        j--;
                    }
                else
                    encontrado = true;
            A[j + 1] = auxiliar;
        }
}
```

Estudio de la complejidad de la inserción lineal

El bucle `for` de la función se ejecuta $n - 1$ veces. Dentro de este bucle existe un bucle `while` que se ejecuta a lo más el valor de i veces para valores de i que están en el rango 1 a $n - 1$. Por consiguiente, en el peor de los casos, las comparaciones del algoritmo vienen dadas por $1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2}$. Por otra parte, el algoritmo mueve los datos como

máximo el mismo número de veces. Existen por tanto, en el peor de los casos, $\frac{n(n - 1)}{2}$ movimientos. Por consiguiente, el algoritmo de ordenación por inserción es $O(n^2)$ en el caso peor.

EJEMPLO 12.12. Seguimiento del método de ordenación de inserción lineal para los datos 14, 9, 16, 1, 13, 17.

Al ejecutar el código del Ejemplo 12.11, y escribir el contenido del vector al final de cada pasada se obtiene la salida siguiente:

```

Vector desordenado
14  9  16  1  13  17
comienza insercion lineal:
 9  14  16  1  13  17
 9  14  16  1  13  17
 1  9  14  16  13  17
 1  9  13  14  16  17
 1  9  13  14  16  17
Vector ordenado
 1  9  13  14  16  17
Presione una tecla para continuar . .

```

EJEMPLO 12.13. *Codificación del método de ordenación por inserción binaria.*

La única diferencia de los dos métodos de ordenación por inserción, radica en el método de realizar la búsqueda. En este caso es una búsqueda binaria, por lo que el desplazamiento de datos hacia la derecha debe hacerse después de haber encontrado la posición donde se debe insertar el elemento, que en esta codificación es siempre izquierda.

```

void Insercionbinaria(float A[], int n )
{
    int i, j, izquierda, derecha, centro;
    float auxiliar;

    for(i = 1; i < n; i++)
    {
        auxiliar = A[i];
        izquierda = 0;
        derecha = i - 1;
        while (izquierda <= derecha) //búsqueda binaria sin interruptor
        {
            centro = (izquierda +derecha) / 2;
            if(A[centro] > auxiliar)
                derecha = centro - 1;
            else
                izquierda = centro + 1;
        }
        // desplazar datos hacia la derecha
        for(j = i - 1; j >= izquierda; j--)
            A[j + 1] = A[j];
        A[izquierda] = auxiliar;
    }
}

```

Estudio de la complejidad de la inserción binaria

Los métodos de ordenación por inserción lineal e inserción binaria, sólo se diferencian entre sí, conceptualmente, en el método de la búsqueda. Por tanto, el número de movimientos de claves será el mismo en ambos casos es $\frac{n(n-1)}{2}$. En cuanto al número de comparaciones, cuando el intervalo tiene i elementos, se realizan $\log_2(i)$ comparaciones. Por tanto, el número de comparaciones es:

$$C = \sum_{i=1}^{n-1} \log_2(i) = \int_1^{n-1} \log_2(x) dx \approx n \log_2(n)$$

Es decir el número de comparaciones es $O(n \log_2(n))$, y el número de movimientos es $O(n^2)$.

12.6. Ordenación Shell

La ordenación Shell se suele denominar también *ordenación por inserción con incrementos decrecientes*. Es una mejora de los métodos de inserción directa y burbuja, en el que se comparan elementos que pueden ser no contiguos. La idea general de algoritmo es la siguiente. Se divide la lista original (n elementos) en $n/2$ grupos de dos elementos, con un intervalo entre los elementos de cada grupo de $n/2$ y se clasifica cada grupo por separado (se comparan las parejas de elementos y si no están ordenados se intercambian entre sí de posiciones). Se divide ahora la lista en $n/4$ grupos de cuatro con un intervalo o salto de $n/4$ y, de nuevo, se clasifica cada grupo por separado. Se repite el proceso hasta que, en un último paso, se clasifica el grupo de n elementos. En el último paso el método Shell coincide con el método de la burbuja.

EJEMPLO 12.14. Codificación del método de ordenación Shell.

```
void shell(float A[], int n)
{
    int i, j, k, salto = n/2;
    float auxiliar;

    while (salto > 0) // ordenación de salto listas
    {
        for (i = salto; i < n; i++) // ordenación parcial de lista i
        {
            // los elementos de cada lista están a distancia salto
            j = i - salto;
            while(j >= 0)
            {
                k = j + salto;
                if (A[j] <= A[k]) // elementos contiguos de la lista
                    j = -1; // fin bucle par ordenado
                else
                {
                    auxiliar = A[j];
                    A[j] = A[k];
                    A[k] = auxiliar;
                    j = j - salto;
                }
            }
            salto = salto / 2;
        }
    }
}
```

EJEMPLO 12.15. Seguimiento del método de ordenación de shell para la siguiente lista de datos 9, 18, 7, 9, 1, 7.

El resultado de ejecución de la función anterior, escribiendo los intercambios realizados así como el valor de salto en cada uno de ellos es:

Vector desordenado	cambio	salto
9 18 7 9 1 7		
comienza shell		
9 1 7 9 18 7	18 1 3	
1 9 7 9 18 7	9 1 1	
1 7 9 9 18 7	9 7 1	
1 7 9 9 7 18	18 7 1	
1 7 9 7 9 18	9 7 1	
1 7 7 9 9 18	9 7 1	
Vector ordenado		
1 7 7 9 9 18		
Presione una tecla para continuar . . .		

EJERCICIOS

- 12.1.** ¿Cuál es la diferencia entre búsqueda y ordenación?
- 12.2.** 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.
- 12.3.** ¿Cuál es la diferencia entre ordenación por selección y ordenación por inserción?
- 12.4.** Dada la siguiente lista 47, 32, 31, 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.
- 12.5.** Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88 y el número 20.
8, 13, 17, 26, 44, 56, 88, 97.

PROBLEMAS

- 12.1.** Modificar el algoritmo de ordenación por selección para que ordene un vector de enteros en orden descendente.
- 12.2.** Escribir un programa que cree un array de 100 enteros aleatorios en el rango de 1 a 200 y, a continuación, utilizando una búsqueda secuencial, realizar la búsqueda de 50 enteros seleccionados aleatoriamente (iguales o distintos). Al final del programa se deben visualizar las siguientes estadísticas:
- Número de búsquedas con éxito.
 - Número de búsquedas fallidas.
 - Porcentajes de éxito y de fallo.
- 12.3.** Escribir una mejora del método de ordenación de burbuja, consistente en no realizar más iteraciones cuando se detecte que el vector ya está ordenado.
- 12.4.** Modificar el método de ordenación por selección, para que en lugar de colocar el elemento menor en el lugar que le corresponde, lo haga con el elemento mayor.
- 12.5.** 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 el número de entradas modificadas (puede suponer que el vector de datos no contiene el valor -1 cuando se llama a la función).
- 12.6.** Escribir un programa que genere un vector de 1000 números enteros aleatorios positivos y menores que 100, y haga lo siguiente:
1. Visualizar los números de la lista en orden creciente.
 2. Calcular e imprimir la mediana (valor central).
 3. Determinar el número que ocurre con mayor frecuencia.

4. Imprimir una lista que contenga
- números menores de 30,
 - números mayores de 70,
 - números que no pertenezcan a los dos grupos anteriores.
- 12.7. Construir una función 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 una estructura con los campos (entero) día, mes, año y número de contrato.
- 12.8. Escribir un programa para leer un texto de datos formado por n cadenas de caracteres, siendo n una variable entera. La memoria que ocupa el texto se ha de ajustar al tamaño real. Una vez leído el texto se debe ordenar el texto original por la longitud de las líneas. El programa debe mostrar el texto leído y el texto ordenado.

SOLUCIÓN DE LOS EJERCICIOS

- 12.1. La ordenación de una lista es el proceso por el cual, los datos que se almacenan en ella se reestructuran de tal manera que quedan en orden creciente o decreciente respecto de una clave. La búsqueda de un elemento en una lista determinada consiste en decidir si el elemento se encuentra o no en la lista, dando alguna posición donde se encuentre, todas las posiciones, o simplemente informando de cuantas veces aparece en la lista.
- 12.2. Después de la primera pasada: 3, 8, 13, 25, 45, 23, 98, 58.
Después de la segunda pasada: 3, 8, 13, 25, 45, 23, 98, 58.
Después de la tercera pasada: 3, 8, 13, 25, 45, 23, 98, 58.
Después de la cuarta pasada: 3, 8, 13, 23, 25, 45, 98, 58.
- 12.3. El método de ordenación por selección, selecciona la posición del elemento más pequeño (o el mayor) y lo coloca en el lugar que le corresponde. Por consiguiente, elige la posición del elemento más pequeño del vector y lo intercambia con la posición 0. Posteriormente, elige la posición del siguiente más pequeño y lo intercambia con el que ocupa la posición 1, etc. El método de ordenación por inserción, parte de un vector parcialmente ordenado (hasta la posición $i - 1$) y decide cuál es la posición donde debe colocar el siguiente elemento del vector (el que ocupa la posición i) para ampliar la ordenación parcial (el vector quede ordenado hasta la posición i). Si se realizan iteraciones incrementando en cada una de ellas el vector parcialmente ordenado en un elemento, al final el vector quedará ordenado.
- 12.4. Después de realizar la primera pasada, el método de la burbuja dejaría el vector ordenado, por lo que este método no puede haberse usado: 3, 21, 32, 47, 56, 92.

Pasadas del método de selección:

primera: 3, 47, 21, 32, 46, 92.
segunda: 3, 21, 47, 32, 56, 92.

Pasadas del método de ordenación por inserción:

primera: 3, 47, 21, 32, 56, 92.
segunda: 3, 21, 47, 32, 56, 92.

En consecuencia, puede haber sido usado tanto el método de ordenación por selección como el de inserción, pero nunca el de burbuja.

- 12.5.** Búsqueda de la clave 88. 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 56, 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 tanto, se terminaría el bucle con éxito.

Búsqueda de la clave 20. Como 20 es menor que la clave central 26, hay que buscar en el subarray 8, 13, 17. En la segunda iteración la clave central es 13 que es menor que 20 por lo que hay que continuar la búsqueda en el subarray 17. Ahora la clave central es 17 y como es menor que 20 hay que buscar en el subarray de la derecha que no existe, y la búsqueda concluye en fracaso.

SOLUCIÓN DE LOS PROBLEMAS

- 12.1.** Para ordenar decrecientemente el vector basta con cambiar el sentido de la comparación de contenidos que se realiza en el método de ordenación $A[j] > A[indicemax]$. En la codificación se ha cambiado además el nombre del índice que se intercambia.

La codificación de este problema se encuentra en la página Web del libro.

- 12.2.** La función aleatorio rellena el array de n números enteros aleatoriamente en el rango solicitado, usando las macros `randomize` y `random` previamente definidas en sendas macros. La función que realiza la búsqueda secuencial de una clave en un vector V de n datos. La función ejercicio resuelve el problema, presentando las estadísticas solicitadas. Almacena en la variable `xitos` el número de éxitos, y en la variable `fallas` el número de fallos que ocurren.

La codificación de este problema se encuentra en la página Web del libro.

- 12.3.** En cada pasada del método de la burbuja, el método compara elementos consecutivos intercambiándolos si no están en el orden correspondiente. Cuando se realiza una pasada, y no se consigue mover ningún dato el vector está ordenado. Este hecho puede aprovecharse, para cambiar el primer bucle `for` de la pasada, por un bucle `while` que finalice cuando no se muevan datos. De esta forma, se mejora el método de ordenación de burbuja. La variable `ordenado` controla en cada pasada este hecho.

La codificación de este problema se encuentra en la página Web del libro.

- 12.4.** Se utiliza la idea de que en cada iteración se busca la posición del elemento mayor del array (que no ha sido ya colocado) y lo coloca en la posición que le corresponde. En primer lugar, se coloca el mayor de todos en la posición $n - 1$, a continuación, el siguiente mayor en la posición $n - 2$, continuando hasta la posición 1.

La codificación de este problema se encuentra en la página Web del libro.

- 12.5.** La función duplicados recibe como parámetro el vector A , y el número de elementos que almacena n . El contador `ntotal` cuenta el número de veces que se cambia un valor por -1. El contador `ntotal1`, cuenta el número de elementos que son cambiados por -1 en la iteración i . 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 `enter a 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.

La codificación de este problema se encuentra en la página Web del libro.

- 12.6.** Se usan las funciones *Burbuja*, para ordenar el vector y *aleatorio*, para generarlo aleatoriamente. La función *mostrar* se encarga de visualizar el contenido de un vector de tamaño *n* que se le pasa como parámetro. En el array *repeticiones*, se almacena el número de veces que se repite cada número. La mediana se encuentra siempre en la posición central del array una vez ordenado.

La codificación de este problema se encuentra en la página Web del libro.

- 12.7.** Hay que tener en cuenta que se pide una ordenación por tres claves, siendo la clave primaria, *año*, la secundaria *mes*, y la terciaria *día*. Siempre que haya que ordenar por este tipo de claves es necesario definir una función *mayor* que reciba dos datos de tipo estructura y decida cuándo la primera estructura es mayor que la segunda, de acuerdo con los datos del problema. En este caso, la estructura tiene el contrato que se define como un puntero a *char* y la fecha con sus tres miembros, definidos en el enunciado. El método de ordenación que se emplea es el de burbuja, pero puede ser cualquier otro. En todos ellos sólo hay que cambiar el operador *>* por la función *mayor*.

La codificación de este problema se encuentra en la página Web del libro.

- 12.8.** Se declara, en primer lugar, un sinónimo de puntero a carácter que es *cadena*. Posteriormente, las variables *Texto* y *Textoaux*, como punteros a cadena, donde se lee el texto original, y se hace la copia del texto para ordenarlo. Se reserva espacio en memoria dinámicamente para cada uno de los textos ajustados a la longitud real de cada línea, para posteriormente llamar a las funciones encargadas de leer el texto transformarlo y visualizarlo.

La función que realiza la lectura de las líneas del texto *Leer_Texto*, asigna dinámicamente la longitud de cada línea del texto, una vez conocida su longitud real dada por el usuario, para minimizar el espacio en memoria de almacenamiento de datos.

La función que transforma el texto, en otro texto ordenado, *Transformar_Texto_en_Oordenado*, copia línea a línea los datos del texto original en el texto auxiliar, así como en el vector *longitud*, la longitud de cada una de las líneas. A este vector de enteros se le asigna memoria dinámicamente, antes del proceso. Posteriormente, se usa un método de ordenación (inserción lineal) para ordenar los arrays paralelos *longitud* y *Textoaux*.

La visualización de los textos de entrada y texto ordenado es realizado por la función *Mostrar_Texto*, que lo muestra línea a línea.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 12.1.** Eliminar todos los números duplicados de una lista o vector. Por ejemplo, si el vector toma los valores 4, 7, 11, 4, 9, 5, 11, 7, 35 ha de cambiarse a 4, 7, 11, 9, 5, 3.
- 12.2.** Escribir la función de ordenación correspondiente al método *Shell* para poner en orden alfabético una lista de *n* nombres.
- 12.3.** Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria.
- 12.4.** Realizar un seguimiento de los distintos métodos de ordenación: inserción, *shell*, burbuja y selección para un vector que almacena los siguientes datos: 7, 4, 13, 11, 3, 2, 7, 9.
- 12.5.** Realizar un seguimiento del método de ordenación por selección para los datos de entrada 3, 7, 10, 1, 6, 4, 0.

PROBLEMAS PROPUESTOS

- 12.1.** 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 clasificada; está en orden inverso?
 2. Realizar los mismos cálculos si se utiliza el algoritmo burbuja.
- 12.2.** Se trata de resolver el siguiente problema escolar. Dadas las notas (1 a 10) 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 inserción binaria.
- 12.3.** Escribir un programa de consulta de teléfonos. El proceso leerá un conjunto de datos de 1000 nombres y números de teléfono de un archivo que contiene los números en orden aleatorio y las consultas han de poder realizarse por nombre y por número de teléfono.
- 12.4.** 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 visualiza de nuevo los vectores.
- Escribir un programa que realice esta tarea. **Nota:** utilizar como algoritmo de ordenación el método selección.
- 12.5.** 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, la visualice. La información debe ser ordenada por ventas de mayor a menor y visualizada de nuevo.
- 12.6.** 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*.
 - d) Buscar si existe el número x (leído del teclado) en la lista. Aplicar la búsqueda binaria.
- 12.6.** Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:
1. Tiempo empleado en *ordenar* la lista por cada uno de los métodos.
 2. Tiempo que se emplearía en *ordenar* la lista ya ordenada.
 3. Tiempo empleado en *ordenar* la lista ordenada en orden inverso.

CAPÍTULO 13

Clases y objetos

Introducción

Una **clase** es un tipo de dato que contiene código (funciones) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa. En este capítulo se tratarán las clases, un nuevo tipo de dato cuyas variables serán objetos. En la programación orientada a objetos, los objetos son los elementos principales de construcción. Sin embargo, la simple comprensión de lo que es un objeto, o bien, el uso de objetos en un programa, no significa que estén programando en un modo orientado a objetos. Un sistema orientado a objetos es aquel en el cual los objetos se interconectan y comunican entre sí.

13.1. Clases y objetos

La idea fundamental en los lenguajes orientados a objetos es combinar en una sola unidad *datos y funciones que operan sobre esos datos*. Tal unidad se denomina *objeto*. El concepto de *objeto*, al igual que los tipos abstractos de datos o tipos definidos por el usuario, es una colección de elementos de datos, junto con las funciones asociadas para operar sobre esos datos.

Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen cómo se construyen ciertos tipos de objetos. Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios, métodos o funciones miembro*. El término *función miembro* se utiliza, específicamente, en C++.

13.2. Definición de una clase

Una *clase* es la descripción de un conjunto de objetos; consta de **métodos** (o funciones miembro) y datos o **atributos** que resumen características comunes de un conjunto de objetos. Generalmente, una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por un *estado* específico y es capaz de realizar una serie de *operaciones*.

Una clase es, en esencia, la declaración de un tipo objeto. Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen cómo se construyen ciertos tipos de objetos. Definir una clase significa dar a la misma un nombre, así como dar nombre a los elementos que almacenan sus datos y describir las funciones que realizarán las acciones consideradas en los objetos.

Una *declaración* de una clase consta de una palabra reservada `class` y el nombre de la clase. Se utilizan tres diferentes *especificadores de acceso* para controlar el acceso a los miembros de la clase. Estos accesos son: `public`, `private` y `protected`.

Sintaxis

```
class nombre_clase {
public:
    // miembros públicos
protected:
    // miembros protegidos
private:
    // miembros privados
};
```

El especificador `public` define miembros públicos, que son aquéllos a los que se puede acceder por cualquier función. A los miembros privados que siguen al especificador `private` sólo se puede acceder por funciones miembro de la misma clase o por funciones y clases amigas. A los miembros que siguen al especificador `protected` se puede acceder por funciones miembro de la misma clase o de clases derivadas¹ de la misma, así como por sus amigas. Los especificadores `public`, `protected` y `private` pueden aparecer en cualquier orden. Si se omite el especificador de acceso, el acceso por defecto es privado. En la Tabla 13.1 cada “x” indica que el acceso está permitido al tipo del miembro de la clase listado en la columna de la izquierda.

Tabla 13.1. Visibilidad.

Tipo de miembro	Miembro de la misma clase	Amiga	Miembro de una clase derivada	Función no miembro
<code>private</code>	x	x		
<code>protected</code>	x	x		
<code>public</code>	x	x	x	x

El uso de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida para permitir que los detalles de implementación de los objetos sean ignorados.

El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

Las funciones miembro y los miembros datos de la *sección pública* forman la interfaz externa del objeto, mientras que los elementos de la *sección privada* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto. La *sección protegida* necesita para su comprensión el concepto de herencia que se explicará posteriormente.

C++ soporta las características de ocultación de datos con las palabras reservadas `public`, `private` y `protected`.

Las **funciones miembro (o métodos)** son funciones que se incluyen dentro de la definición de una clase, y pueden ser de los siguientes tipos:

- *Constructores y destructores*: funciones miembro a las que se llama automáticamente cuando un operador se crea o se destruye.

¹ Las clases derivadas se estudian en el Capítulo 14.

- **Selectores:** devuelven los valores de los miembros dato.
- **Modificadores o mutadores:** permiten a un programa cliente cambiar los contenidos de los miembros dato.
- **Operadores:** permiten definir operadores estándar C++ para los objetos de las clases.
- **Iteradores:** procesan colecciones de objetos, tales como arrays y listas.

EJEMPLO 13.1. Declaración de la clase *Punto* del espacio tridimensional que contiene las coordenadas *x*, *y*, *z* del espacio y funciones miembro para obtener y poner cada una de las coordenadas.

```
class Punto                                // nombre de la clase
{
public:                                     // zona pública con declaración de funciones miembro
    float ObtenerX() { return x; }           //selector x
    float ObtenerY() { return y; }           //selector y
    float ObtenerZ() { return z; }           //selector z
    void PonerX (float valx) { x = valx; }   //modificador x
    void PonerY (float valy) { y = valy; }   //modificador y
    void PonerZ (float valz) { z = valz; }   //modificador z
private:                                     // zona privada con declaración de atributos.
    float x, y, z;
};
```

13.3. Objetos

Una **clase** es la declaración de un tipo objeto, y una variable de tipo clase es un objeto. Cada vez que se construye un objeto a partir de una clase se crea *una instancia* (ejemplar) de esa clase. Por consiguiente, los objetos no son más que instancias de una clase. *Una instancia es una variable de tipo objeto*.

En general, instancia de una clase y objeto son términos intercambiables. Los tipos objetos definidos por el usuario se comportan como tipos incorporados que tienen datos internos y operaciones internas y externas. La sintaxis de declaración de un objeto es:

Sintaxis

```
nombre_clase identificador;
```

Así, la definición de un objeto *Punto* es: *Punto p;*

El **operador de acceso** a un miembro (*.*) selecciona un miembro individual de un objeto de la clase.

Ejemplo: Crear un punto *p*, fijar su coordenada *x* y visualizar dicha coordenada.

```
Punto p;
p.PonerX (100);      //pone la coordenada x a 100
cout << " coordenada x es " << p.ObtenerX();
```

Cada objeto consta de:

- **Estado (atributos)** determinados por sus datos internos. El estado de un objeto es simplemente el conjunto de valores de todas las variables contenidas dentro del objeto en un instante dado. Un atributo consta de dos partes: un nombre de atributo y un valor de atributo.
- **Métodos, funciones miembro**, operaciones o comportamiento (métodos invocados por mensajes). Estas operaciones se dividen en tres grandes grupos: operaciones que *manipulan* los datos de alguna forma específica (añadir, borrar, cambiar formato...); operaciones que realizan un *cálculo o proceso*; y operaciones que comprueban (*monitorizan*) un objeto frente a la ocurrencia de algún suceso de control. Cada método tiene un nombre y un cuerpo que realiza la acción o comportamiento asociado con el nombre del método.

Los objetos ocupan espacio en memoria y, en consecuencia, existen en el tiempo, y deberán *crearse* o *instanciarse*. Por la misma razón, se debe liberar el espacio en memoria ocupado por los objetos.

Un *mensaje* es la acción que realiza un objeto. El conjunto de mensajes a los cuales puede responder un objeto se denomina *protocolo* del objeto. Un mensaje consta de tres partes:

- *Identidad* del receptor.
- El *método* que se ha de ejecutar.
- *Información especial* necesaria para realizar el método invocado (argumentos o parámetros requeridos).

Dos operaciones comunes típicas en cualquier objeto son:

- *Constructor*: una operación que crea un objeto y/o inicializa su estado.
- *Destructor*: una operación que libera el estado de un objeto y/o destruye el propio objeto.

Estas operaciones se ejecutarán implícitamente por el compilador o explícitamente por el *programador*, mediante invocación a los citados constructores.

Los objetos tienen las siguientes características:

- Se agrupan en tipos llamados *clases*.
- Tienen *datos internos* que definen su estado actual.
- Soportan *ocultación de datos*.
- Pueden *heredar* propiedades de otros objetos.
- Pueden comunicarse con otros objetos pasando *mensajes*.
- Tienen *métodos* que definen su *comportamiento*.

EJEMPLO 13.2. Declarar la clase punto con distintas funciones miembros: modificadores, selectores, y varios tipos de constructores (se explican posteriormente). Los objetos declarados se inicializan y visualizan.

El programa principal declara cuatro objetos que se inicializan de distinta forma y se visualizan datos.

```
#include <cstdlib>
#include <iostream>
using namespace std;
class Punto // nombre de la clase
{
public: // zona pública con declaración de funciones miembro
    float ObtenerX() { return x; } //selector x
    float ObtenerY() { return y; } //selector y
    float ObtenerZ() { return z; } //selector z
    void PonerX (float valx) { x = valx; } //modificador x
    void PonerY (float valy) { y = valy; } //modificador y
    void PonerZ (float valz) { z = valz; } //modificador z

    Punto( float valx): x(valx), y(0.0), z(0.0){} //constructor
    // constructor que inicializa miembros
    //Las declaraciones anteriores son definiciones de métodos, ya que
    //incluyen el cuerpo de cada función. Son funciones en línea, inline

    Punto(); //constructor por defecto
    Punto(float , float, float); //constructor alternativo
    Punto( const Punto &p); // constructor de copia
    void EscribirPunto(); // selector
    void AsignarPunto(float , float, float); // modificador
```

```
//Las declaraciones anteriores no incluyen el cuerpo de cada función
//Son funciones fuera de línea. El cuerpo de la función se declara
//independientemente
private:           // zona privada con declaración de atributos.
    float x, y, z;
};

Punto::Punto()
{
    x = 0;           //cuerpo constructor por defecto
    y = 0;
    z = 0;
}

Punto::Punto(float valx, float valy, float valz)
{
    x = valx;        //cuerpo constructor alternativo
    y = valy;
    z = valz;
}

Punto::Punto ( const Punto &p)
{
    x = p.x;         // cuerpo del constructor de copia
    y = p.y;
    z = p.z;
}

void Punto::EscribirPunto()
{
    cout << " " << ObtenerX() << " " << ObtenerY()           // cuerpo
        << " " << ObtenerZ() << endl;
}

void Punto::AsignarPunto(float valx, float valy, float valz)
{
    //Cuerpo. El nombre de los parámetros puede omitirse en declaración
    x = valx;
    y = valy;
    z = valz;
}

int main(int argc, char *argv[])
{
    Punto p, p1;           // p, y p1 son objetos de la clase Punto
    Punto p2 = Punto(5.0, 6.0, 7.0);      //p2 objeto de la clase Punto
    Punto p3(8.0);          //p3 objeto de la clase Punto

    p.AsignarPunto(2.0,3.0,4.0);           // llamada a función miembro;
    p.EscribirPunto();                   // llamada a función miembro
    p1.EscribirPunto();                 //Escribe valores constructor por defecto
    p2.EscribirPunto();                 //Escribe valores constructor alternativo
    p3.EscribirPunto();                  //Escribe valores constructor lista
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Observaciones

- Los métodos ObtenerX, ObtenerY, ObtenerZ PonerX, PonerY, PonerZ, son funciones miembro en línea, y se declaran en una sola línea.
- Los métodos Punto, EscribirPunto, y AsignarPunto, tienen su prototipo en la declaración de la clase, pero la declaración de su cuerpo se realiza fuera de línea. La declaración de su prototipo termina en punto y coma ;.
- Los cuerpos de las funciones Punto, EscribirPunto, y AsignarPunto se realizan fuera de línea. Estas declaraciones van precedidas del operador de resolución de ámbito cuatro puntos (::). Las funciones miembro de una clase se definen de igual modo que cualquier otra función e xcepto que se necesita incluir el operador de resolución de ámbito :: en la definición de la función (en su cabecera).

Formato

```
Tipo_devuelto Nombre_clase ::Nombre_función (Lista_parámetros)
{
    sentencias del cuerpo de la función
}
```

El símbolo :: (*operador de resolución de ámbito*) se utiliza en sentencias de ejecución para acceder a los miembros de la clase. Por ejemplo, la expresión Punto::X se refiere al miembro dato X de la clase Punto.

- Las llamadas a las funciones miembro de los objetos p, p1, p2, p3 fuera de la clase (en el programa principal) tienen el formato p.funcionmiembro.
- Las llamadas a las funciones miembro de una clase dentro de la propia clase no van precedidas de punto, como se hace por ejemplo en el método EscribirPunto.
- Hay tres tipos de constructores que se explican posteriormente.
- Una clase de C++ es una generalización de una estructura de C. Una clase que sólo tiene miembros públicos y sin funciones es una estructura.
- Cada clase contiene un puntero implícito this que apunta a sí misma. El puntero this es la dirección de la instancia de una clase. *this es el objeto real.

13.4. Constructores

Un *constructor* es una función miembro que se ejecuta automáticamente cuando se crea el objeto de una clase. Los constructores tienen siempre el mismo nombre que la propia clase. Cuando se define un constructor no se puede especificar un valor de retorno, ni incluso void. Un constructor puede, sin embargo, tomar cualquier número de parámetros (cero o más).

Constructor por defecto. Un constructor que no tiene parámetros se llama *constructor por defecto*. Normalmente, inicializa los miembros dato asignándoles valores por defecto. Así, por ejemplo, en el caso de la declaración de la clase Punto, el constructor Punto(), es llamado en el momento de la declaración de cada objeto, y ejecutándose en el mismo momento sus sentencia de inicialización:

```
Punto p1; // p1.x = 0 p1.y = 0 p1.z = 0
```

Regla: C++ crea automáticamente un constructor por defecto cuando no existen otros constructores. Sin embargo, tal constructor no inicializa los miembros dato de la clase a un valor previsible, de modo que siempre es conveniente al crear su propio constructor por defecto, darle la opción de inicializar los miembros dato con valores previsibles.

Precaución: Tenga cuidado con la escritura de la siguiente sentencia: Punto p();

Aunque parece que se realiza una llamada al constructor por defecto lo que se hace es declarar una función de nombre p que no tiene parámetros y devuelve un resultado de tipo Punto.

Constructores alternativos. Es posible pasar argumentos a un constructor asignando valores específicos a cada miembro dato de los objetos de la clase. Un constructor con parámetros se denomina *constructor alternativo*. Estos constructores son llamados en sentencias del tipo

```
Punto p2 = Punto( 5.0, 6.0, 7.0); // p2.x = 5.0 p2.y = 6.0 p3.z = 7.0
```

Constructor de copia. Existe un tipo especializado de constructor denominado *constructor de copia*, que se crea automáticamente por el compilador. El constructor de copia se llama automáticamente cuando un objeto se pasa por valor: se construye una copia local del objeto que se construye. El constructor de copia se llama también cuando un objeto se declara e inicializa con otro objeto del mismo tipo. Por ejemplo el código siguiente asigna a los atributos del objeto actual los atributos del objeto p.

```
Punto::Punto ( const Punto &p)
{
    x = p.x;
    y = p.y;
    z = p.z;
}
```

Constructor lista inicializadora. Inicialización de miembros. No está permitido inicializar un miembro dato de una clase cuando se define. La inicialización de los miembros dato de una clase se realiza en el **constructor especial lista inicializadora de miembros** que permite *inicializar* (en lugar de asignar) a uno o más miembros dato. Una lista inicializadora de miembros se sitúa inmediatamente después de la lista de parámetros en la definición del constructor. Consta de un carácter dos puntos, seguido por uno o más *inicializadores de miembro*, separados por comas. Un inicializador de miembros consta del nombre de un miembro dato seguido por un valor inicial entre paréntesis. Por ejemplo las declaración de constructor siguiente, inicializa x a valx, y a 0.0 así como z a 0.0.

```
Punto( float valx): x(valx), y(0.0), z(0.0){} // constructor
```

Sobrecarga de funciones miembro. Al igual que se puede sobrecargar una función global, se puede también sobrecargar el constructor de la clase o cualquier otra función miembro de una clase excepto el destructor (posteriormente se describirá el concepto de destructor, pero no se puede sobrecargar). De hecho los constructores sobrecargados son bastante frecuentes; proporcionan medios alternativos para inicializar objetos nuevos de una clase. Sólo un constructor se ejecuta cuando se crea un objeto, con independencia de cuántos constructores hayan sido definidos.

13.5. Destructores

En una clase se puede definir también una función miembro especial conocida como *destructor*, que se llama automáticamente siempre que se destruye un objeto de la clase. El nombre del destructor es el mismo que el nombre de la clase, precedida con el carácter ~. Al igual que un constructor, un destructor se debe definir sin ningún tipo de retorno (ni incluso void); al contrario que un constructor no puede aceptar parámetros, y además cada clase tiene sólo un destructor. El uso más frecuente de un destructor es liberar memoria que fue asignada por el constructor. Si un destructor no se declara explícitamente, C++ crea un vacío automáticamente.

EJEMPLO 13.3. Objeto Contador con destructor.

La sentencia c = Contador(); asigna a c el valor del constructor por defecto que en este caso es 0. Cada vez que se produzca un suceso el contador se incrementa en 1. El contador puede ser consultado para encontrar la cuenta actual.

```
#include <cstdlib>
#include <iostream>
using namespace std;
class Contador
```

```

{
private:
    unsigned int cuenta;                                // contar
public:
    Contador() {cuenta = 0;}                           // constructor
    void incrementa() {cuenta++;}                      // Iterador
    int valor()   {return cuenta;}                     //devuelve cuenta
    ~Contador() {}                                     // destructor
};

int main(int argc, char *argv[])
{
    Contador c;                                       // define e inicializa a cero

    for (int i = 0; i < 6; i++)
    {
        cout << " c = " << c.valor() << endl;
        c.incrementa();
        if(i == 2)
            c = Contador();
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 13.4. Clase pila de reales con funciones miembro para poner un elemento y sacar un elemento de la pila.

Una pila es un tipo de dato “en el cual el último en entrar es el primero en salir”. Las primitivas (operaciones) básicas tradicionales de gestión de una pila son poner y sacar. La primera añade un elemento a la pila y la segunda extrae un elemento borrándolo. En el ejemplo siguiente, la pila se implementa en un array `a` de longitud máxima de 100 reales. La zona privada de la clase pila contiene: la `cima` que apunta a la siguiente posición libre de la pila, en consecuencia la pila estará vacía cuando `cima` apunte a cero, y estará llena si la `cima` apunta a un número mayor o igual que 100; y el array `a` de 100 reales. La zona pública contiene el constructor, el destructor y las dos funciones miembro de gestión básica. Para poner un elemento en la pila, se coloca en la posición `a[cima]` y se incrementa `cima`. Para sacar un elemento de la pila, se decrementa `cima` en una unidad y se retorna el valor de `a[cima]`.

```

#include <cstdlib>
#include <iostream>
#include <string.h>
using namespace std;
#define max 100

class pila
{
private:
    int cima;
    float a[max];
public:
    pila(){cima = 0;};
    float sacar();
    void poner(float x);
    ~pila(){};
};

```

```

float pila::sacar()
{
    if(cima <= 0)
        cout << " error pila vacía ";
    else
        return a[--cima];
}

void pila::poner(float x)
{
    if(cima >= max)
        cout << " error pila llena";
    else
        a[cima++]= x;
}

int main(int argc, char *argv[])
{
    pila p;

    p.poner(6.0);
    p.poner(5.0);
    cout << p.sacar()<< endl;
    cout << p.sacar()<< endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

EJEMPLO 13.5. La clase empleado que contiene como miembros el nombre, la dirección y el sueldo; como funciones miembro poner y obtener cada uno de sus atributos así como Leerdatos() y Verdatos() que leen los datos del teclado y los visualizan en pantalla, respectivamente.

El nombre y la dirección se declaran como punteros a char, por lo que la asignación de memoria a los atributos se realiza dinámicamente, en la función miembro Leerdatos(). Se escribe un programa que utilice la clase, creando un array de tipo empleado y luego llenándolo con datos correspondientes a 50 empleados. Una vez llenado el array, se visualizan los datos de todos los empleados.

```

#include <cstdlib>
#include <iostream>
using namespace std;

class empleado
{
    char *nombre;
    char *direccion;
    float sueldo;
public:
    empleado(){
        nombre = new char[0]; nombre = "";
        direccion = new char[0]; direccion = "";
        sueldo = 0;
    }
    ~empleado(){}
    void Pnombre( char*s){ nombre = s; }

```

```

        void Pdireccion(char*s){direccion = s;}
        void Psueldo(float s){sueldo = s;}
        char * Onombre() { return nombre; }
        char* Odireccion(){ return direccion; }
        float Osueldo(){ return sueldo; }
        void leerdatos();
        void visualizardatos();
    };
    void empleado::leerdatos()
    {
        char nombre[50], *s;
        float su;

        cout << " introduzca nombre : ";
        cin.getline(nombre,49);
        s = new char(strlen(nombre)+1);           // memoria dinámica
        strcpy(s,nombre);
        Pnombre(s);
        cout << " introduzca ndireccion : ";
        cin.getline(nombre,49);
        s = new char(strlen(nombre)+1);
        strcpy(s,nombre);
        Pdireccion(s);
        cout << " introduzca sueldo : ";
        cin >> su;
        cin.getline(nombre,2);                     // limpiar buffer de entrada
        Psueldo(su);
    }

    void empleado::visualizardatos()
    {
        cout << " Nombre = " << Onombre() << endl;
        cout << " Direccion = " << Odireccion() << endl;
        cout << " sueldo = " << Osueldo() << endl << endl;
    }
}

int main(int argc, char *argv[])
{
    empleado Empleados[50];
    int i;

    for (i = 0; i < 50; i++)
        Empleados[i].leerdatos();
    for (i = 0; i < 50; i++)
        Empleados[i].visualizardatos();
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

13.6. Clase compuesta

Una *clase compuesta* es aquella clase que contiene miembros dato que son así mismo objetos de clases. Antes de crear el cuerpo de un constructor de una clase compuesta, se deben construir los miembros dato individuales en su orden de declaración.

EJEMPLO 13.6. Clase compuesta.

La clase Estudiante contiene miembros dato de tipo Expediente y Dirección:

```
class Expediente { // ... }
class Direccion { // ... };
class Estudiante
{
public :
    Estudiante()
    {
        PonerId(0);
        PonerNotaMedia(0.0);
    }
    void PonerId(long);
    void PonerNotaMedia(float);
private:
    long id;
    Expediente exp;
    Direccion dir;
    float NotMedia;
};
```

Aunque Estudiante contiene Expediente y Direccion, el constructor de Estudiante no tiene acceso a los miembros privados o protegidos de Expediente o Direccion. Cuando un objeto Estudiante sale fuera de alcance, se llama a su destructor. El cuerpo de ~Estudiante se ejecuta antes que los destructores de Expediente y Direccion. En otras palabras, el orden de las llamadas a destructores a clases compuestas es exactamente el opuesto al orden de llamadas de constructores.

13.7. Errores de programación

- Mal uso de palabras reservadas.** Es un error declarar una clase sin utilizar formalmente una de las palabras reservadas `class`, `struct` o `union` en la declaración.
- Uso de constructores y destructores.** No se puede especificar un tipo de retorno en un constructor o destructor, ni en su declaración ni en su definición. Tampoco se pueden especificar argumentos, incluso `void`, en la declaración o definición del destructor de una clase. Un constructor de una clase C++ no puede esperar un argumento de tipo C.
- Inicializadores de miembros.**

```
class C {
    // ERROR
    int x = 5;
};

class C {
    int x;
    ...
public:
    C() = {x = 5;}           // CORRECTO
};
```

- No se puede utilizar `this` como parámetro, ya que es una palabra reservada (`this` es un puntero al propio objeto). Tampoco se puede utilizar en una sentencia de asignación tal como:

```
this = ...;           // this es constante; ERROR
```

5. **Olvido de puntos y coma en definición de clases.** Las llaves {} son frecuentes en código C++, y normalmente no se sitúa un punto y coma después de la llave de cierre. Sin embargo la definición de class siempre termina en }. Un error típico es olvidar ese punto y coma.
6. **Olvido de punto y coma en prototipos y cabeceras de funciones.** La omisión de un punto y coma al final del prototipo de una función puede producir el mensaje de error .

EJERCICIOS

13.1. ¿Cuál es el error de la siguiente declaración de clase?

```
union float_bytes
{
    private:
        char mantisa[3], char exponente ;
    public:
        float num;
        char exp(void);
        char mank(void);
};
```

13.2. Examine la siguiente declaración de clase y vea si existen errores.

```
class punto {
    int x, y;
    void punto(int x1, int y1) {x = x1 ; y = y1;}
};
```

13.3. Dado el siguiente programa, ¿es legal la sentencia de main()?

```
class punto {
public:
    int x;
    int y;
    punto(int x1, int y1) {x = x1 ; y = y1;}
};

main()
{
    punto(25, 15);           //¿es legal esta sentencia?
}
```

13.4. Dadas las siguientes declaraciones de clase y array. ¿Por qué no se puede construir el array?

```
class punto {
public:
    int x, y;
    punto(int a, int b) {x = a ; y = b;}
};

punto poligono[5];
```

13.5. Se cambia el constructor del ejercicio anterior por un constructor por defecto. ¿Se puede construir, en este caso, el array?

13.6. ¿Cuál es el constructor que se llama en las siguientes declaraciones?

```
class prueba_total
{
    private:
        int num;
    public:
        prueba_total(void) {num = 0;}
        prueba_total(int n) {num = n;}
        int valor(void) {return num;}
};
prueba_total prueba
```

13.7. ¿Cuál es la diferencia de significado entre la estructura?

```
struct a
{
    int i, j, k;
};
```

y la clase

```
class
{
    int i, j, k;
};
```

Explique la razón por la cual la declaración de la clase no es útil. ¿Cómo se puede utilizar la palabra reservada `public` para cambiar la declaración de la clase en una declaración equivalente a `struct`?

PROBLEMAS

13.1. Crear una clase llamada `hora` que tenga miembros `datos` separados de tipo `int` para horas, minutos y segundos. Un constructor inicializará este dato a 0, y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo `hora` pasados como argumentos. Una función principal `main()` crea dos objetos inicializados y uno que no está inicializado. Sumar los dos valores inicializados y dejar el resultado en el objeto no inicializado. Por último, visualizar el valor resultante.

13.2. Un número complejo tiene dos partes: una parte real y una parte imaginaria; por ejemplo, en $(4.5+3.0i)$, 4.5 es la parte real y 3.0 es la parte imaginaria. Realizar una clase `Complejo` que permita la gestión de números complejos (un número complejo = dos números reales). Las operaciones a implementar son las siguientes:

- Una función `leerComplejo()` permite leer un objeto de tipo `Complejo`.
- Una función `escribirComplejo()` realiza la visualización formateada de un `Complejo`.

13.3. Suponga que $a = (A, Bi)$ y $c = (C, Di)$. Se desea añadir a la clase complejo las operaciones:

- **Suma:** $a + c = (A + C, (B + D)i)$.
- **Resta:** $a - c = (A - C, (B - D)i)$.
- **Multiplicación:** $a * c = (A * C - B * D, (A * D + B * C)i)$.
- **Multiplicación:** $x * c = (x * C, x * Di)$, donde x es real.
- **Conjugado:** $\sim a = (A, -Bi)$.

13.4. Escribir una clase Conjunto que gestione un conjunto de enteros (`int`) con ayuda de una tabla de tamaño fijo (un conjunto contiene una lista ordenada de elementos y se caracteriza por el hecho de que cada elemento es único: no se debe encontrar dos veces el mismo valor en la tabla). Las operaciones a implementar son las siguientes:

- La función `vacía()` vacía el conjunto.
- La función `agregar()` añade un entero al conjunto.
- La función `eliminar()` retira un entero del conjunto.
- La función `copiar()` recopila un conjunto en otro.
- La función `es_miembro()` reenvía un valor booleano (valor lógico que indica si el conjunto contiene un entero dado).
- La función `es_igual()` reenvía un valor booleano que indica si un conjunto es igual a otro.
- La función `imprimir()` realiza la visualización “formateada” del conjunto.

13.5. Añadir a la clase Conjunto del ejercicio anterior las funciones `es_vacio`, y `cardinal` de un conjunto, así como, la unión, intersección, diferencia y diferencia simétrica de conjuntos.

SOLUCIÓN DE LOS EJERCICIOS

13.1. Es un error declarar una clase sin utilizar fielmente una de las palabras reservadas `class`, `struct` o `union` en la declaración. En este caso se ha utilizado la palabra `union`, por lo que no es un error, pero sí que es un error la declaración `char mantisa[3]`, `char exponente` ya que o bien se ha cambiado la coma por un punto y coma, o bien sobre la palabra reservada `char` segunda.

Una declaración correcta puede ser:

```
union float_bytes
{
    private:
        char mantisa[3];
        char exponente;
    public:
        float num;
        char exp(void);
        char mank(void);
};
```

13.2. Esta declaración tiene varios errores. El primero es declarar un constructor con un tipo `void`. Los constructores no tienen tipo. El segundo es que los constructores deben estar en la zona pública de una clase, y no en la privada. Una declaración correcta es, por ejemplo, la siguiente:

```
class punto
{
```

```

int x, y;
public:
    punto(int x1, int y1) {x = x1 ; y = y1;}
};

```

- 13.3.** La sentencia es sintácticamente correcta, ya que se realiza una llamada al constructor de `punto`, pero hay que tener en cuenta que el constructor, retorna un objeto del tipo `punto` y no es recogido por ninguna instancia de la clase. Sería conveniente, haber realizado una sentencia del tipo siguiente: `punto p = punto(25, 15);`

- 13.4.** El array no puede construirse, porque al hacer la declaración, necesariamente se llama al constructor que en este caso tiene dos parámetros de tipo entero, y en la declaración no puede realizarse. Para resolver el problema, puede optarse por declarar explícitamente el constructor de `punto` sin ningún parámetro. Por ejemplo podría ser de la siguiente forma:

```

class punto
{
    public:
        int x, y;
        punto(int a, int b) {x = a ; y = b;}
        punto(){}
};

punto poligono[5];

```

- 13.5.** Al desaparecer el constructor con dos parámetros, el constructor por defecto entra en acción y, por tanto, sí que es correcto, tanto si se incluye como si no se hace en la declaración de la clase.

- 13.6.** El constructor llamado es el definido por defecto, que en este caso es el que no tiene parámetro: `prueba_total(void)`.

- 13.7.** La única diferencia es que `i, j, k` son públicos en la estructura, y privados en la clase, por lo que la declaración de la clase no es útil, ya que sólo puede accederse a los atributos `i, j, k`, mediante funciones miembros que no tiene. Para hacer equivalentes ambas declaraciones basta con añadir la palabra reservada `public` tal y como se expresa a continuación:

```

struct a
{
    int i, j, k;
};

class
{
    public:
        int i, j, k;
};

```

SOLUCIÓN DE LOS PROBLEMAS

- 13.1.** La codificación de este problema se encuentra en la página Web del libro.

- 13.2.** La clase `Complejo` es implementada con sus dos atributos, parte real `r`, y parte imaginaria `i`. Tiene dos constructores, uno por defecto que inicializa a cero tanto la parte real como la imaginaria, y otro que inicializa el número complejo con sus dos parámetros correspondientes. Las funciones miembro, `Or` y `Pr`, se encargan de obtener y poner respectivamente la par-

te real de un número complejo. Por su parte las funciones O_i , y P_i , obtienen y ponen la parte imaginaria. Las funciones de leerComplejo y escribirComplejo, usan las funciones miembro anteriores para tratar los atributos que implementan la parte real y la parte imaginaria del número complejo.

La codificación de este problema se encuentra en la página Web del libro.

- 13.3. A la clase del problema anterior se le añaden las funciones miembro indicadas. Observar la sobrecarga de la función multiplicación.

La codificación de este problema se encuentra en la página Web del libro.

- 13.4. Se representa el Conjunto con atributos Numelementos, que indica el número de elementos que tiene el conjunto, así como con un array L, de enteros que almacena hasta un valor máximo de 512 números enteros, declarado en una macro. Los datos se almacenan en el arr ay en orden creciente. Para facilitar la inserción y eliminación de elementos en el conjunto, se programa una función miembro no pública Buscar encargada de buscar si un elemento que se le pasa como parámetro se encuentra en el conjunto actual, indicando además en caso de que se encuentre la posición en la que se halla para poder borrarlo, y en caso de que no se encuentre la posición donde habría que añadirlo, en caso de que fuera necesario. Esta función se programa en tiempo logarítmico, mediante una búsqueda binaria clásica, tal y como se ha explicado en el capítulo de ordenación. El resto de las funciones miembros se programan de forma natural. También se incluyen un constructor y un destructor.

La codificación de este problema se encuentra en la página Web del libro.

- 13.5. Se añaden a la clase Conjunto las funciones solicitadas, incluyendo solamente la codificación.

- Un conjunto está vacío si no tiene elementos, y el número de elementos de un conjunto viene dado por el valor de su atributo Numelementos.
- La función union se implementa basándose en la mezcla o unión de dos arrays de números enteros, teniendo en cuenta que deben aparecer los elementos de los dos arrays, y que sólo puede aparecer el elemento una vez. Siempre que se añade un nuevo elemento a la unión se comprueba que no se superpasa la capacidad del conjunto.
- La función intersección que se implementa se basa en realizar la mezcla de dos listas o conjuntos incluyendo los elementos que están en las dos listas.
- La función Diferencia añade al conjunto los elementos que estén en el conjunto c1 y no estén en el conjunto c2, realizando un recorrido del conjunto c1.
- La función diferenciaSimétrica, se implementan usando las propiedades de los conjuntos.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 13.1. Considerar una pila como un tipo abstracto de datos. Se trata de definir una clase que implementa una pila de 100 caracteres mediante un array. Las funciones miembro de la clase deben ser: meter, sacar, pilavacia y pilallena.
- 13.2. Clasificar los siguientes constructores cuyos prototipos son:

- a. rect::rect(int a, int h);
- b. nodo::nodo(void);
- c. calculadora::calculadora (float *val i = 0.0);

- d. cadena::cadena(cadena &c);
- e. abc::abc(float f);

- 13.3. Crear una clase llamada empleado que contenga como miembro dato el nombre y el número de empleado, y como funciones miembro Leerdatos() y Verdatos() que lean los datos del teclado y los visualice en pantalla, respectivamente. Escribir un programa que utilice la clase, creando un array de tipo empleado y luego llenándolo con datos correspondientes a 50 empleados. Una vez llenado el array, visualizar los datos de todos los empleados.

PROBLEMAS PROPUESTOS

- 13.1. Se desea realizar una clase `vector3d` que permita manipular vectores de tres componentes (coordenadas x , y , z) de acuerdo a las siguientes normas:
- Sólo posee una función constructor y es en línea.
 - Tiene una función miembro igual que permite saber si dos vectores tienen sus componentes o coordenadas iguales (la declaración de igual se realizará utilizando:*a*) transmisión por valor; *b*) transmisión por dirección; *c*) transmisión por referencia).
- 13.2. Incluir en la clase `vector3d` del ejercicio anterior una función miembro denominada `normamax` que permita obtener la norma mayor de dos vectores (*Nota*: La norma de un vector $v = (x, y, z)$ es $\sqrt{x^2+y^2+z^2}$).
- 13.3. Añadir a la clase `vector3d` las funciones miembros `suma` (suma de dos vectores), `productoescalar` (producto escalar de dos vectores: $v1 = (x1, y1, z1); v2 = (x2, y2, z2); v1 * v2 = x1 * x2 + y1 * y2 + z1 * z2$).
- 13.4. Crear una clase `lista` que realice las siguientes tareas:
- Una lista simple que contenga cero o más elementos de algún tipo específico.
 - Crear una lista vacía.
 - Añadir elementos a la lista.
- 13.5. Determinar si la lista está vacía.
- 13.6. Implementar una clase `Random` (aleatorios) para generar números pseudoaleatorios.
- 13.7. Implementar una clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, un constructor de copia, funciones de acceso, una función `reiniciar` (`int d, int m, int a`) para reiniciar la fecha de un objeto existente, una función `adelantar`(`int d, int m, int a`) para avanzar a una fecha existente (dia, d, mes, m, y año a) y una función `imprimir()`. Emplear una función de utilidad `normalizar()` para asegurar que los miembros dato están en el rango correcto
 $1 \leq \text{año}, 1 \leq \text{mes} \leq 12, \text{día} \leq \text{días(Mes)}$, donde `días(Mes)` es asegurar otra función que devuelve el número de días de cada mes.
- 13.8. Ampliar el programa anterior de modo que pueda aceptar años bisiestos. (*Nota*: un año es bisiesto si es divisible por 400, o si es divisible por 4 pero no por 100. Por ejemplo el año 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.)

CAPÍTULO 14

Herencia y polimorfismo

Introducción

En este capítulo se introduce el concepto de *herencia* y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base las clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, bajo coste, adaptable y reutilizable.

El *polimorfismo* permite que diferentes objetos respondan de modo diferente al mismo mensaje; adquiere su máxima potencia cuando se utiliza en unión de herencia y hace los sistemas más flexibles, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación.

14.1. Herencia

La propiedad que permite a los objetos ser construidos a partir de otros objetos se denomina **herencia**. Dicho de otro modo, la capacidad de un objeto para utilizar las estructuras de datos y los métodos previstos en antepasados o ascendientes. El objetivo final es la **reutilización**, es decir, reutilizar código anteriormente ya desarrollado.

La herencia supone una *clase base* y una *jerarquía de clases* que contienen las *clases derivadas* de la clase base. Las clases derivadas pueden heredar el código y los datos de su clase base añadiendo su propio código especial y datos a ellas, incluso cambiar aquellos elementos de la clase base que necesitan ser diferentes.

Las clases derivadas heredan características de su clase base, pero añaden otras características propias nuevas. En las jerarquías de generalización/especialización la clase más alta constituye la *generalización*, mientras que la clase de más bajo nivel constituye una *especialización*. La implementación de estas jerarquías generalización/especialización da lugar a la herencia (véase la Figura 14.1).

Una clase *hereda* sus características (datos y funciones) de otra clase.

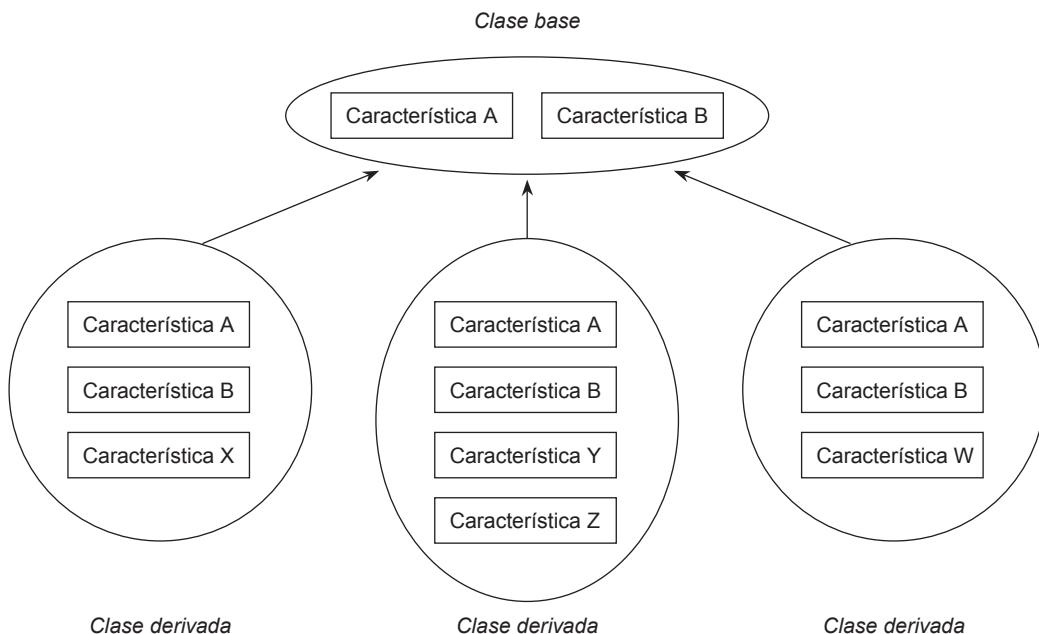


Figura 14.1. Jerarquía de clases.

Las clases derivadas pueden heredar el código y los datos de su clase base, añadiendo características propias nuevas. Las jerarquías de clases se organizan en forma de árbol invertido. Así, se puede decir que una clase de objetos es un conjunto de objetos que comparten características y comportamientos comunes. Estas características y comportamientos se definen en una clase base. Las clases derivadas se crean en un proceso de definición de nuevos tipos y reutilización del código anteriormente desarrollado en la definición de sus clases base. Este proceso se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden, a su vez, servir como definiciones base de otras clases. Las jerarquías de clases se organizan en forma de árbol.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el especificador de acceso que indica el tipo de herencia (*pública*, *privada* y *protegida*).

```

Nombre de la clase derivada      Especificador de acceso
                                Normalmente público
                                Herencia
                                ↓
class ClaseDerivada : public ClaseBase {
    public:
        // sección pública
    ...
    private:
        // sección privada
    ...
};
```

Nombre de la clase base

Símbolo de derivación o herencia

Especificador de acceso public, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

Herencia pública es aquella en que el especificador de acceso es *public* (público). En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y privados de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos.

Herencia protegida es aquélla en que el especificador de acceso es `protected` (protegido). Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles.

Herencia privada es aquélla en que el especificador de acceso es `private` (privado). Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada.

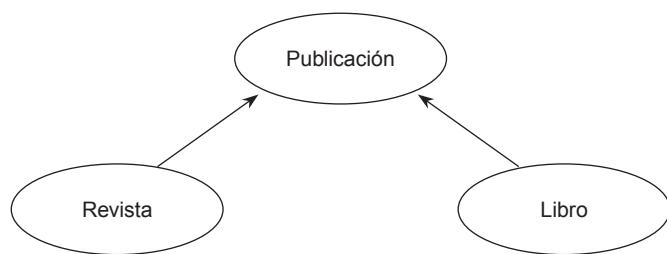
El especificador de acceso que declara el tipo de herencia es opcional (`public`, `private` o `protected`); si se omite el especificador de acceso, se considera por defecto `private`. La *clase base* (*ClaseBase*) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```
class nombre_clase: especificador de acceso opcional ClaseBase {
    lista de miembros;
};
```

Tabla 14.1. Tipos de herencia y acceso que permiten

Tipo de herencia	Acceso a miembro clase base	Acceso a miembro a clase derivada
public	public protected private	public protected inaccesible
protected	public protected private	protected protected inaccesible
private	public protected private	private private inaccesible

EJEMPLO 14.1. Representar la jerarquía de clases de publicaciones que se distribuyen en una librería: revistas y libros.



Todas las publicaciones tienen en común una editorial y una fecha de publicación. Las revistas tienen una determinada periodicidad lo que implica el número de ejemplares que se publican al año y, por ejemplo, el número de ejemplares que se ponen en circulación controlados oficialmente. Los libros, por el contrario tienen un código de ISBN y el nombre del autor.

```
#include <cstdlib>
#include <iostream>
using namespace std;
class Publicacion
{
public:
    void NombrarEditor(const char *s){strcpy(editor,s);}
    void PonerFecha(unsigned long fe){fecha = fe;}
    void EscribirEditor(){cout << editor << endl;}
    void EscribirEditro(){cout << fecha << endl;}
};
```

```

private:
    char editor[50];
    unsigned long fecha;
};

class Revista :public Publicacion
{
public:
    void NumerosPorAnyo(unsigned n){ numerosPorAnyo = n;}
    void FijarCirculacion(unsigned long n){ circulacion = n;}
    void EscribirNumerosPorAnyo(){ cout << numerosPorAnyo << endl;}
    void EscribirCirculacion(){ cout << circulacion << endl;}
private:
    unsigned numerosPorAnyo;
    unsigned long circulacion;
};

class Libro :public Publicacion
{
public:
    void PonerISBN(const char *s){ strcpy(ISBN,s);}
    void PonerAutor(const char *s){ strcpy(autor,s);}
    void EscribirISBN(){ cout << ISBN << endl;}
    void EscribirAutor(){ cout << autor << endl;}
private:
    char ISBN[20];
    char autor[40];
};

int main(int argc, char *argv[])
{
    Libro L;
    Revista R;
    L.NombrarEditor("McGraw-Hill serie Schaum");
    L.PonerFecha(23052005);
    L.PonerISBN("84-481-4514-3");
    L.PonerAutor("Sanchez, García Lucas");
    R.NombrarEditor("McGraw-Hill");
    R.PonerFecha(1106005);
    R.NumerosPorAnyo(12);
    R.FijarCirculacion(20000);
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

14.1.1. TIPOS DE HERENCIA

Existen dos mecanismos de herencia utilizados comúnmente en programación orientada a objetos: *herencia simple* y *herencia múltiple*. En C++ la herencia se conoce como *heredación*.

La Figura 14.2 representa los gráficos de herencia simple y herencia múltiple de la clase figura y persona, respectivamente.

En herencia simple, un objeto (*clase*) puede tener sólo un ascendiente, o un solo parente. La herencia simple permite que una clase herede las propiedades de su superclase en una cadena de jerarquía. La sintaxis de la herencia simple:

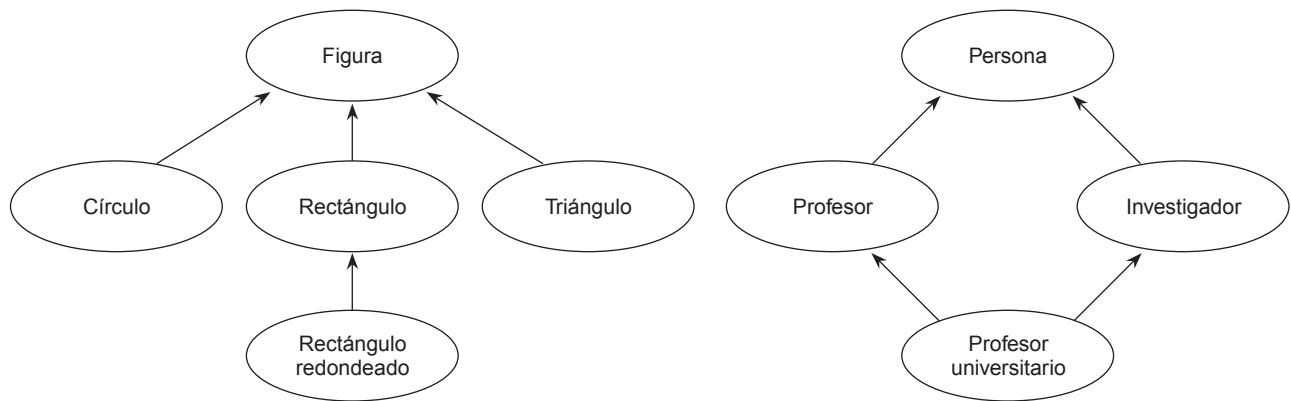


Figura 14.2. Tipos de herencia.

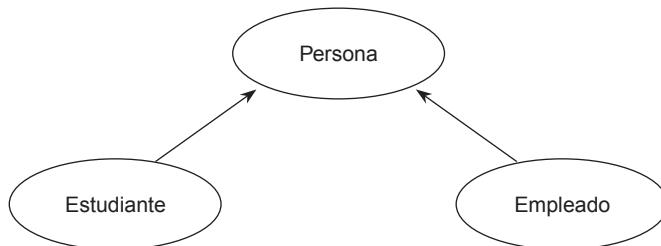
```

class Base { ... }
class Derivada : [acceso] Base { ... }
  
```

el modificador de acceso indica el tipo de visibilidad, si bien el más usado es siempre `public`, utilizándose otros tipos de acceso para el caso de la reutilización del *software*.

EJEMPLO 14.2. *Herencia simple.*

La clase `estudiante` hereda de la clase `persona` el atributo `nombre`, y añade el atributo `estudios`, pudiendo utilizar las funciones miembro `ponernombre` y `obtenernombre`. La clase `empleado` hereda de la clase `persona` el `nombre` y añade el atributo `sueldo`.



```

#include <cstdlib>
#include <iostream>

using namespace std;

class persona
{
protected:
    char nombre[40];
public:
    persona(){}
    void ponernombre(char *s){strcpy(nombre,s);}
    char * obtenernombre() {return nombre;}
};

class estudiante: public persona
  
```

```

{
protected:
    char estudios[30] ;
public:
    estudiante(){}
    void ponerestudios( char *c){strcpy(estudios,c);}
    char *obtenerestudios() {return estudios;}
};

class empleado : public persona
{
private:
    float sueldo;
public:
    empleado(){}
    void ponersueldo( float r) { sueldo = r;}
    float obtensueldo(){return sueldo;};
};

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

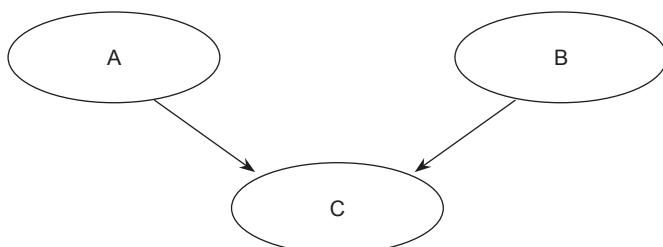
La herencia múltiple es la propiedad de una clase de poder tener más de un ascendiente inmediato, o más de un solo parente. Una herencia múltiple es aquella en la que cada clase puede heredar métodos y variables de cualquier número de superclases. La herencia múltiple es en esencia la transmisión de métodos y datos de más de una clase base a la clase derivada. La sintaxis de la herencia múltiple es igual a la empleada en la herencia simple, con la única diferencia de que se enumeran las diferentes clases base de las que depende, en lugar de una sola. La herencia múltiple plantea diferentes problemas tales como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación o preponderancia* de funciones o datos.

Problemas que se pueden presentar cuando se diseñan clases con herencia múltiple son:

- **colisiones de nombres** de diferentes clases base (dos o más clases base tienen el mismo identificador para algún elemento de su interfaz. Se resuelve con el operador de ámbito ::);
- **herencia repetida** de una misma clase base (una clase puede heredar indirectamente dos copias de una clase base. Se resuelve con el operador de ámbito ::);

EJEMPLO 14.3. Herencia múltiple sin problemas de colisión de nombres o de herencia repetida.

La clase C hereda de dos clases A y B, los atributos y los métodos. En concreto, los atributos entero de la clase A y letra de la clase B. Para activar el constructor de la clase A y B en el momento de crear el objeto de la clase derivada C, se incluye expresamente el constructor de la clase base A y B en la lista de inicialización del constructor de la clase derivada C.



```
#include <cstdlib>
#include <iostream>

using namespace std;

class A
{
protected:
    int entero;
public:
    A(){}
    A(int i){entero = i;} // constructor por defecto
    void ponerentero( int i){ entero = i;} // constructor alternativo
    int obtenerentero() {return entero;}
};

class B
{
protected:
    char letra;
public:
    B(){}
    B(char c){letra = c;} // constructor por defecto
    void ponerletra( char c){ letra = c;} // constructor alternativo
    char obtenerletra() {return letra;}
};

//La Clase C hereda de las dos clases anteriores A y B

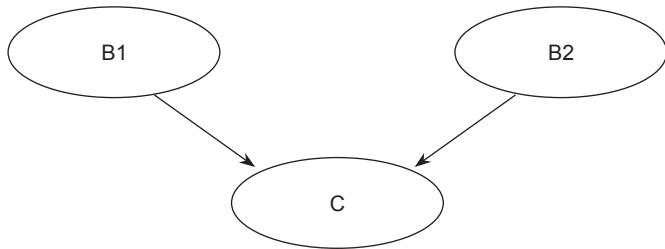
class C : public A, public B
{
private:
    float real;
public:
    C(){}
    C(int, char, float); // constructor por defecto
    void ponerreal( float r) { real = r;} // constructor alternativo
    float obtenerreal(){return real;};
};

C::C(int i, char c, float f) : A(i), B(c)
{
    real = f;
}

int main(int argc, char *argv[])
{
    C c1( 23,'C',24.5);
    cout << c1.obtenerentero()<<endl;
    cout << c1.obtenerletra()<<endl;
    cout << c1.obtenerreal()<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

EJEMPLO 14.4. *Herencia múltiple con colisión de nombre. Repetición de un atributo en las clases base y derivada.*

Las clases B1 y B2 tienen un atributo entero x. La clase derivada C es derivada de las clases B1 y B2, por lo que tiene dos atributos x asociados a las clases: C hereda de B1; y C hereda de B2. La clase C hereda públicamente de la clase B1 y B2, por lo que puede acceder a los tres atributos con el mismo nombre x. Estos atributos son: uno local a C, y otros dos de B1 y B2. La función miembro `verx` puede retornar el valor de x, pero puede ser el de B1, B2, C resuelto con el operador de ámbito `::`. La función miembro `probar` es pública y amiga de la clase C, por lo que puede modificar los tres distintos atributos x de cada una de las clases. Para resolver el problema del acceso a cada uno de ellos se usa el operador de resolución de ámbito `::`:



```

#include <cstdlib>
#include <iostream>

class B1
{
protected:
    int x;
};

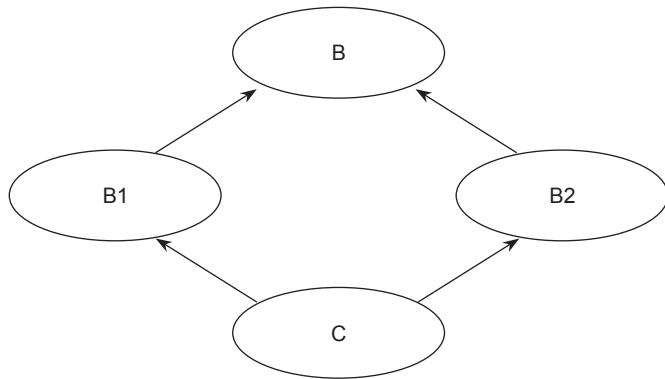
class B2
{
protected:
    int x;
};

class C: public B1,B2
{
protected:
    int x;
public:
    friend void probar(); // es amiga y tiene acceso a x
    int verx()
    {
        return x; // x de C
        return B1::x; // x de B1
        return B2::x; // x de B2
    }
};

void probar()
{
    C c1;
    c1.x = 12; // atributo x de C
    c1.B1::x = 123; // atributo x de B1 resolución de colisión
    c1.B2::x = 143; // atributo x de B2 resolución de colisión
}
  
```

EJEMPLO 14.5. *Herencia múltiple repetida. Repetición de una clase a través de dos clases derivadas.*

Las clases B1 y B2 tienen un mismo atributo entero x. La clase derivada C también tiene el mismo atributo entero x. La clase C hereda públicamente de la clase B1 y B2, por lo que puede acceder a los tres atributos con el mismo nombre x. La función miembro fijax fija los dos atributos x resolviendo el problema con el operador de ámbito ::.



```

#include <cstdlib>
#include <iostream>

class B
{
protected:
    int x;
};

class B1: public B
{
protected:
    float y;
};

class B2: public B
{
protected:
    float z;
};

class C: public B1,B2
{
protected:
    int t;
public:
    void fijax( int xx1, int xx2)
    {
        B1::x=xx1;
        B2::x=xx2;
    }
};
  
```

14.1.2. CLASES ABSTRACTAS

Una clase abstracta normalmente ocupa una posición adecuada en la jerarquía de clases que le permite actuar como un depósito de métodos y atributos compartidos para las subclases de nivel inmediatamente inferior. Las clases abstractas definen un tipo generalizado y sirven solamente para describir nuevas clases.

Las clases abstractas no tienen instancias directamente. Se utilizan para agrupar otras clases y capturar información que es común al grupo. Sin embargo, las subclases de clases abstractas que corresponden a objetos del mundo real pueden tener instancias. Las superclases creadas a partir de subclases con atributos y comportamientos comunes, y empleadas para derivar otras clases que comparten sus características, son **clases abstractas**.

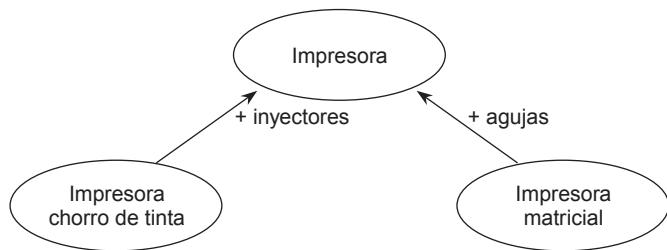
En C++, una clase abstracta tiene al menos una función miembro que se declara pero no se define; su definición se realiza en la clase derivada. Estas funciones miembro se denominan funciones virtuales puras, cuya definición formal es:

```
virtual tipo nombrefuncion(argumentos) = 0;
```

Las siguientes reglas se aplican a las clases abstractas:

- Una clase abstracta debe tener al menos una función virtual pura.
- Una clase abstracta no se puede utilizar como un tipo de argumento o como un tipo de retorno de una función aunque sí un puntero a ella.
- No se puede declarar una instancia de una clase abstracta.
- Se puede utilizar un puntero o referencia a una clase abstracta.
- Una clase derivada que no proporcione una definición de una función virtual pura, también es una clase abstracta.
- Cada clase (derivada de una clase abstracta) que proporciona una definición de todas sus funciones virtuales es una clase concreta.
- Sólo está permitido crear punteros a las clases abstractas y pasárselos a funciones.

EJEMPLO 14.6. Una clase abstracta puede ser una impresora.



```
#include <cstdlib>
#include <iostream>
using namespace std;

class impresora
{
public:
    virtual int arrancar() = 0;
};

class ChorroTinta: public impresora
{
protected:
    float chorro;
public:
    int arrancar() { return 100; }
};
```

```

class matricial: public impresora
{
protected:
    int agujas;
public:
    int arrancar( ) { return 10; }
};

int main(int argc, char *argv[])
{
//impresora im;      no puede declararse es abstracta
matricial m;
ChorroTinta ch;
system("PAUSE");
return EXIT_SUCCESS;
}

```

14.2. Ligadura

La **ligadura** representa, generalmente, una conexión entre una entidad y sus propiedades. Desde el punto de vista de los atributos, la ligadura es el proceso de asociar un atributo a su nombre. En las funciones, la ligadura es la conexión entre la llamada a la función y el código que se ejecuta tras la llamada. El momento en que se asocia es el *tiempo de ligadura*.

La ligadura se clasifica según sea el tiempo de ligadura en: *estática* y *dinámica*.

- En un programa con **ligadura estática**, todas las referencias y, en general cualquier llamada a una función, se resuelven en tiempo de compilación. El compilador define directamente la posición fija del código que se ha de ejecutar en cada llamada a función. En C++ por defecto la ligadura es estática.
- La **ligadura dinámica** supone que el código a ejecutar en una llamada a un función no se determinará hasta el momento de ejecución. Normalmente, el valor de un puntero, o una referencia, determina la ligadura efectiva entre las posibilidades de cada clase derivada. En C++ la ligadura dinámica se produce con las funciones virtuales.

14.3. Funciones virtuales

Una función miembro de una clase se dice que es virtual, si la palabra reservada `virtual` precede a la declaración de una función. Esta palabra reservada le indica al compilador que la función puede ser redefinida en una clase derivada.

Los usos comunes de las funciones virtuales son: declaración de una clase abstracta (funciones virtuales puras); implementación del polimorfismo.

14.4. Polimorfismo

El *polimorfismo* permite que diferentes objetos respondan de modo diferente al mismo mensaje. El polimorfismo adquiere su máxima potencia cuando se utiliza en unión de herencia. El polimorfismo permite referirse a objetos de diferentes clases por medio del mismo elemento y realizar la misma operación de formas diferentes, de acuerdo al objeto a que se hace referencia en cada momento. Un ejemplo típico es la operación `arrancar` cuando se aplica a diferentes tipos de motores. En cada caso la operación de arrancar se realiza de forma diferente.

EJEMPLO 14.7. Polimorfismo y funciones virtuales.

Si `Polygon` es una clase base de la que cada figura geométrica hereda características comunes, C++ permite que cada clase utilice funciones o métodos `Area`, `Visualizar`, `Perimetro`, `PuntoInterior` como nombre de una función miembro de las clases derivadas. En estas clases derivadas donde se definen las funciones miembro.

```

class Poligono
{
    // superclase
public:
    virtual float Perimetro();
    virtual float Area();
    virtual bool PuntoInterior();
    virtual void Visualizar();

};

// la clase Rectángulo debe definir las funciones virtuales que use de
// la clase polígono
class Rectangulo : public Poligono
{
private:
    float Alto, Bajo, Izquierdo, Derecho;
public:
    float Perimetro();
    float Area();
    bool PuntoInterior();
    void Visualizar();
    void fijarRectangulo();

};

// la clase Triángulo debe definir las funciones virtuales que use de
// la clase polígono

class Triangulo : public Poligono{
    float uno, dos, tres;
public:
    float Area();
    bool PuntoInterior();
    void fijarTriangulo();
}

```

EJERCICIOS

14.1. ¿Cuál es la salida del siguiente programa que contiene funciones virtuales y no virtuales en la clase base?

```

#include <cstdlib>
#include <iostream>
using namespace std;

class base
{
public:
    virtual void f(){ cout << "f(): clase base " << endl;}
    void g() {cout << "g(): clase base " << endl;}
};

```

```

class Derivada1 : public base
{
public:
    virtual void f(){cout << "f(): clase Derivada 1 " << endl;}
    void g() {cout << "g(): clase Derivada 1!" << endl;}
};

class Derivada2 : public Derivada1
{
public:
    void f() {cout << "f(): clase Derivada 2 !" << endl;}
    void g() {cout << "g(): clase Derivada 2 !" << endl;}
};

int main(int argc, char *argv[])
{
    base b;
    Derivada1 d1;
    Derivada2 d2;
    base *p;

    p = &b;      p ->f();      p ->g();
    p = &d1;     p ->f();      p ->g();
    p=&d2;     p ->f();      p ->g();
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

- 14.2.** Declarar las clases *Punto unidimensional*, *Punto bidimensional* y *punto tridimensional*, que permitan tratar puntos en el espacio de dimensión uno, dos y tres respectivamente, mediante una jerarquía de clases.
- 14.3.** Declarar la clase abstracta *Figura* con las funciones virtuales *Área* y *perímetro* y que tenga como clases derivadas las figuras geométricas *círculo*, *cuadrado* y *rectángulo*.
- 14.4.** Diseñar una jerarquía de clases con la clase base *Deportes*, de la que se derivan las clases *Fútbol*, *Baloncesto*, *Voleibol* y *Ball*, que tenga una ligadura dinámica.

PROBLEMAS

- 14.1.** Definir la clase *objeto geométrico* y las clases *círculo* y *cuadrado* que hereden públicamente las coordenadas del centro de un círculo y de un cuadrado respectivamente. La clase *círculo* deberá permitir calcular la longitud de la circunferencia y el área del círculo de radio dado. La clase *cuadrado* debe permitir calcular el perímetro y el área del cuadrado conocido además un vértice del cuadrado.
- 14.2.** Definir la clase *persona* con los atributos *EdadPersona*, *estado_civil*, *sexo*, *NombrePersona*, *direccion* y las clases *alumno* y *profesor* derivadas de la anterior y que añadan el atributo *departamento* para el caso del profesor y los atributos *curso* y *Num_asignaturas* para la clase *alumno*. Añadir los constructores de cada una de las clases y una función miembro visualizar a cada una de las clases.
- 14.3.** Escribir un programa que permita leer mediante un menú un vector de personas que puedan ser alumnos o profesores. Usar el Problema resuelto 14.2.

SOLUCIÓN DE LOS EJERCICIOS

- 14.1.** La clase base tiene dos funciones una virtual *f* que es redefinida en las dos clases derivadas y una no virtual *g* que se vuelve a redefinir en las clases derivadas. Con las tres primeras sentencias se llama a las funciones de la clase base. Con las siguientes tres sentencias se llama a la función virtual redefinida en la clase *Derivada1* y la función de la clase base *g*. En las tres siguientes llamadas se ejecuta la función virtual *f* definida en la clase *Derivada2* y a la función *g* de la clase base. El resultado de ejecución del programa se muestra a continuación. Si se quiere que se ejecuten la función *g* de las clases *Derivada1* y *Derivada2*, hay que optar escribir *d1.g()* o *d2.g()*, ya que en las asignaciones *p = &d1*, o *p = &d2*, como *p* es un puntero a la clase base, entonces la referencia *p->g()* accede a la función no virtual de la clase base.
- 14.2.** La jerarquía de clases es prácticamente obvia. La clase *Punto1D*, sólo tiene un atributo que es la coordenada *x* de un punto unidimensional. La clase *Punto2D* hereda el atributo *x* y las funciones miembro de la clase *Punto1D*. Además, esta clase añade el atributo *y*, así como las funciones miembro correspondientes. La clase *Punto3D* hereda los atributos *x* e *y* de la clase *Punto2D* así como todas sus funciones miembro, y añade el atributo *z*.

La codificación de este ejercicio se encuentra en la página Web del libro.

- 14.3.** La clase abstracta *figura* sólo tiene el constructor y la declaración de las funciones virtuales *perímetro* y *Área*. Las clases derivadas *círculo*, *cuadrado*, y *rectángulo* añaden los atributos, *radio*, *lado*, *largo* y *ancho* a cada una de las clases. Se definen las funciones virtuales *perímetro* y *Área* así como los constructores correspondientes y las correspondientes funciones miembro para manipular los atributos de cada una de las clases derivadas. Se incluye además un programa principal y un resultado de ejecución de dicho programa principal.

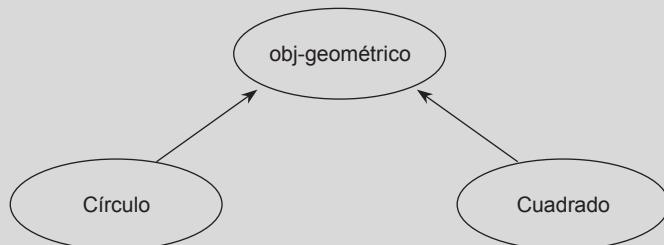
La codificación de este ejercicio se encuentra en la página Web del libro.

- 14.4.** Se considera la clase *Deportes* como la clase base de la que se derivan otras clases, tales como *Fútbol*, *Baloncesto*, *Voleibol*. Cada tipo de deporte tiene la posibilidad de hacer que una persona juegue al deporte correspondiente. En este caso la clase *Deportes* declara la función virtual *jugar*, y cada tipo de deporte se declara como una clase derivada de la clase base *Deportes* y define la función virtual *jugar*. En la declaración *Deportes *Dep[4]* se declara un array de punteros a *Deportes*. Posteriormente, se inicializa *Dep* a los distintos deportes. En la llamada *Dep[i]->jugar("Lucas")* se produce una ligadura dinámica, ya que el compilador no puede determinar cuál es la implementación específica de la función *jugar()* que se ha de llamar hasta que no se ejecuta el programa.

La codificación de este ejercicio se encuentra en la página Web del libro.

SOLUCIÓN DE LOS PROBLEMAS

- 14.1.** Se considera la jerarquía *Obj_geometrico*, *Cuadrado* y *Círculo*.



Un círculo se caracteriza por su centro y su radio. Un cuadrado se puede representar también por su centro y uno de sus cuatro vértices. Se declaran las dos figuras geométricas como clases derivadas. La clase *obj_geom* contiene solamente el centro. Las clases *círculo* y *cuadrado* añaden respectivamente los atributos del radio, y las coordenadas de un punto

cualquiera. De esta forma se pueden obtener el área del círculo y del cuadrado, así como el perímetro del cuadrado y la longitud de la circunferencia, mediante las siguientes expresiones:

Círculo

$$\text{Área} = \pi * r^2$$

$$\text{Circunferencia} = 2 * \pi * r$$

Cuadrado

$$\text{Área} = \text{diámetro} * \text{diámetro}/2$$

$$\text{perímetro} = 4 * \text{lado}$$

$$\text{diámetro} = 2 * \sqrt{(a - x)^2 + (b - y)^2}$$

$$\text{lado} = \sqrt{2 * \left(\frac{\text{diámetro}}{2}\right)^2}$$

La codificación de este problema se encuentra en la página Web del libro.

El caso del programa anterior se representa en el siguiente gráfico.

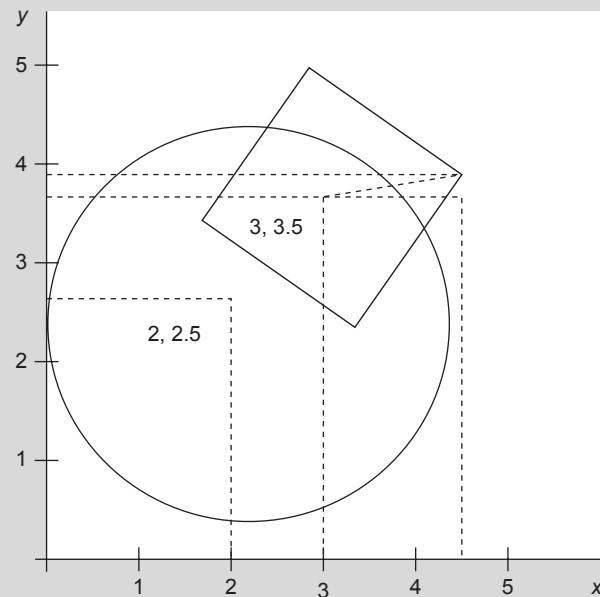
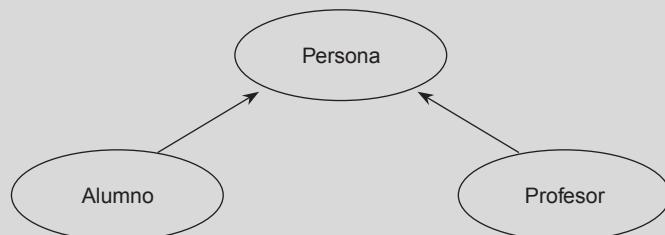


Figura 14.3. Círculo (centro: 2, 2.5), cuadrado (centro: 3, 3.5)

- 14.2. La jerarquía de clases viene definida por el gráfico adjunto. Se declaran las clases de acuerdo con lo indicado, así como los constructores correspondientes. La función miembro *Visualizar* de la Clase *Persona* se declara virtual, y vuelve a definirse en las clases *Alumno* y *Profesor* añadiendo simplemente las sentencias correspondientes que permite mostrar los datos. El constructor *Persona* inicializa los atributos de una persona, independientemente de que sea un alumno o un profesor. El constructor *Alumno* llama al constructor de *Persona* e inicializa el resto de los atributos de un alumno. Este mismo trabajo es realizado por el constructor *Profesor* para la clase *Profesor*.



La codificación de este problema se encuentra en la página Web del libro.

- 14.3.** Se declara en el programa principal un vector de punteros a la clase persona. Cada vez que se lee los datos de un alumno o de un profesor se crea el objeto correspondiente mediante una orden de búsqueda de memoria libre y se almacena en el vector. El programa que se presenta tiene un menú de opciones que permite elegir entre alumno, profesor, visualizar los datos del vector o terminar. En este último caso, se libera la memoria mediante las correspondientes órdenes de liberación. Se codifica además una función `leerdatosP` que lee los datos de una persona.

La codificación de este problema se encuentra en la página Web del libro.

PROBLEMAS PROPUESTOS

- 14.1.** Definir una clase base `persona` que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.) Diseñar una jerarquía de clases que contemple las clases siguientes: estudiante, empleado, estudiante_empleado. Escribir un programa que lea de la entrada de datos y cree una lista de personas: *a)* general; *b)* estudiantes; *c)* empleados; *d)* estudiantes empleados. El programa debe permitir ordenar alfabéticamente por el primer apellido.
- 14.2.** Implementar una jerarquía `Librería` que tenga al menos una docena de clases. Considérese una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etc.
- 14.3.** Implementar una jerarquía `Empleado` de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos cuatro niveles, con herencia de miembros dato, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para `Empleados` de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a `Empleados`.
- 14.4.** Implementar una clase `Automovil` (`Carro`) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *vehículo*, un automóvil es también una *comodidad, un símbolo de estado social, un modo de transporte*, etc.
- 14.5.** Implementar una jerarquía de tipos datos numéricos que extienda los tipos de datos fundamentales tales como `int`, y `float`, disponibles en C++. Las clases a diseñar pueden ser `Complejo`, `Fracción`, `Vector`, `Matriz`, etc.
- 14.6.** Implementar una jerarquía `Empleado` de cualquier tipo de empresa que le resulte conocida. La jerarquía debe tener al menos 3 niveles, con herencia de miembros. Escribir un programa que cree objetos de los diferentes tipos de empleados y realice operaciones polimórficas.
- 14.7.** Implementar una jerarquía de herencia de animales tal que contenga al menos seis niveles de derivación y doce clases.

CAPÍTULO 15

Plantillas, excepciones y sobrecarga de operadores

Introducción

Para definir clases y funciones que operan sobre tipos arbitrarios se definen los *tipos parametrizado* o *tipos genéricos*. La mayoría de los lenguajes de programación orientados a objetos proporcionan soporte para la genericidad. C++ proporciona la característica **plantilla** (*template*), que permite a los tipos ser parámetros de clases y funciones. C++ soporta esta propiedad a partir de la versión 2.1 de AT&T. Las *plantillas* o *tipos parametrizado* se pueden utilizar para implementar estructuras y algoritmos que son en su mayoría independientes del tipo de objetos sobre los que operan.

Uno de los problemas más importantes en el desarrollo de software es la gestión de condiciones de error. Las *excepciones* son, normalmente, condiciones de errores imprevistos. Estas condiciones suelen terminar el programa del usuario con un mensaje de error proporcionado por el sistema. *El manejo de excepciones* es el mecanismo previsto por C++ para el tratamiento de excepciones. Normalmente, el sistema aborta la ejecución del programa cuando se produce una excepción y C++ permite al programador intentar la recuperación de estas condiciones y continuar la ejecución del programa.

La sobrecarga de operadores es una de las características de C++ y, naturalmente de la programación orientada a objetos. La *sobrecarga de operadores* hace posible manipular objetos de clases con operadores estándar tales como +, *, [] y <<. Esta propiedad de los operadores permite redefinir el lenguaje C++, que puede crear nuevas definiciones de operadores. En este capítulo, se mostrará la sobrecarga de operadores unitarios y binarios, incluyendo el operador de moldeado (*cast*) para conversión de tipos implícitos y explícitos, operadores relacionales, operador de asignación y otros operadores.

15.1. Genericidad

La **genericidad** es una propiedad que permite definir una clase (o una función) sin especificar el tipo de datos de uno o más de sus miembros (parámetros). De esta forma se puede cambiar la clase para adaptarla a los diferentes usos sin tener que rescribirla. Las clases genéricas son, normalmente, clases contenedoras que guardan objetos de un determinado tipo. Ejemplos de clases contenedoras son pilas, colas, listas, conjuntos, diccionarios, árboles, etc. C++ propone dos tipos de plantillas: las plantillas de funciones y las plantillas de clases. Para definir clases y funciones que operan sobre tipos de datos arbitrarios se definen los tipos parametrizados o tipos genéricos.

Una **plantilla de funciones** especifica un conjunto infinito de funciones sobrecargadas. Cada función de este conjunto es una función plantilla y una instancia de la plantilla de función. La sintaxis de una plantilla de funciones tiene el formato:

```
template <argumentos genéricos> tipo nombreFunción(Argumentos función)
{ ... }
```

La declaración de la plantilla de funciones comienza con `template`, seguida de una lista separada por comas de parámetros formales, encerrados entre corchetes tipo ángulo (`<<` y `>>`); esta lista no puede ser vacía. Cada parámetro formal consta de la palabra reservada `class`, seguida por un identificador. Esta palabra reservada indica al compilador que el parámetro representa un posible tipo incorporado definido por el usuario. Se necesita, al menos, un parámetro `T` que proporcione datos sobre los que pueda operar la función. También se puede especificar un puntero (`T *` parámetro) o una referencia (`T & parámetro`).

Las **plantillas de clase** permiten definir clases genéricas que pueden manipular diferentes tipos de datos. Una aplicación importante es la implementación de *contenedores*, clases que contienen objetos de un tipo dato, tales como vectores (*arrays*), listas, secuencias ordenadas, tablas de dispersión (*hash*); en esencia, los contenedores manejan estructuras de datos.

Así, es posible utilizar una clase plantilla para crear una pila genérica, por ejemplo, que se puede instanciar para diversos tipos de datos predefinidos y definidos por el usuario. Puede tener también clases plantillas para colas, vectores (*arrays*), matrices, listas, árboles, tablas (*hash*), grafos y cualquier otra estructura de datos de propósito general.

La definición de una clase genérica tiene la siguiente sintaxis:

```
template <argumentos genéricos> class NombreClase { ... }
```

Los argumentos genéricos, generalmente, representan tipos de datos; en ese caso se utiliza la palabra reservada `class`. Entonces la sintaxis es:

```
template <class T> class NombreClase { ... }
```

La implementación de una función miembro de clase genérica fuera de la clase, exige indicar que es una función genérica con el prefijo `template <class T>`.

El identificador o nombre de una clase genérica es el nombre de la clase seguido, entre corchetes angulares, de los argumentos genéricos.

Los argumentos genéricos de las plantillas pueden ser, además de los tipos de datos, cualquier otro que se desee realizar como, por ejemplo, para dimensionar un array unidimensional genérico.

```
template < class T, int n>
class ejemplo
{
    T V[n]
    ...
}
```

La plantilla de clase genérica debe ser instanciada adecuadamente en el momento de declarar un objeto de una clase genérica, para lo cual ha de indicarse el tipo entre ángulo. En el momento que se realiza la declaración de una instancia de la clase genérica se pasan los argumentos correspondientes de la declaración a los argumentos genéricos, entre corchetes angulares, que aparecen a continuación del nombre de la clase. Para el caso de la declaración anterior una instancia puede ser:

```
ejemplo <int> instancia(4);
//el tipo de la instancia es entero y almacena un vector de 4 datos.
```

EJEMPLO 15.1. Pila genérica con vector dinámico.

El prefijo `template <class T>` en la declaración de clases indica que se declara una plantilla de clase y que se utilizará `T` como el tipo genérico. Por consiguiente, `PilaG` es una clase parametrizada con el tipo `T` como parámetro.

```
template <class T>
class PilaG
{
    private:
        T* Vector;
        int cima, maximo;
```

```

public:
    PilaG(){ cima = -1;}
    ~PilaG(){}
    PilaG(int n)
    { cima =-1;
        maximo = n-1;
        Vector = new T[n];
    }
    void Anade(T x);
    T Borrar();
    bool Vacia();
};

template <class T>
void PilaG<T>:: Anade( T x) {...} // añadir codificación

template <class T>
void PilaG<T>:: Borrar( ) {...} // añadir codificación

template <class T>
bool PilaG<T>:: Vacia () {...} // añadir codificación

...
PilaG < int > P(40); // declara una pila de 40 enteros
PilaG < float > P(60); // declara una pila de 60 reales.

```

EJEMPLO 15.2. *Array genérico con número de datos variables.*

Se declara una clase que define un vector de n datos variables y cuyo tipo es genérico.

```

#include <cstdlib>
#include <iostream>

using namespace std;
template <class T, int n>
class Array
{
public:
    T V[n];
};

int main(int argc, char *argv[])
{   Array < float, 20> A; // array real de 20 elementos
    Array <char, 30> B; // array char de 30 elementos
    Array <double, 40> C; // array double de 40 elementos

    A.V[2]=6;
    cout << A.V[2];
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

15.2. Excepciones

Las *excepciones* son anomalías (condiciones de error no esperadas) producidas durante la ejecución de un programa. Si ocurre una excepción y está activa un segmento de código denominado *manejador de excepción* para esa excepción, entonces el flujo de control se transfiere al manejador. Si en lugar de ello, ocurre una excepción y no existe un manejador para la excepción, el programa se termina.

El modelo de un mecanismo de excepciones de C++ se basa en el concepto de salto no local y se implementa con las palabras reservadas `try`, `throw` y `catch`.

- `try`, es un bloque para detectar excepciones.
- `catch`, es un manejador para capturar excepciones de los bloques `try`.
- `throw`, es una expresión para levantar (*raise*) o lanzar excepciones.

Los pasos del modelo son:

1. Primero programar “intentar” (`try`) una operación para anticipar errores.
2. Cuando se encuentra un error, se “lanza” (`throws`) una excepción. El lanzamiento (*throwing*) de una excepción es el acto de levantar una excepción.
3. Por último, alguien interesado en una condición de error (para limpieza y/o recuperación) se anticipa al error y “capturará” (`catch`) la excepción que se ha lanzado.

EJEMPLO 15.3. Esquema de tratamiento de una excepción.

El esquema tiene tres partes: código que produce la excepción que se lanza; llamada al código que puede tener error; captura de la excepción.

```
void f()
{
    // código que produce una excepción que se lanza
    throw (i);
    // ...
}
int main()
{
    try {
        f();           // Llamada a f - preparada para cualquier error
        // Código normal aquí
    }
    catch(...)
    {
        // capturar cualquier excepción lanzada por f()
        // hacer algo
    }
    // Resto código normal de main()
}
```

15.2.1. LANZAMIENTO DE EXCEPCIONES

Las excepciones se lanzan utilizando la sentencia `throw`, cuya sintaxis es:

`throw (expresión);` o bien `throw();`

El primer formato especifica que la expresión se pasa necesariamente al **manejador de excepciones** (`catch`), que es quien la recoge y la trata. El segundo formato no devuelve ninguna expresión e indica que se ha producido una excepción que es cap-

turada por la función `terminar()`. `throw` se comporta como `return`. Lo que sucede es lo siguiente: el valor *devuelto* por `throw` se asigna al objeto del `catch` adecuado.

La sentencia `throw` levanta una excepción. Cuando se encuentra una excepción en un programa C++, la parte del programa que detecta la excepción puede comunicar que la misma ha ocurrido por le vantamiento, o *lanzamiento* de una excepción.

Uno de los usos más corrientes de `throw` es el de las funciones. En efecto, esta cláusula puede especificar qué tipos de excepciones pueden ocurrir en la declaración de una función. Una *especificación de excepciones* proporciona una solución que se puede utilizar para listar las excepciones que una función puede lanzar con la declaración de función. Una especificación de excepciones se añade a una declaración o a una definición de una función. Los formatos son:

```
tipo nombre_función(signatura) throw (e1, e2, eN);           // prototipo
tipo nombre_función(signatura) throw (e1, e2, eN);           // definición
{
    cuerpo de la función
}
```

donde `e1, e2, eN` es una lista separada por comas de nombres de excepciones (la lista especifica que la función puede lanzar directa o indirectamente, incluyendo excepciones derivadas públicamente de estos nombres).

15.2.2. MANEJADORES DE EXCEPCIONES

Las excepciones levantadas dentro de un bloque son capturadas utilizando cláusulas `catch` asociadas con el bloque. Una cláusula `catch` consta de tres pares: la palabra reservada `catch`, la declaración de un solo tipo o un simple objeto dentro de paréntesis (referenciado como *declaración de excepciones*), y un conjunto de sentencias. Si la cláusula `catch` se selecciona para manejar una excepción, se ejecuta la sentencia compuesta.

La especificación del manejador `catch` se asemeja a una definición de función sin tipo de retorno.

```
catch (TipoParámetro NombreParámetro)
{
    CuerpoManejador
}
```

Al igual que con funciones sobrecargadas, el manejador de excepciones se activa sólo si el argumento que se pasa (*o se lanza “thrown”*) se corresponde con la declaración del argumento de la sentencia `catch`.

15.2.3. DISEÑO DE EXCEPCIONES

La palabra reservada `try` designa un bloque `try`, que es un área de su programa que detecta excepciones. En el interior de bloques `try`, normalmente se llaman a funciones que pueden levantar o *lanzar* excepciones. La palabra reservada `catch` designa un manejador de capturas con una firma que representa un tipo de excepción. Los manejadores de captura siguen inmediatamente a bloques `try` o a otro manejador `catch` con una firma diferente.

Un manipulador consiste en un bloque `try`, donde se incluye el código que puede producir la excepción. Un bloque `try` es el contexto para decir cuáles son los manipuladores que se invocan cuando se levanta una excepción. El orden en que se definen los manejadores determina el orden en que se prueban los mismos para una excepción levantada coincidente en tipos.

Un bloque `try` tiene la siguiente sintaxis:

```
try
{
    código del bloque try
}
catch (firma)
{
    código del bloque catch
}
```

EJEMPLO 15.4. Tratamiento de las excepciones para la resolución de una ecuación de segundo grado.

Una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$ tiene las siguientes raíces reales.

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Los casos en los cuales las expresiones anteriores no tienen sentido son: 1) $a = 0$; 2) $b^2 - 4ac < 0$, que no produce raíces reales, sino imaginarias. En consecuencia, se consideran excepciones de tipo error tales como:

```
enum error {no_raices_reales, coeficiente_a_cero};
```

Una codificación completa es:

```
#include <cstdlib>
#include <iostream>
#include <math.h>

using namespace std;
enum error {no_raices_reales, Coeficiente_a_cero};

void raices(float a, float b, float c, float &r1, float &r2)
throw(error)
{
    float discr;
    if(b*b < 4 * a * c)
        throw no_raices_reales;
    if(a==0)
        throw Coeficiente_a_cero;
    discr = sqrt(b * b - 4 * a * c);
    r1 = (-b - discr) / (2 * a);
    r2 = (-b + discr) / (2 * a);
}

int main(int argc, char *argv[])
{
    float a, b, c, r1, r2;
    cout << " introduzca coeficientes de la ecuación de 2º grado :";
    cin >> a >> b >> c;
    try {
        raices (a, b, c, r1, r2);
        cout << " raíces reales " << r1 << " " << r2 << endl;
    }
    catch (error e)
    {
        switch(e) {
            case no_raices_reales :
                cout << "Ninguna raíz real" << endl;
                break;
            case Coeficiente_a_cero :
                cout << "Primer coeficiente cero" << endl;
        }
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

15.3. Sobrecarga

En C++ hay un número determinado de operadores predefinidos que se pueden aplicar a tipos estándar incorporados o integrados. Estos operadores predefinidos, realmente, están sobrecargados ya que cada uno de ellos se puede utilizar para diferentes tipos de datos. Casi todos los operadores predefinidos para tipos estándar se pueden sobrecargar. La Tabla 15.1 muestra dichos operadores

Tabla 15.1. Operadores que se pueden sobrecargar

New	delete	New[]	delete[]						
+	-	*	/	%	^	&			~
!	=	<	>	+=	-=	*=	/=		%=
^=	&=	=	<<	>>	>>=	<<=	==		!=
<=	>=	&&		++	--	,	->*		->
()	[]								

Un *operador unitario* es un operador que tiene un único operando. Un *operador binario*, por el contrario, tiene dos operandos. Algunos operadores tales como + son unitarios y binarios, dependiendo de su uso.

Los siguientes operadores no se pueden sobrecargar:

.	acceso a miembro	.x	indirección de acceso a miembro
::	resolución de ámbitos	?:	if aritmético

Una función operador es una función cuyo nombre consta de la palabra reservada `operator` seguida por un operador unitario o binario.

Sintaxis función: tipo operator operador(lista de parámetros)

Excepto para el caso de los operadores `new`, `delete` y `->`, las funciones operador pueden devolver cualquier tipo de dato. La precedencia, agrupamiento y número de operandos no se puede cambiar. Con excepción del operador `()`, las funciones operador no pueden tener argumentos por defecto. Una función operador debe ser o bien una función miembro no estática o una función o miembro que tenga al menos un parámetro cuyo tipo es una clase, referencia o una clase enumeración, o referencia a una enumeración. Cuando se declaran nuevos y propios operadores para clases lo que se hace es escribir una función con el nombre `operatorX` en donde `x` es el símbolo de un operador. Existen dos medios de construir nuestros propios operadores, bien formulándolos como funciones amigas, o bien como funciones miembro.

EJEMPLO 15.5. Operadores unitarios binarios y funciones operador.

```

n++;           +x;           // unitarios
x / y;         x + y;       // binarios

int operator *(int x, int y)
// código no válido para definir una función operador + ya que no se
// define sobre una clase.
    return x * y;
}

class Integer {
// ...
public:
    int valor;
};

```

```

int operator * (const Integer &x, int y)
{ //función no miembro válida ya que al menos un parámetro es clase:
    return x.valor * y;
}

```

15.3.1. SOBRECARGA DE OPERADORES UNITARIOS

Los operadores unitarios son aquellos que actúan sobre un único operando. Estos operadores son los siguientes:

new	delete	new[]	delete[]
++	incremento	--	decremento
()	llamada a función	[]	subíndice
+	más	-	menos
*	desreferencia (indirección)	&	dirección de
!	NOT lógico	~	NOT bit a bit
,	coma		

Cuando se sobrecargan operadores unitarios en una clase el operando es el propio objeto de la clase donde se define el operador. Por tanto, los operadores unitarios dentro de las clases no requieren operandos explícitos. La sintaxis de declaración es la siguiente:

```
<tipo> operador <unitario>();
```

Si el operador unitario, tiene forma postfija y prefija, como por ejemplo los operadores ++ o — entonces se usa la siguiente declaración para cada una de las formas:

prefija	postfija
<tipo> operador <unitario>();	<tipo> operador <unitario>(int);

Los operadores unitarios pueden sobrecargarse como *funciones miembro* o como *funciones amigas*. En el primer caso el sistema pasa el puntero `this` implícitamente. Por consiguiente, el objeto que llama a la función miembro se convierte en el operando de este operador. En el segundo caso el sistema no pasa el puntero `this` implícitamente. Por consiguiente, se debe pasar explícitamente el objeto de la clase.

EJEMPLO 15.6. Sobre carga de operadores unitario como función miembro y como función amiga.

La función operador unitario `--()` está declarada, y ya que es una función miembro, el sistema pasa el puntero `this` implícitamente. Por consiguiente, el objeto que llama a la función miembro se convierte en el operando de esta operador. La función operador `++()` está definida, como función amiga, por lo que el sistema no pasa el puntero `this` implícitamente. Por consiguiente, se debe pasar explícitamente el objeto de la clase `Unitaria`.

```

#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

class Unitaria {
    int x;
public:
    Unitaria()           {x = 0;}          // constructor por defecto
    Unitaria(int a)      {x = a;}         // constructor con un parámetro
    friend Unitaria& operator++(Unitaria y) {++y.x; return y; }
    Unitaria& operator-- () {--x ; return *this;}
    void visualizar(void) {cout << x << "\n";}
};

```

```

int main(int argc, char *argv[])
{
    Unitaria ejemplo = 65;
    for(int i = 5; i > 0; i--)
    {
        ++ejemplo;                                // operator++(ejemplo);
    }
    for(int i = 5; i > 0; i--)
    {
        --ejemplo;                                // ejemplo.operator--();
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

15.3.2. CONVERSIÓN DE DATOS Y OPERADORES DE CONVERSIÓN FORZADA DE TIPOS

Cuando en C++ una sentencia de asignación asigna un valor de un tipo estándar a una variable de otro tipo estándar, C++ convierte automáticamente el valor al mismo tipo que la variable receptora, haciendo que los dos tipos sean compatibles. El compilador funcionará de modo diferente según que la conversión sea entre tipos básicos, tipos básicos y objetos, o bien entre objetos de diferentes clases si la conversión automática falla, se puede utilizar una conversión forzada de tipos (*cast*):

```

int *p =(int *) 50;                      // p y (int *) 50 son punteros

varentera = int(varfloat);

```

El moldeado (*casting*) es una *conversión explícita* y utiliza las mismas rutinas incorporadas a la con versión implícita.

En la conversión entre tipos definidos por el usuario y tipos básicos (incorporados) no se pueden utilizar las rutinas de conversión implícitas. En su lugar se deben escribir estas rutinas. Así, para convertir un tipo estándar o básico —por ejemplo, `float`— a un tipo definido por el usuario, por ejemplo, `Distancia`, se utilizará un *constructor con un argumento*. Por el contrario, para convertir un tipo definido por el usuario a un tipo básico se ha de emplear un *operador de conversión*.

Las *funciones de conversión* son funciones operador. El argumento en estas funciones es el argumento implícito `this`. Para convertir a un tipo *nombretipo* se utiliza una función de conversión con este formato:

```
operator nombretipo();
```

y debe tener presente los siguientes puntos:

- La función de conversión debe ser un método de la clase a convertir.
- La función de conversión no debe especificar un tipo de retorno.
- La función de conversión no debe tener argumentos.
- El nombre de la función debe ser el del tipo al que se quiere convertir el objeto.

EJEMPLO 15.7. Conversión de coordenadas cartesianas a polares.

Se presenta una función de conversión diseñada para convertir clases diferentes: `Rect`, que representa la posición de un punto en el sistema de coordenadas rectangulares y `Polar` que representa la posición del mismo punto en el sistema de coordenadas polares. Las coordenadas de un punto en el plano se pueden representar por las coordenadas rectangulares `x` e `y`, o bien por las coordenadas polares `Radio` y `Angulo`. Las clases `Rec` y `Polar`, son de estructura diferente y la sentencia de asignación

```
rect = polar;
```

implica una conversión que se realiza mediante la función de conversión. Las fórmulas a utilizar son:

```
x = radio*coseno(angulo)
y = radio*seno(angulo)
```

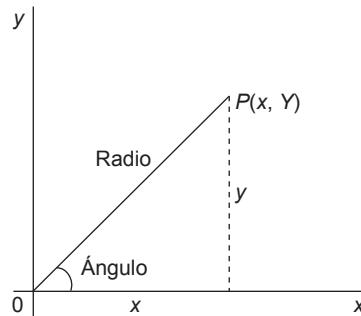


Figura 15.1. Coordenadas rectangulares de un punto.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

class Rect
{
private:
    double xco;           // coordenada x
    double yco;           // coordenada y
public:
    Rect()
        {xco = 0.0; yco = 0.0;} // constructor sin argumentos
    Rect(double x, double y)// constructor con dos argumentos
        {xco = x; yco = y;}
    void visualizar()
        {cout << " ( " << xco << " , " << yco << " ) ";}
};

class Polar
{
private:
    double radio, angulo;
public:
    Polar()           // constructor sin argumentos
        {radio = 0.0; angulo = 0.0;}
    Polar(double r, double a) // constructor dos argumentos
        {radio = r; angulo = a ;}
    void visualizar()
    {
        cout << " ( " << radio << " , " << angulo << " ) ";
    }

    operator Rect()           // función de conversión
    {
        double x = radio * cos(angulo);           // calcular x
        double y = radio * seno(angulo);           // calcular y
        return Rect(x, y);
    }
}
```

```

        double y = radio * sin(angulo);           // calcular y
        return Rect(x, y);
    } // invoca constructor de la clase Rect
}

int main(int argc, char *argv[])
{
    Rect rec;                                // Rect utiliza constructor
    Polar pol(100.0,0.585398);               // Polar que utiliza constructor 2
    //rec = pol;                            // convierte Polar a Rect
    // utilizando función de conversión
    rec = Rect(pol);
    cout << "\n polar="; pol.visualizar();      // polar original
    cout << "\n rectang="; rec.visualizar();    // rectangular equivalente
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Resultado de ejecución:

```

polar= < 100 , 0.585398 >
rectang= < 83.3492 , 55.2531 >

```

15.3.3. SOBRECARGA DE OPERADORES BINARIOS

Los operadores binarios toman dos argumentos, uno a cada lado del operador. La Tabla 15.2 lista los operadores binarios que se pueden sobrecargar.

Tabla 15.2. Operadores binarios que se pueden sobrecargar

Operador	Significado	Operador	Significado
+	Adición (suma)	=	Or bit a bit con asignación
-	Resta	==	Igual
*	Multiplicación	!=	No igual
/	División	>	Mayor que
%	Módulo	<	Menor que
=	Asignación	>=	Mayor o igual que
+=	Suma con asignación	<=	Menor o igual que
-=	Resta con asignación		OR lógico
*=	Multiplicación con asignación	&&	AND lógico
/=	División con asignación	<<	Desplazamiento a izquierda
%=	Módulo con asignación	<<=	Desplazamiento a izquierda con asignación
&	AND bit a bit	>>	Desplazamiento a derecha
	OR bit a bit	>>=	Desplazamiento a derecha con asignación
^	OR exclusivo bit a bit	->	Puntero
^=	OR exclusivo bit a bit con asignación	->*	Puntero a miembro
%=	AND bit a bit con asignación		

Los operadores binarios pueden ser sobrecargados pasando a la función dos argumentos. El primer argumento es el operando izquierdo del operador sobrecargado y el segundo argumento es el operando derecho, si la función no es miembro de la clase. Consideremos un tipo de clase *t* y dos objetos *x1* y *x2* de tipo *t*. Definamos un operador binario + sobrecargado. Entonces:

x1 + *x2* se interpreta como operator+(*x1*, *x2*) o como *x1.operator+(x2)*

Un operador binario sobrecargado puede ser definido:

- bien como una *función amiga* de dos argumentos,
- bien como una *función miembro* de un argumento, y es el caso más frecuente.

EJEMPLO 15.8. *Sobrecarga de operadores binarios como función amiga y como función miembro.*

La *función miembro* + sobrecarga el operador suma por lo que sólo se le pasa un operando como argumento. El primer operando es el propio objeto, y el segundo es el que se pasa como parámetro. La función operador binario - () está declarado como *función amiga* por lo cual el sistema no pasa el puntero `this` implícitamente y, por consiguiente, se debe pasar el objeto Binario explícitamente con ambos argumentos. Como consecuencia, el primer argumento de la función miembro se convierte en el operando izquierdo de este operador y el segundo argumento se pasa como operando derecho.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
class Binario
{
    float x;
public:
    Binario()          {x = 0;}
    Binario(float a)   {x = a;}
    Binario operator +(Binario&);           // función miembro
    friend Binario operator -(Binario&,Binario&); // función amiga
    void visualizar(void) {cout << x << "\n";}
};

Binario Binario::operator+(Binario& a)
{
    Binario aux;
    aux.x = x + a.x;
    return aux;
}
Binario operator -(Binario& a, Binario& b)
{
    Binario aux;
    aux.x = a.x - b.x;
    return aux;
}

int main(int argc, char *argv[])
{
    Binario primero(2.8), segundo(4.5), tercero;
    tercero = primero + segundo;
    cout << " suma :";
    tercero.visualizar();
    tercero = primero - segundo;
    cout << " resta :";
    tercero.visualizar();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Sobrecarga del operador de asignación

Por defecto C++ sobrecarga el operador de asignación para objetos de clases definidas por el usuario, copiando todos los miembros de la clase. Cuando se asigna un objeto a otro objeto, se hace una copia miembro a miembro de forma opuesta a como se hace la copia bit a bit (*bitwise*). La asignación de un objeto de una clase a otro objeto de esa clase se realiza utilizando el operador de asignación de copia. Los mecanismos son esencialmente los mismos que los citados de inicialización por defecto miembro a miembro, pero hace uso de un operador de asignación de copia implícito en lugar del constructor de copia. El formato general del operador de asignación de copia es el siguiente:

```
// formato general del operador de asignación de copia
nombreClase& nombreClase::operator=(const nombreClase&v)
{
    // evita autoasignaciones
    if (this != &v)
    {
        // semántica de la copia de clases
    }
    // devolver el objeto asignado
    return *this;
}
```

Reglas¹

Cuando un objeto de una clase se asigna a otro objeto de su clase, tal como `nuevoObj = antiguoObj`, se siguen los siguientes pasos:

1. Se examina la clase para determinar si se proporciona un operador de asignación de copia explícita.
2. Si es así, su nivel de acceso se comprueba para determinar si se puede invocar o no, dentro de una porción del programa.
3. Si no es accesible, se genera un mensaje de error en tiempo de compilación; en caso contrario, se invoca para ejecutar la asignación.
4. Si no se proporciona una instancia explícita, se ejecuta la asignación miembro a miembro.
5. Bajo la asignación miembro a miembro por defecto, cada miembro dato de un tipo compuesto o incorporado se asigna al valor de su miembro correspondiente.

A cada miembro del objeto de la clase se aplica recursivamente los pasos 1 a 6 hasta que se asignan todos los miembros dato de los tipos compuestos o incorporados.

¹ Lipman, S. y Lajoie, J.: *C++ Primer*, 3.^a ed., Reading, Massachusetts: Addison-Wesley, 1998, p. 730.

EJERCICIOS

- 15.1.** Declarar una clase genérica para un árbol binario, que permita trabajar con una implementación de punteros.
- 15.2.** Explique el funcionamiento de la función `f` definida para el tratamiento de excepciones de la clase `cadena` en el siguiente fragmento de programa.

```

class cadena
{
    char* s;
public:
    enum {minLong = 1, maxLong = 1000};
    cadena();
    cadena(int);
};

cadena::cadena(int longitud)
{
    if (longitud < minLong || longitud >= maxLong)
        throw (longitud);
    s = new char [longitud+1];
    if (s == 0)
        throw ("Fuera de memoria");
}

void f(int n)
{
    try{
        cadena cad (n);
    }

    catch (char* Msgerr)
    {
        cout << Msgerr << endl;
        abort();
    }
    catch(int k)
    {
        cout << "Error de fuera de rango :" << k << endl;
    }
}

```

PROBLEMAS

- 15.1.** Diseñar una pila de tipo genérico que permita inserción y borrado de elementos.
- 15.2.** Escribir un programa que permita generar aleatoriamente un vector, lo ordene, y posteriormente visualice el resultado del vector ordenado. Deben realizarse todas las funciones mediante el uso de plantillas.

- 15.3.** Una pila es un tipo estructurado de datos que recibe datos por un extremo y los retira por el mismo, siguiendo la regla “último elemento en entrar, primero en salir”. Diseñar una pila con tratamiento de excepciones que permita crear una pila de enteros de longitud variable y detectar si se produce un desbordamiento porque se quieren añadir más datos de los que se han reservado, o bien se quiere retirar un dato y no lo tiene.
- 15.4.** Definir una clase genérica para arrays, con estos argumentos genéricos: `<class T, int n>`. El segundo argumento será la capacidad del array. Las operaciones a implementar: `asignarElemento`, `devolverElement(index)` y visualizar todos los elementos asignados.
- 15.5.** Sobrecargar el operador unitario `++` en forma prefija y postfija para la clase `tiempo` que contiene los atributos `horas`, `minutos` y `segundos`, encargada de incrementar el tiempo cada vez que se llame en un segundo, realizando los cambios necesarios en los atributos `minutos` y `horas` si fuera necesario. Programar, además, las funciones miembro encargadas de leer una hora y de visualizarla.
- 15.6.** Declarar la clase `vector` que contiene cuatro coordenadas `x`, `y`, `z`, `t`, y sobrecargar operadores binarios que permitan realizar la suma y diferencia de vectores, así como el producto escalar. Sobrecargar operadores unitarios que permitan cambiar de signo a un vector incrementar o decrementar en una unidad todas las coordenadas de un vector, así como calcular la norma de un vector.
- 15.7.** Añadir a la clase `vector` del Problema 15.6 dos funciones amigas que sobrecarguen los operadores `<< y >>` para permitir sobrecargar los flujos de entrada y salida `leer` y `escribir` vectores con `cin` y `cout` respectivamente.
- 15.8.** Definir una clase `Complejo` para describir números complejos. Un número complejo es un número cuyo formato es: $a + b * i$, donde a y b son números de tipo `double` e i es un número que representa la cantidad $\sqrt{-1}$. Las variables a y b se denominan reales e imaginarias. Sobrecargar los siguientes operadores de modo que se apliquen completamente al tipo `Complejo`: `=, +, -, *, /, >> y <<`. Para añadir o restar dos números complejos, se suman o restan las dos variables miembro de tipo `double`. El producto de dos números complejos viene dado por la fórmula siguiente:

$$(a + b * i) * (c + d * i) = (a * c - b * d) + (a * d + b * c) * i$$

el cociente viene dado por la fórmula:

$$(a + b * i) / (c + d * i) = ((a * c + b * d) + (-a * d + b * c)) / (c * c + d * d)$$

SOLUCIÓN DE LOS EJERCICIOS

- 15.1.** Un árbol binario no vacío, tiene almacenado en su zona protegida un elemento `e`, de tipo genérico `T`, y además un hijo izquierdo `hi` y un hijo derecho `hd` que son punteros a la clase `Arbol`. Para tratar con árbol hay que hacerlo realmente con un puntero a la clase `Arbol`, tal y como puede observarse en el programa principal. Se declara en primer lugar, la clase genérica `Arbol` precedida de su correspondiente `template`. Esta declaración de plantilla es válida para todas las funciones miembro que se definen dentro de la propia declaración de la clase. Todas las funciones miembro que se definen fuera de la clase, debido por ejemplo a su gran longitud de código, deben ser precedidas de la correspondiente orden `template`, tal y como aparece en el ejercicio. La codificación de funciones miembro muy similares a las aquí declaradas pueden consultarse en el capítulo de árboles.

La codificación de este ejercicio se encuentra en la página Web del libro.

- 15.2.** La clase cadena contiene un atributo que es un puntero a `char`; por ello para poder almacenar cadenas de caracteres se necesita reservar memoria. Esta operación es realizada con el constructor `cadena(int)`. Para poder determinar la longitud máxima y mínima de una cadena de caracteres se utiliza un tipo enumerado con valores 1 y 1000. Mediante estos valores se determina el rango dentro del cual puede reservarse memoria, de tal manera que si no se cumple la condición de

estar en el rango, entonces lanza una excepción retornando la longitud de cadena que no es viable debido a la declaración. Si por alguna circunstancia, puede reservarse memoria, pero al hacerlo el sistema no dispone de ella, entonces se lanza otra excepción con el mensaje de que el tamaño fuera de memoria. La función `f` realiza la reserva de memoria, y si se produce un error es capturado con el `catch` e informado al usuario. La misma codificación con comentarios adecuados en los puntos más importantes es la siguiente:

La codificación de este ejercicio se encuentra en la página Web del libro.

- 15.3.** En el capítulo de Pilas y Colas se explica con detalles la implantación de una cola en un array de tamaño fijo. En este ejercicio sólo se incluye el prototipo de las funciones miembro de la clase `Cola`, así como el tipo enumerado `errores` que sirve para capturar las excepciones y poder tomar decisiones adecuadas.

La codificación de este ejercicio se encuentra en la página Web del libro.

SOLUCIÓN DE LOS PROBLEMAS

- 15.1.** Se implementa la clase `Pila` con array estático, y genericidad de tipos usando las plantillas. Sólo se incluyen las funciones miembro `Pop`, `Push`, el constructor y el destructor. El resto de las funciones miembro son omitidas, y, por ello, los métodos `Push` y `Pop` deben consultar el atributo protegido `cima` para comprobar que no se excede el tamaño máximo de la pila, y que hay datos en la pila respectivamente.

La codificación de este problema se encuentra en la página Web del libro.

- 15.2.** La solución se ha realizado usando las siguientes funciones que tratan vectores de un tipo genérico `T`.

- Aleatorio. Realiza la generación aleatoria del vector de `n` datos.
- Intercambio. Realiza el intercambio de los valores de dos variables de tipo genérico.
- Mayor. Recibe como parámetros dos valores genéricos y decide si el primero es mayor que el segundo.
- Burbuja. Realiza la ordenación de un vector de `n` datos genéricos por el conocido método de ordenación que lleva su nombre.
- Mostrar. Visualiza la información de un vector genérico de `n` datos.
- El programa principal realiza las llamadas correspondientes.

La codificación de este problema se encuentra en la página Web del libro.

- 15.3.** Se declara la pila como una clase que tiene como zona privada un atributo `lapila` que es puntero a enteros (será un array dinámico), y otros dos enteros `cima` y `tamapila` que indicarán la posición donde está el último elemento que se añadió a la pila y cuantos números enteros puede almacenar como máximo la pila. La pila tiene dos constructores, uno por defecto y otro que genera una pila del tamaño que se pase como parámetro, mediante la correspondiente orden de reserva de espacio de memoria para el array dinámico. La pila contiene en la zona pública dos clases anidadas para el tratamiento de las excepciones que son: `Desbordamiento` y `Subdesbordamiento`. En esta zona pública contiene, además, las funciones miembro `meter` un dato en la pila y `sacar` un dato de la pila, encargadas de realizar las correspondientes llamadas a los objetos de desbordamiento o subdesbordamiento de la pila. El programa principal fuerza a la pila para que se desborde, haciendo que se invoque al manejador de excepciones.

La codificación de este problema se encuentra en la página Web del libro.

- 15.4.** La clase `array` tiene una zona privada con un atributo `ndatos` que indica el número de elementos que puede almacenar el array y un puntero a un tipo genérico `T` denominado `Parray`. El constructor de la clase reserva espacio de memoria del tipo `T` para una cantidad de elementos indicada en su parámetro `n`. El destructor, se encarga de liberar la memoria del array. La función miembro `visualizar` se encarga de visualizar el vector completo de datos. Además de las funciones miembro `AsignarElemento` y `DevolverElemento` encargadas de asignar en una posición `i` del array un ele-

mento o de recuperar el elemento que se encuentra almacenado en la posición i , se sobrecarga² el operador `[]` para poder trabajar con el array como si fuera un vector definido de la forma estándar en C++. Se presenta, además, un programa principal en el que se declara un objeto array de 5 elementos denominado `tabla` y se rellena y visualiza la tabla con el operador sobrecargado `[]`.

La codificación de este problema se encuentra en la página Web del libro.

- 15.5.** La clase `tiempo` tiene como atributos privados las horas los minutos y los segundos. Se trata de hacer que el tiempo se incremente en una unidad mediante el operador sobrecargado `++` tanto en forma prefija como forma postfija. Para sobre cargar el operador `++`, en forma prefija se declara la función miembro de la siguiente forma: `void operator ++ (void)`. Para hacerlo en forma postfija se declara la función miembro con `int` entre paréntesis `void operator ++ (int)`. El constructor `tiempo` inicializa el tiempo a cero. La función miembro `verHora` visualiza la hora de la clase `tiempo`. La función miembro `leerTiempo`, solicita la hora de la entrada y la convierte en datos correctos. Los minutos y segundos deben estar en el rango 0 a 59. Una vez declarado el objeto `tiempo t1`, se pueden realizar las llamadas en forma prefija siguientes: `++t1; o t1.operator++();`. Las llamadas en forma postfija son `t1++; o t1.operator++(0);`.

La codificación de este problema se encuentra en la página Web del libro.

- 15.6.** La clase `vector` tiene como atributos privados las cuatro coordenadas x, y, z, t del vector de dimensión 4. Se programan las siguientes funciones miembro:

- Constructor `vector`, encargado de inicializar el vector a las coordenadas 0,0,0,0 o bien con los parámetros que reciba.
- Operador binario `+` sobrecargado para realizar la suma de dos vectores componente a componente:

$$a - b = a.x - b.x, \quad a.y - b.y, \quad a.z - b.z, \quad a.t - b.t$$

- Operador unitario – que cambia de signo todas las componentes de un vector:

$$-a = (-a.x, -a.y, -a.z, -a.t)$$

- Operador binario – cuya sobrecarga permite realizar la diferencia de dos vectores, entendida como diferencia de las correspondientes componentes:

$$a - b = (a.x - b.x, \quad a.y - b.y, \quad a.z - b.z, \quad a.t - b.t)$$

- Operadores unitarios `++` y `--` que realizan el incremento o decremento de una unidad en todas las componentes del vector:

$$a-- = (a.x--, \quad a.y--, \quad a.z--, \quad a.t--)$$

$$a--- = (a.x--, \quad a.y--, \quad a.z--, \quad a.t--)$$

- Operador binario `*` encargado de realizar el producto escalar definido como la suma de los productos de las componentes de dos vectores:

$$a * b = a.x * b.x - a.y * b.y - a.z * b.z - a.t * b.t$$

- Operador unitario `*` sobrecargado para calcular la norma de un vector:

$$\text{norma}(a) = \sqrt{a.x * a.x + a.y * a.y + a.z * a.z + a.t * a.t}$$

La codificación de este problema se encuentra en la página Web del libro.

- 15.7.** La sobrecarga del flujo de salida `<<` añade a la clase `vector` la posibilidad de visualizar el vector usando la sentencia del flujo de salida. Análogamente, ocurre con la sobrecarga del operador `>>` para el flujo de entrada, añadiendo a la clase punto un operador de flujo de salida `<<` sobrecargado, se podrán escribir objetos punto en sentencias de flujo de salida. La función amiga `operator<<()` devuelve `ostream&` y declara dos parámetros os de tipo `ostream` y v, una referencia a un objeto `vector`. Análogamente ocurre con la función amiga operador `<<()`.

La codificación de este problema se encuentra en la página Web del libro.

² Véase el capítulo de sobrecarga de operadores. Apartado de operadores unitarios.

- 15.8.** Un número complejo tiene una parte real *a* y una parte imaginaria *b*. La forma normal de representar en matemáticas un número complejo es la siguiente: *a + b * i* (forma binómica). Donde el símbolo *i* se denomina “unidad imaginaria” y simboliza la raíz cuadrada de -1 ($i = \sqrt{-1}$). Debido a su naturaleza compuesta, un número complejo se representa de forma natural por una clase que contenga como atributos la parte real *a* y la parte imaginaria *b*. La sobrecarga de operadores pedida se divide en dos bloques los operadores binarios *+*, *-*, ** /* que se codifican como funciones miembro de la clase *complejo* y los operadores unitarios *<<* y *>>* que se codifican como funciones amigas de la clase *complejo*.

La codificación de este problema se encuentra en la página Web del libro.

EJERCICIOS PROPUESTOS

- 15.1.** Declarar una clase genérica para trabajar con listas enlazadas implementadas con punteros.
- 15.2.** Diseñar una clase genérica para poder tratar matrices.
- 15.3.** Definir una función plantilla que devuelva el valor absoluto de cualquier tipo de dato incorporado o predefinido pasado a ella.

PROBLEMAS PROPUESTOS

- 15.1.** Escribir el código de una clase C++ que lance excepciones para cuantas condiciones estime convenientes. Utilizar una cláusula *catch* que utilice una sentencia *switch* para seleccionar un mensaje apropiado y terminar el cálculo. (Nota: Utilice un tipo enumerado para listar las condiciones *enum error_pila {overflow, underflow, ...}.*)
- 15.2.** Escribir una clase plantilla que pueda almacenar una pequeña base de datos de registros, y escribir un programa que pueda manejar objetos de la base de datos.
- 15.3.** Modificar el programa del problema anterior para tratar las excepciones que considere oportunas.
- 15.4.** Escribir una clase genérica para el tratamiento de conjuntos.
- 15.5.** Escribir un programa que utilice la clase genérica del problema anterior y que trabaje con enteros, con reales y cadenas de caracteres.
- 15.6.** Modificar la clase genérica de tratamiento de conjuntos para permitir el tratamiento de excepciones
- 15.7.** Escribir un programa que utilice la clase genérica del problema anterior y realice un tratamiento de las excepciones.
- 15.8.** Escribir una clase *Racional* para números racionales. Un número racional es un número que puede ser representado como el cociente de dos enteros. Por ejemplo, $1/2$, $3/4$, $64/2$, etc. Son todos números racionales. Sobrecargar los restantes de los siguientes operadores de modo que se apliquen al tipo *Racional*: *==*, *<*, *<=*, *>*, *>=*, *+*, *-*, *** y */*.
- 15.9.** Describir una clase *Punto* que defina la posición de un punto en un plano cartesiano de dos dimensiones. Diseñar e implementar una clase *Punto* que incluya la función operador *(-)* para encontrar la distancia entre dos puntos.
- 15.10.** Crear la clase matriz cuadrada y crear un operador función que permita sumar, restar y multiplicar dos matrices.
- 15.11.** Definir los operadores necesarios para realizar operaciones con cadenas tales como sumar (concatenar), comparar o extraer cadenas.
- 15.12.** Escribir una clase *Vector* con *n* componentes que permita sumar y restar vectores, así como calcular el *producto escalar* de dos vectores de igual longitud (u_1, u_2, \dots, u_n) y (v_1, v_2, \dots, v_n) se puede definir como la suma.
- $$\sum_{k=1}^n u_k v_k$$

CAPÍTULO 16

Flujos y archivos

Introducción

La biblioteca estándar de C++ proporcionan un conjunto amplio de capacidades de entrada/salida. El software de C++ incluye una biblioteca estándar que contiene funciones que son utilizadas por los programadores de C++. Sin embargo, C++ introdujo el archivo `iostream`¹ que implementa sus propias colecciones de funciones de entrada/salida. En este capítulo aprenderá a utilizar las características típicas de E/S (Entrada/Salida) de C++ y obtener el máximo rendimiento de las mismas.

16.1. Flujos (*streams*)

La entrada/salida de C++ se produce por flujos de bytes. Un flujo es una abstracción que se refiere a una interfaz común a diferentes dispositivos de entrada y salida de una computadora. Un **flujo** (*stream*) es simplemente, una secuencia de bytes. En las operaciones de entrada, los bytes fluyen desde un dispositivo a la memoria principal. En operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo. Existen dos formas de flujo: texto y binario. Los flujos de texto se usan con caracteres ASCII, mientras que los flujos binarios se pueden utilizar con cualquier tipo de datos.

El archivo `iostream` declara tres clases para los flujos de entrada y salida estándar. La clase `istream` es para entrada de datos desde un flujo de entrada, la clase `ostream` es para la salida de datos a un flujo de salida y la clase `iostream` es para entrada de datos dentro de un flujo de datos. El archivo `fstream` declara las clases `ifstream`, `ofstream` y `fstream` para operaciones de entrada y salida por archivos. Estas clases declaran también los cuatro objetos ya conocidos.

<code>cin</code>	Un objeto de la clase <code>istream</code> conectado a la entrada estándar.
<code>cout</code>	Un objeto de la clase <code>ostream</code> conectado a la salida estándar.
<code>cerr</code>	Un objeto de la clase <code>ostream</code> conectado al error estándar para salida sin búfer.
<code>clog</code>	Un objeto de la clase <code>ostream</code> conectado al error estándar, con salida a través de búfer.

El archivo de cabecera `iostream` se incluye en la mayoría de los programas de C++ y contiene las funcionalidades requeridas para todas las operaciones de entrada y salida básicas. El archivo de cabecera `iomanip` contiene información útil para realizar la entrada y salida con formato con manipuladores de flujo parametrizados. La Tabla 16.1 muestra los objetos de flujos estándar en la biblioteca `iostream`.

¹ En las versiones anteriores al último estándar de C++, el archivo de Cabecera empleado se denomina `iostream.h` (véase Capítulo 1).

Tabla 16.1. Objetos estándar de flujo

Nombre	Operador	Clase de flujos	Significado
cin	>>	istream	entrada estándar (con búfer)
cout	<<	ostream	salida estándar (con búfer)
cerr	<<	ostream	error estándar (sin búfer)
clog	<<	ostream	error estándar (con búfer)

16.2. Las clases istream y ostream

Las clases `istream` y `ostream` se derivan de la clase base `ios`.

```
class istream:virtual public ios{...}
class ostream:virtual public ios{...}
```

La clase `istream` permite definir un flujo de entrada y soporta métodos para entrada formateada y no formateada. El operador de extracción `>>` está sobrecargado para todos los tipos de datos integrales de C++, haciendo posible las operaciones de entrada de alto nivel, aplicándose a un objeto `istream` y un objeto variable.

```
objeto_istream >> objeto_variable;
cin >> precio >> cantidad;           // lee el precio y la cantidad.
```

El operador `>>` extrae una secuencia de caracteres correspondientes a un valor del tipo `objeto_variable` de `objeto_istream`, por lo que normalmente se conoce como *operador de extracción*, debido a que su comportamiento es extraer valores de una clase `istream`.

La clase `ostream` mantiene, esencialmente variables booleanas denominadas indicadores para cada uno de los estados

Tabla 16.2. Estado de errores en el flujo

Llamada a la función	Devuelve verdadero si y sólo si
<code>cin.good()</code>	Todo bien en <code>istream</code>
<code>cin.bad()</code>	Algo incorrecto en <code>istream</code>
<code>Cin.fail()</code>	La última operación no se ha terminado

La función miembro `clear()` se utiliza normalmente para restaurar un estado de flujo a `good` de modo que la E/S se pueda proceder en ese flujo. (Una vez que el operador `good` se pone a falso, ninguna operación de entrada posterior se puede ejecutar hasta que se limpie su estado.)

La función miembro `ignore()` reinicia los indicadores de estado.

```
cin.clear();           //reinicializa good a verdadero; bad y fail a falso
cin.ignore( 123, ' ') // salta como máximo 123 caracteres, o hasta blanco
```

EJEMPLO 16.1. Función miembro clear() y control de errores de flujo de entrada.

El siguiente programa lee datos enteros de la entrada contándolos, hasta que se pulsa `control + z` en cuyo momento escribe la media parcial. Posteriormente, limpia la entrada y vuelve a leer datos de entrada hasta que se vuelve a pulsar `control + z`. Por último escribe la media total.

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

```

int main(int argc, char *argv[])
{
    int n, suma = 0, c = 0;
    while(cin >> n)
    {
        suma+= n;
        c++;
    }
    cout << " la media parcial es: " << (float)suma/c << endl;
    cin.clear();
    while(cin >> n)
    {
        suma+= n;
        c++;
    }
    cout << " la media total es " << (float) suma/c << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

La clase `ostream` de C++ proporciona la capacidad de realizar salida formateada y sin formatear. Esta clase sirve para llevar el flujo de caracteres desde un programa en ejecución a un dispositivo de salida arbitrario.

`cout`, un `ostream` para visualizar la salida normal.

`cerr` un `ostream` para visualizar mensajes de error o de diagnóstico.

El operador de inserción de flujos es `<<`, se aplica a un objeto `ostream` y una expresión:

```
objeto_ostream << expresión;
```

Este operador de inserción se puede poner en cascada, de modo que pueden aparecer varios elementos en una sola sentencia de C++, mezclando expresiones, numéricas, alfanuméricas, etc., pero sin poner blancos de separación.

EJEMPLO 16.2. *Uso de cout.*

```

#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    cout << " Mazarambroz se encuentra a " << 100;
    cout << " km de Toledo. Tiene " << 1500 << " habitantes " << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

La biblioteca `iostream`, que contiene entre otras las clases `istream` y `ostream`, proporciona manipuladores que permiten formatear el flujo de salida. La entrada y salida de datos realizada con los operadores de inserción y extracción pueden formatearse ajustándola a la izquierda o derecha, proporcionando una longitud mínima o máxima, precisión, etc. Normalmente, los manipuladores se utilizan en el centro de una secuencia de inserción o extracción de flujos. Casi todos los manipuladores están incluidos fundamentalmente en el archivo de cabecera `iomanip.h`. La Tabla 16.3 recoge los manipuladores más importantes de flujo de entrada y salida.

Tabla 16.3. Manipuladores de flujo

Manipulador	Descripción
endl	Inserta nueva línea y limpia ostream.
ends	Inserta el byte nulo en ostream.
flush	Limpia ostream.
dec	Activa conversión decimal.
oct	Activa conversión octal.
hex	Activa conversión hexadecimal.
left	Justifica salida a la izquierda del campo.
right	Justifica salida a la derecha del campo.
setbase(int b)	Establece base a b. Por defecto es decimal.
setw(arg)	En la entrada, limita la lectura a arg caracteres; en la salida utiliza campo arg como mínimo.
setprecision(arg)	Fija la precisión de coma flotante a arg sitios a la derecha del punto decimal.
setiosflags(fmtflags f)	Establece los indicadores de ios en f.
resetiosflags(fmtflags f)	Borra indicadores ios en f.
setfill(char car)	Usa car para llenar caracteres (por omisión blanco).

EJEMPLO 16.3. Uso de algunos manipuladores de flujo.

```
#include <cstdlib>
#include <iostream>
#include <iomanip.h>
using namespace std;

int main(int argc, char *argv[])
{
    const int n =15;
    float dato = 123.4567;
    cout << n << endl;                                // salida en base diez
    cout << oct << n<< endl;                          // salida en base ocho
    cout << hex<< n << endl;                          // salida en base 16
    cout << setw(8) << " hola" << endl;                // anchura de campo 8
    cout << setw(10);                                  // anchura 10
    cout.fill('#');                                    // rellena con #
    cout << 34 << endl;      // escribe 34 en base 16 anchura 10 y relleno
    cout<< setprecision(5) << dato << endl;        // precision 5 3 antes del
                                                    // punto y 2 después
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Lectura de datos tipo cadena

Cuando se utiliza el operador de extracción para leer datos tipo cadena, se producen anomalías si las cadenas constan de más de una palabra separadas por blancos, ya que el operador `>>` hace que termine la operación de lectura siempre que se encuentre cualquier espacio en blanco. Para resolver este problema se puede usar la función `get()` o bien la función `getline()`.

La función miembro `get()` del objeto `cin` de la clase `istream`, lee un sólo carácter de datos del teclado, o bien una cadena de caracteres.

Si esta función se usa con un parámetro carácter , entonces lee el siguiente carácter del flujo de entrada en su parámetro que se pasa por referencia. La función de vuelve falso cuando se detecta el final de archivo. Su prototipo es:

```
istream&get(car &c)
```

EJEMPLO 16.4. *Uso de la función get(), para leer carácter a carácter.*

El programa lee caracteres hasta que se teclea *control +z*. Posteriormente lo visualiza.

```
#include <cstdlib>
#include <iostream>

using namespace std;
int main(int argc, char *argv[])
{
    char c;
    cout << "comienza entrada de cadena\n";
    while( cin.get(c))
        cout << c;
    cout << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

La función *get()* tiene un formato que permite leer cadenas de caracteres, cuyo prototipo es:

```
istream &get(char*bufer, int n char sep='\n')
```

El primer parámetro es un puntero a un array de caracteres; el segundo parámetro es el número máximo de caracteres a leer más uno; y el tercer parámetro es el carácter de terminación, si no aparece es *intro*.

EJEMPLO 16.5. *Uso de la función get(), para leer cadenas de caracteres.*

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char cadena[20];
    cout << " introduzca cadena de caracteres\n";
    cin.get(cadena, 20);
    cout << cadena<< endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

La función miembro de la clase *iostream* *getline()*. Tiene la siguiente sintaxis:

```
istream& getline(signed char* buffer, int long, char separador = '\n' );
```

El primer argumento de la función miembro *getline()* es el identificador de una variable cadena, en la que se retorna la cadena que se lee. El segundo argumento es el número máximo de caracteres que se leerán que debe ser al menos dos caracteres mayor que la cadena real, para permitir el carácter nulo '*\0*' y el '*\n*'. Por último, el carácter separador se lee y almacena como el siguiente al último carácter de la cadena y es el último carácter que se lee.

Reglas

- La llamada *cin.getline(cad, n, car)* lee todas las entradas hasta la primera ocurrencia del carácter separador *car* en *cad*.
- Si el carácter especificado *car* es el carácter de nueva línea '*\n*', la llamada anterior es equivalente a *cin.getline(cad, n)*.

La función `getline()` presenta problemas cuando se intente utilizar una variable cadena, después de que se ha utilizado `cin` para leer una variable carácter o numérico. La razón es que no se lee el retorno de carro por lo que permanece en el buffer de entrada. Cuando se da la orden `getline()`, se lee y almacena este retorno que por defecto es el carácter separador , por lo que no se lee la cadena de caracteres que se pretende sea el dato. Para resolver este problema se puede dar una orden de lectura con `getline(cad, 2)` y se limpia el buffer de entrada.

EJEMPLO 16.6. Uso de la función `getline()`.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    char cadena[20];
    int valor;
    cout << " introduzca cadena \n";
    cin.getline(cadena, 20);
    cout << " cadena leida :" << cadena << endl;
    cout << " introduzca dato numerico:\n";
    cin >> valor;
    cout << " valor leido: " << valor << endl;
    cin.getline(cadena, 2);
    cout << " introduzca nueva cadena \n";
    cin.getline(cadena, 20);
    cout << " nueva cadena leida: " << cadena << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

16.3. Las clases `ifstream` y `ofstream`

Las clases que manejan los archivos en C++ son: `ifstream` para la entrada (solo flujos de lectura), `ofstream` para la salida (sólo flujos de salida) y `fstream`, para manejar tanto la entrada como la salida (flujos de entrada y salida).

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 vuelve 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.

16.3.1. 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 método para abrir un archivo es `fopen()`, y su sintaxis de llamada:

```
void open(char *nombre_archivo, int modo);

nombre ≡ cadena      Contiene el identificador externo del archivo.
modo                  Contiene el modo en que se va a tratar el archivo. Puede ser uno o una combinación del tipo enumerado open_mode, de la clase ios:: separados por |.
```

`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 en **Modo** en la Tabla 16.4.

Tabla 16.4. Modos de apertura de un archivo

Modo	Significado
<code>ios::in</code>	Abre para lectura, situándose al principio.
<code>ios::out</code>	Abre para crear nuevo archivo (si ya existe se pierden sus datos). Se sitúa al principio.
<code>ios::app</code>	Abre para añadir al final.
<code>ios::ate</code>	Abre archivo ya existente para modificar (leer/escribir), situándose al final.
<code>ios::trunc</code>	Crea un archivo para escribir/leer (si ya existe se pierden los datos).
<code>ios::binary</code>	Archivo binario.

Para abrir un archivo en modo binario hay que especificar la opción `ios::binary` en el modo. Los archivos binarios son secuencias de bytes y optimizan el espacio, sobre todo con campos numéricos. Así, al 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 según el formato y después se escribe en el archivo.

Al terminar la ejecución del programa podrá ocurrir que existan 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 miembro `fclose()` cierra el archivo asociado a una clase. El prototipo es: `void fclose();`

16.3.2. FUNCIONES DE LECTURA Y ESCRITURA EN FICHEROS

El operador de flujo de entrada `<<` se puede usar con flujos de entrada, al igual que se hace con `cin`, cuando se trabaje en modo texto (no binario).

El operador de flujo de salida `>>` puede ser usado con flujos de salida al igual que se hace con `cout` cuando se trabaje en modo texto.

El método de salida `fput()` sirve para insertar un carácter en cualquier dispositivo de salida. Su sintaxis es `ostream& put(char c);`

Los métodos de entrada `get()` y `getline()` para la lectura de datos de tipo cadena funcionan tal y como se ha explicado para el caso de la lectura del objeto `cin`.

El método `feof()` verifica si se ha alcanzado el final del archivo, devuelve un valor nulo si no es así. El prototipo de la función es el siguiente: `int feof();`

Cada vez que se produzca una condición de error en un flujo es necesario eliminarla, ya que en caso contrario ninguna operación que se realice sobre él tendrá éxito. La función miembro `clear()` se utiliza normalmente para restaurar un estado de flujo a `good` de modo que la E/S se pueda proceder en ese flujo, tal y como aparece expresado en la Tabla 16.2

EJEMPLO 16.7. Tratamiento de archivos de secuencia.

El programa solicita al usuario un archivo de texto, lo almacena y posteriormente lo visualiza.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    ofstream fichero;
    ifstream fichero1;
    char c;
    fichero.open("datos.cpp", ios::out);
```

```

cout << " escriba fichero de texto: control+z para terminar\n";
while( cin.get(c))
    fichero.put(c);
fichero.close();

cout << " lectura y escritura del fichero anterior\n";

ficherol.open("datos.cpp", ios::in);
while(ficherol.get(c))
    cout.put(c);
ficherol.close();
system("PAUSE");
return EXIT_SUCCESS;
}

```

El método `write()` escribe el número de caracteres indicado en el segundo tamaño desde cualquier tipo de dato suministrado por el primero. El prototipo de la función es:

```
ostream & write(const char *buffer, int tamaño);
```

El método `read()` lee el número de caracteres indicado en el parámetro tamaño dentro del *buffer* de cualquier tipo de dato suministrado por el primer parámetro. El prototipo de la función es:

```
ostream & read(char * buffer, int tamaño);
```

La extracción de caracteres continúa hasta que se hayan leído tamaño caracteres o se encuentre el fin de archivo de entrada.

Para poder leer y escribir registros o estructuras con los métodos `read` o `write` se usa el operador `reinterpret_cast`. Su sintaxis es:

```
reinterpret_cast< Tipo > (arg)
```

El operando `Tipo` es en este caso `char *` y `arg` es `®istro`, donde `registro` es, normalmente, una variable del tipo `registro` que se desea leer o escribir. (Véase el Problema resuelto 16.2.)

Los métodos `seekg()` y `seekp()` permiten tratar un archivo en C++ como un array que es una estructura de datos de acceso aleatorio. `seekg()` y `seekp()` sitúan el puntero del archivo de entrada o salida respectivamente, en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos.

Los prototipos de los métodos son:

```

istream & seekg (long desplazamiento);
istream & seekg (long desplazamiento, seek_dir origen);

ostream & seekp (long desplazamiento);
ostream & seekp (long desplazamiento, seek_dir origen);

```

El primer formato de ambos métodos cambia la posición de modo absoluto, y el segundo lo hace de modo relativo, dependiendo de tres valores.

El primer parámetro indica el desplazamiento, absoluto o relativo dependiendo del formato.

El segundo parámetro de los dos métodos es el origen del desplazamiento. Este origen puede tomar los tres siguientes valores:

<code>beg</code>	Cuenta desde el inicio del archivo.
<code>cur</code>	Cuenta desde la posición actual del puntero al archivo.
<code>end</code>	Cuenta desde el final del archivo.

La posición actual del cursor de un archivo se puede obtener llamando a los métodos `tell()` o `tellp()`. El primer método la obtiene para archivos de entrada o el segundo para salida. Sus prototipos son:

```
long int tellg();
long int tellp();
```

El método `gcount()` retorna el número de caracteres o bytes que se han leído en la última lectura realizada en un flujo de entrada. Su formato es

```
int gcount();
```

EJEMPLO 16.8. *Tratamiento de archivos binario con los métodos read y write.*

El programa lee un archivo aleatorio binario `datos.dat` del que no se tiene ninguna información, y lo copia en otro archivo binario aleatorio `res.dat`.

```
#include <cstdlib>
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream f;                                // archivo de entrada
    ofstream g;                                // archivo de salida

    char cadena[81];                           // cadena de 81 bytes para leer de un fichero y
                                                // escribir en el otro
                                                // Abrir el archivo de f de lectura y binario
    f.open("datos.dat", ios::binary);
    if(!f.good())
    {
        cout <<"El archivo " << "datos.dat " << " no existe \n";
        exit (1);
    }
    // Crear o sobreescribir el archivo g en binario
    g.open("res.dat", ios::binary);
    if(!g.good())
        cout <<"El fichero " << "res.dat " << " no se puede crear\n";
    f.close();
    exit (1);
}
do
{
    f.read(cadena, 81);                         // lectura de f
    g.write(cadena, f.gcount());                // escritura de los bytes leídos
} while(f.gcount() > 0);                      // repetir hasta que no se lean datos
f.close();
g.close();
system("PAUSE");
return EXIT_SUCCESS;
}
```

EJERCICIOS

- 16.1. 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.
- 16.2. Un archivo de texto contiene enteros positivos y negativos. Utiliza operador de extracción `>>` para leer el archivo, visualizarlo y determinar el número de enteros negativos y positivos que tiene.
- 16.3. Escribir un programa que lea un texto del teclado lo almacene en un archivo binario y posteriormente vuelva a leer el archivo binario y visualice su contenido.
- 16.4. 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.

PROBLEMAS

- 16.1. Utilizar 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 contar las veces que ocurre la máscara en el archivo.
- 16.2. 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. Los datos deben ser leídos del teclado. Una vez escrito en el archivo los datos, el programa debe visualizar el contenido completo del archivo.

SOLUCIÓN DE LOS EJERCICIOS

- 16.1. Éstas son las operaciones básicas para realizar la apertura de un archivo que puede ser de entrada o de salida, luego debe ser un objeto de la clase `fstream`. Se trata de declarar una cadena de caracteres para leer el nombre del archivo, y abrir el archivo en modo entrada. En caso de que no pueda abrirse debe crear de escritura. 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 quedarse fuera de nuestro control esta operación se realiza comprobando el valor de `good`.

La codificación de este ejercicio se encuentra en la página Web del libro.

- 16.2. Una vez que se conoce el funcionamiento del operador de extracción `>>` para la clase `cout`, es muy sencillo su uso para cualquier objeto de la clase `fstream`, ya que su funcionamiento es el mismo. El archivo 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 el operador de extracción `>>` cuando tiene una variable de tipo entero en su parte derecha. Es la misma operación de conversión que reali-

za el operador de extracción cuando lee secuencias de códigos de teclas desde la entrada estándar. El programa utiliza dos contadores para contar los positivos y negativos y una variable para leer el dato del archivo. El programa visualiza además el archivo.

La codificación de este ejercicio se encuentra en la página Web del libro.

- 16.3.** Se define un objeto archivo de clase `fstream` que puede ser abierto para leer o para escribir en modo binario. Se lee el texto de la pantalla carácter a carácter (el final del texto de entrada viene dado por control + z), y se escribe en el archivo binario también carácter a carácter. Posteriormente, lo cierra el archivo binario, se vuelve a abrir en modo lectura, para leerlo byte a byte y presentarlo en pantalla.

La codificación de este ejercicio se encuentra en la página Web del libro.

- 16.4.** 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 con virtiéndolo a su código ASCII y en el archivo binario se vuelve a escribir de la memoria sin realizar ninguna transformación. En el programa codificado si se produce un error en la apertura de uno de los dos archivos, se retorna el control al sistema operativo.

La codificación de este ejercicio se encuentra en la página Web del libro.

SOLUCIÓN DE LOS PROBLEMAS

- 16.1.** La organización de un archivo de tipo texto es la siguiente: 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 en 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 lee cada línea de un archivo de texto en la variable de tipo cadena `linea` y buscar en ella la ocurrencia de una subcadena que se denomina `cadena_a_buscar`, recibida en `arg[1]`, para lo cual se utiliza la función estándar de C++ `strstr` de tratamiento de cadenas. El nombre del archivo es recibido en el segundo parámetro de la función `main()` `arg[2]`.

La codificación de este problema se encuentra en la página Web del libro.

- 16.2.** Los datos que hay que almacenar son definidos por el usuario por lo que deben ser estructuras. Es decir, deben ser de tipo `struct`; una de ellas debe definir el registro de entrenamiento, y la otra el registro del pulsómetro. Se usa un archivo binario `f` para almacenar los datos, por lo que como no se tiene ninguna marca para distinguir la información almacenada en ellos son los programas los que deben leer con las mismas estructuras con que se escribieron. En este caso, se escribe en primer lugar la estructura del registro del entrenamiento y, posteriormente, en un bucle `for` las estructuras que almacenan los distintos registros del pulsómetro. La lectura de los datos del archivo, es realizada de la misma forma que ha sido creado para que los datos tengan el sentido con el que inicialmente fueron escritos.

La codificación de este ejercicio se encuentra en la página Web del libro.

PROBLEMAS PROPUESTOS

- 16.1.** 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.
- 16.2.** Escribir un programa que visualice por pantalla las líneas de texto de un archivo, numerando cada línea del mismo.
- 16.3.** Escribir un programa para listar el contenido de un determinado subdirectorio, pasado como parámetro a la función `main()`.
- 16.4.** 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 (celular), 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.
- 16.5.** 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 la referencia para la ordenación de los registros.
- 16.6.** 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 se tienen todos los archivos parciales ordenados se van 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.
- 16.7.** 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 la nota final como media del resto de las calificaciones.
- 16.8.** 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 reconstruir 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.
- 16.9.** 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 el 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 descendente por la cantidad total pagada.
- 16.10.** Se quiere escribir una carta de felicitación 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 encuentran 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`.

## CAPÍTULO 17

# Listas enlazadas

## Introducción

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.

### 17.1. Fundamentos teóricos

Una **lista** es una secuencia de 0 o más elementos, de un tipo de dato determinado almacenados en memoria. Son estructuras lineales, donde cada elemento de la lista, excepto el primero, tiene un único predecesor y cada elemento de la lista, excepto el último, tiene un único sucesor. El número de elementos de una lista se llama longitud. Si una lista tiene 0 elementos se denomina *lista vacía*. Es posible considerar distintos tipos de listas, contiguas y enlazadas:

**Contiguas.** Los elementos son adyacentes en la memoria de la computadora y tienen unos límites, izquierdo y derecho, que no pueden ser rebasados cuando se añade un nuevo elemento. Se implementan a través de arrays. La inserción o eliminación de un elemento en la lista suele implicar la traslación de otros elementos de la misma.

**Enlazadas.** Los elementos se almacenan en posiciones de memoria que no son contiguas o adyacentes, por lo que cada elemento necesita almacenar la posición o dirección del siguiente elemento de la lista. Son mucho más flexibles y potentes que las listas contiguas. La inserción o supresión de un elemento de la lista no requiere el desplazamiento de otros elementos de la misma. 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*, *Sig*, etc.) que

apunta al siguiente elemento de la lista. La representación gráfica es la siguiente. Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos.

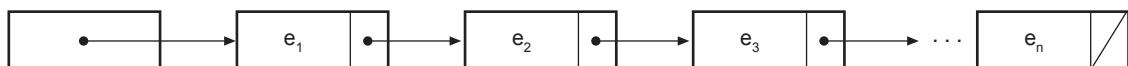


Figura 17.1. Lista enlazada (representación gráfica típica).

El nodo último ha de ser representado de forma diferente para significar que este nodo no se enlaza a ningún otro.

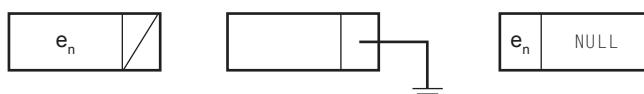


Figura 17.2. Diferentes representaciones gráficas del nodo último.

## 17.2. Clasificación de las listas enlazadas

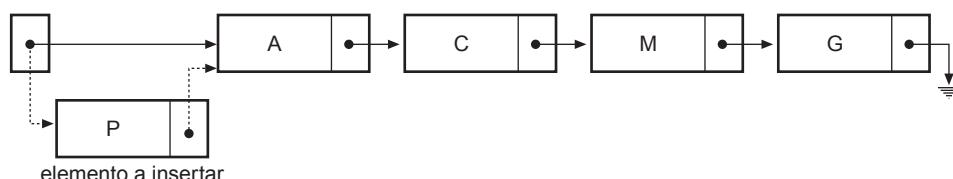
Las listas se pueden dividir en cuatro categorías:

- **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. Su recorrido puede realizarse tanto de frente a final como de final a frente.
- **Lista circular simplemente enlazada.** Es una modificación de la lista enlazada en la cual el puntero del último elemento apunta al primero de la lista.
- **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.

**EJEMPLO 17.1.** Lista enlazada implementada con punteros y con arrays a la que se le añade un primer elemento.

**Implementación con punteros:**



**Implementación con arrays:**

inicio = 3, elemento a insertar P

Antes del "A", es decir antes del primero de la lista

|   | Info | Sig |
|---|------|-----|
| 1 | C    | 4   |
| 2 |      |     |
| 3 | A    | 1   |
| 4 | M    | 6   |
| 5 |      |     |
| 6 | G    | 0   |

inicio = 2

Nuevo elemento

|   | Info | Sig |
|---|------|-----|
| 1 | C    | 4   |
| 2 | P    | 3   |
| 3 | A    | 1   |
| 4 | M    | 6   |
| 5 |      |     |
| 6 | G    | 0   |

Como se puede observar, tanto cuando se implementa con punteros como cuando se hace a través de arrays, la inserción de un nuevo elemento no requiere el desplazamiento de los que le siguen. Para observar la analogía entre ambas implementaciones se puede recurrir a representar la implementación con arrays de la siguiente forma:



## 17.3. Operaciones en listas enlazadas

Las operaciones sobre listas enlazadas más usuales son: *Declaración de los tipos nodo, puntero a nodo y clase nodo ; Declaración de clase lista, inicialización o creación; insertar elementos en una lista; b usar elementos de una lista; eliminar elementos de una lista; recorrer una lista enlazada; comprobar si la lista está vacía.*

### 17.3.1. DECLARACIÓN DE LOS TIPOS NODO Y PUNTERO A NODO Y CLASE NODO

En C++, se puede definir un nuevo tipo de dato por un nodo mediante las palabras reservadas `struct` o `class` que contiene las dos partes citadas.

```

struct Nodo
{
 int dato;
 Nodo *enlace;
};

class Nodo {
public:
 int dato;
 Nodo *enlace;
 // constructor
};

```

#### EJEMPLO 17.2. Declaración de un nodo en C++.

En C++, se puede declarar un nuevo tipo de dato por un nodo mediante la palabra reservada `class` de la siguiente forma:

```

class Nodo
{
public:
 int info;
 Nodo* sig;
};

typedef class Nodo
{
public:
 int info;
 Nodo *sig;
}NODO;

typedef double Elemento
class Nodo
{
public:
 Elemento info;
 Nodo *sig;
};

```

#### EJEMPLO 17.3. Declaración e implementación de la clase nodo que contiene una información de tipo elemento y el siguiente de tipo puntero a nodo.

La clase `Nodo` tiene dos atributos protegidos que son el elemento `e`, y `Sig` que es un puntero a la propia clase `Nodo`. Ambos atributos sirven para almacenar la información del nodo, y la dirección del siguiente nodo. Se declaran como funciones miembro tres constructores. Estos constructores son: el constructor por defecto; el constructor que inicializa el atributo `x` del `Nodo` al valor de `x`, y pone el atributo `Sig` a `NULL`; el constructor, que inicializa los dos atributos del `Nodo`. Las funciones miembro encargadas de Obtener y Poner tanto el elemento `e` como el puntero `Sig` son: `TElemento OE(); void PE(TElemento e); Nodo * OSig(); y void PSig(Nodo *p);`. Por último, la función miembro destructor, se declara por defecto. El tipo elemento de la clase `Nodo` es un entero, pero al estar definido en un `typedef`, puede cambiarse de una forma sencilla y rápida para que almacene cualquier otro tipo de información.

```

#include <cstdlib>
#include <iostream>
using namespace std;

```

```

typedef int Telemento;

class Nodo
{
protected:
 Telemento e;
 Nodo *Sig;
public:
 Nodo(){}
 // constructor vacío
 Nodo (Telemento x);
 // constructor
 Nodo(Telemento x, Nodo* s);
 // Constructor
 ~Nodo(){}
 // destructor por defecto
 Telemento OE();
 // Obtener elemento
 void PE(Telemento e);
 // Poner elemento
 Nodo * OSig();
 // Obtener siguiente
 void PSig(Nodo *p);
 // Poner siguiente
};

Nodo::Nodo(Telemento x)
{
 e = x; // constructor que inicializa e a x y Sig a Null
 Sig = NULL;
}

Nodo(Telemento x, Nodo* s)
{
 e = x; // constructor que inicializa e a x y Sig a s
 Sig = s;
}

Telemento Nodo::OE()
{
 return e; // obtiene una copia del atributo e
}

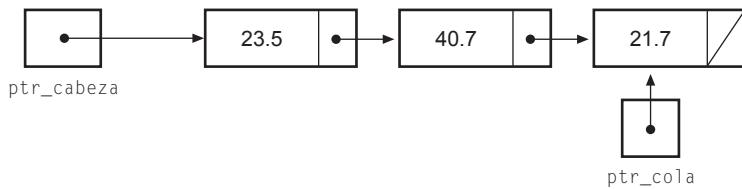
void Nodo::PE(Telemento x)
{
 e = x; // pone el atributo e a x
}

Nodo* Nodo::OSig()
{
 return Sig; // obtiene una copia del atributo Sig
}

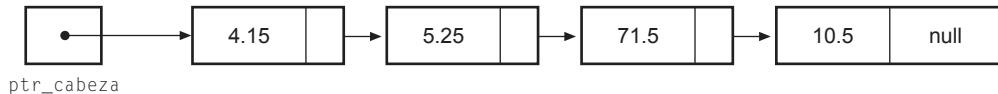
void Nodo:: PSig(Nodo *p)
{
 Sig = p; // Pone el atributo Sig a p
}

```

**Puntero de cabecera y cola.** Un puntero al primer nodo de una lista se llama **puntero cabeza** (Nodo \*ptr \_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** (Nodo \*ptr\_col;). Cada puntero a un nodo debe ser declarado como una variable puntero.



**El 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 v acía.



**El 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`

### 17.3.2. DECLARACIÓN DE CLASE LISTA, INICIALIZACIÓN O CREACIÓN

La clase lista se declara como una clase que contiene un solo atributo protegido `p` que es un puntero a la clase `Nodo` (Agregación lógica). Como miembros de la clase lista, se declaran el constructor, el destructor y dos funciones miembros para obtener y poner el puntero a la lista.

**EJEMPLO 17.4.** Clase `lista` simple con constructor, destructor, y funciones miembro para poner y obtener el puntero al primer nodo de la lista.

```

#include <cstdlib>
#include <iostream>
using namespace std;

class ListaS //clase lista simplemente enlazada
{
protected:
 Nodo *p;
public:
 ListaS(); // constructor
 ~ListaS(); // destructor
 Nodo * Op(); // Obtener el puntero
 void Pp(Nodo *pl); // Poner el puntero
 // otras funciones miembro
};

ListaS::ListaS()
{
 p = NULL;
}

ListaS::~ListaS(){}

```

```

Nodo * ListaS:: Op()
{
 return p;
}

void ListaS:: Pp(Nodo *p1)
{
 p = p1;
}

int main(int argc, char *argv[])
{
 ListaS l1;

 system("PAUSE");
 return EXIT_SUCCESS;
}

```

### 17.3.3. INSERTAR ELEMENTOS 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: en la cabeza de una lista; no en la cabeza de lista; al final de la lista que es un caso particular de la inserción no en la cabeza de la lista.

El proceso de inserción se puede resumir en este algoritmo:

- Asignar un nuevo nodo a un puntero `aux` que apunte al nuevo nodo que se va a insertar en la lista, y situar el nuevo elemento en atributo `e` del nuevo nodo.
- Hacer que el campo enlace `Sig` del nuevo nodo apunte al primer nodo de la lista apuntado por `p`.
- Hacer que el puntero `p` que apunta al primer elemento de la lista apunte al nuevo nodo que se ha creado `aux`.

**EJEMPLO 17.5.** Función miembro de la clase `ListaS` que añade un nuevo elemento `e` a la lista.

Se supone las declaraciones de los Ejemplos 17.3 y 17.4 de las clases `Nodo` y `ListaS`. En la zona pública de la clase `ListaS` se debe incluir el prototipo de la función miembro.

```

void AnadePL(Telemento e); // prototipo dentro de la clase ListaS

void ListaS::AnadePL(Telemento e) // código
{
 Nodo *aux;
 aux = new Nodo(e); // reserva de memoria, y asigna e
 aux->PSig(p); // poner p en el siguiente de aux
 p = aux; // Pp(aux); poner p a aux
}

```

#### 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 `insertar_ptr`, y situar el nuevo elemento `x` como atributo del nuevo nodo.
- Hacer que el campo enlace `Sig` del nuevo nodo `insertar_ptr` apunte al nodo que va después de la posición del nuevo nodo.

- En la variable puntero anterior\_ptr hay que tener la dirección del nodo que está antes de la posición deseada (siempre existe) para el nuevo nodo. Hacer que anterior\_ptr->PSig() apunte al nuevo nodo que se acaba de crear insertar\_ptr.

### 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 atributo Sig de nuevo nodo apuntado por el puntero insertar\_ptr siempre apunta a NULL.

**EJEMPLO 17.6.** Segmento de código que añade un nuevo elemento e a la lista, entre las posiciones anterior\_ptr y pos. (Si pos es NULL, es el último de la lista.)

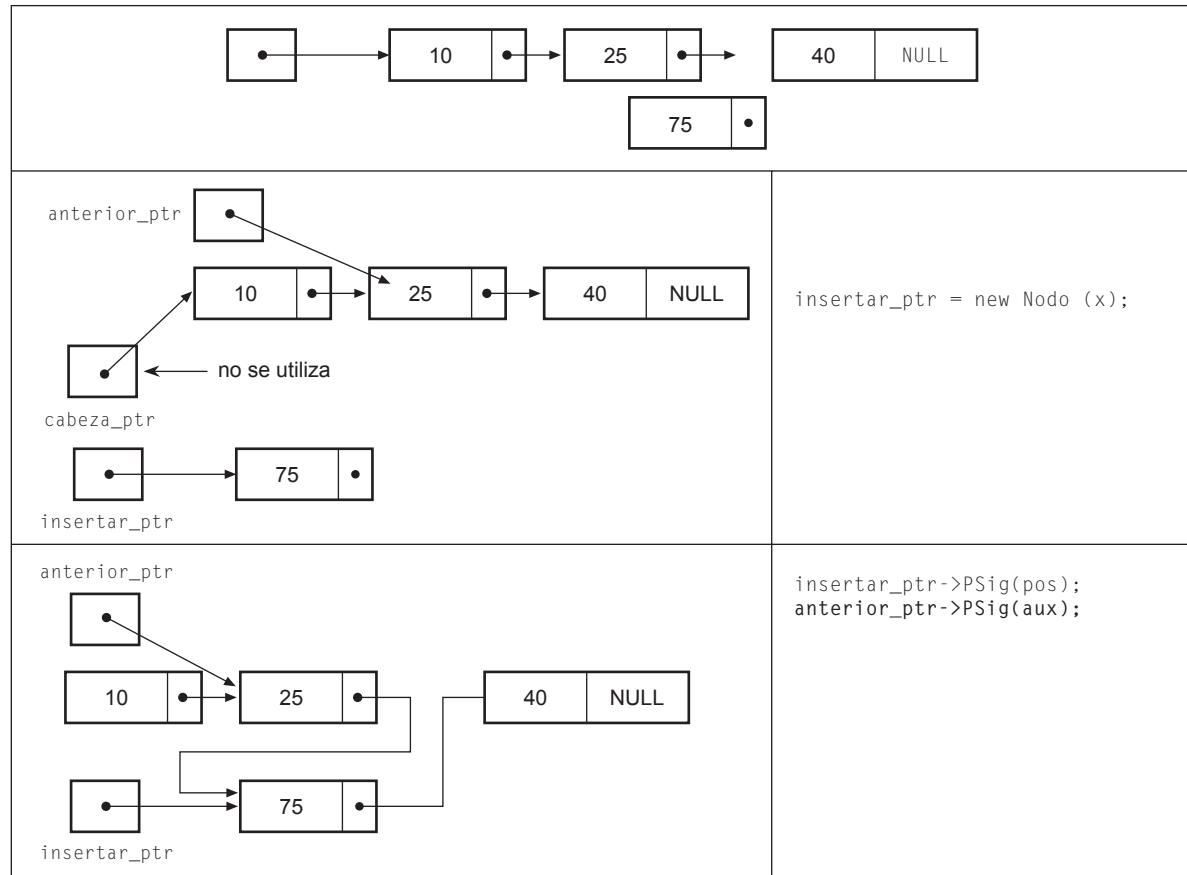
Se supone declarada la clase Nodo, y que previamente, se ha colocado el puntero a Nodo aux apuntando al nodo anterior donde hay que colocarlo (se supone que al no ser el primero, siempre existe), y el puntero a Nodo pos apuntando al nodo que va después.

```
Nodo *anterior_ptr, *pos, *insertar_ptr;

// búsqueda omitida.
insertar_ptr = new Nodo(x);
insertar_ptr->PSig(pos);
anterior_ptr->PSig(aux);
```

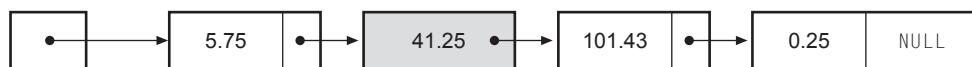
**EJEMPLO 17.7.** Insertar un nuevo elemento 35 entre el elemento 75 y el elemento 40 en la lista enlazada 10, 25, 40.

El siguiente gráfico muestra un seguimiento del código del Ejemplo 17.6.



#### 17.3.4. BUSCAR ELEMENTOS DE UNA LISTA

Dado que una función en C++ puede devolver un puntero, el algoritmo que sirva para localizar un elemento en una lista enlazada puede devolver un puntero a ese elemento.



**EJEMPLO 17.8.** Búsqueda de un elemento en una lista enlazada implementada con clases.

La función miembro Buscar retorna un puntero al nodo que contiene la primera aparición del dato `x`, en la lista en el caso de que se encuentre, `NULL` en otro caso. Se usa un puntero al nodo `pos` que se inicializa con el primer puntero de la lista `p`. Un bucle `while`, itera, mientras queden datos en la lista y no haya encontrado el elemento que se busca. Se incluye, además, la función miembro Buscar1, que realiza la misma búsqueda sin utilizar interruptor `enc` de la búsqueda y con un bucle `for`.

```

Nodo * ListaS::Buscar(Telemento x)
{
 Nodo *pos = p;
 bool enc = false;

 while (!enc && pos)
 if (pos->OE() != x)
 pos = pos->OSig();
 else enc = true;

 if (enc)// Encontrado // es equivalente a poner return pos
 return pos;
 else
 return NULL;
}

Nodo* ListaS::Buscar1(Telemento x)
{
 Nodo *pos = p;
 for (; pos != NULL; pos = pos->OSig())
 if (x == pos->OE())
 return pos;
 return NULL;
}

```

**EJEMPLO 17.9.** *Función miembro de la clase ListaS que añade el elemento x en la posición pos i que recibe como parámetro.*

Se realiza una búsqueda en la lista enlazada, quedándose además de con la posición `pos`, con un puntero al nodo anterior `ant`. El bucle que realiza la búsqueda lleva un contador que va contando el número de elementos que se recorren, para que cuando se llegue al límite `pos`, poder terminar la búsqueda. A la hora de añadir el nodo a la lista, se observa las dos posibilidades existentes en la inserción en una lista enlazada: primero de la lista o no primero (incluye al último).

```

while (pos && c < posi)
{
 ant = pos;
 pos = pos->0Sig();
 c++;
}
if(c < posi)
// se va a insertar aunque no tenga menos elementos
{
 aux->PSig(pos);
 if(ant) // centro o final de la lista
 ant->PSig(aux);
 else p = aux; // primero de la lista.
}
}
}

```

### 17.3.5. ELIMINAR ELEMENTOS DE 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 y que esté apuntado por pos. Hay que tener la dirección del nodo a eliminar y la del inmediatamente anterior ant.
- El puntero Sig del nodo anterior ant ha de apuntar al Sig del nodo a eliminar.
- Si el nodo a eliminar es el primero de la lista se modifica el atributo p de la clase lista que apunta al primero para que tenga la dirección del nodo Sig del nodo a eliminar pos.
- Se libera la memoria ocupada por el nodo pos.

#### EJEMPLO 17.10. Eliminación de un nodo de una lista implementada con un clase *ListaS*.

El código que se presenta, corresponde a la función miembro *Borrar* de la clase *ListaS* implementada en ejemplos anteriores. *void ListaS::Borrar(Telemento x)*

```

{
Nodo *ant = NULL, *pos =p;
// búsqueda del nodo a borrar
.....
// se va a borrar el nodo apuntado por pos

if(ant != NULL)
 ant->PSig(pos->0Sig()); // no es el primero
else
 p = pos->0Sig(); // es el primero
pos->PSig(NULL);
delete pos;
}

```

#### EJEMPLO 17.11. Función miembro de la clase *ListaS* que elimina la primera aparición del elemento x de la lista si es que se encuentra.

Primero, se realiza la búsqueda de la primera aparición del elemento x, quedándose, además, con un puntero ant que apunta al nodo inmediatamente anterior, en caso de que exista. Si se ha encontrado el elemento a borrar en la posición pos, se diferencian las dos posibilidades que hay en el borrado en una lista enlazada simple. Ser el primer elemento, en cuyo caso el puntero anterior apunta a NULL, o no serlo. Si el elemento a borrar es el primero de la lista, hay que mover el primer puntero de la lista p al nodo siguiente de p. En otro caso, hay que poner el puntero siguiente del nodo anterior en el puntero siguiente de la posición donde se encuentre.

```

void ListaS::BorrarL(Telemento x)
{
 Nodo *ant = NULL, *pos=p;
 bool enc = false;

 while (!enc && pos) //búsqueda
 if (pos->OE()!= x)
 {
 ant = pos;
 pos = pos->OSig();
 }
 else enc =true;

 if (enc) // borrar la primera aparición
 {
 if (ant) // no primero
 ant->PSig(pos->OSig());
 else // borrado del primero
 p= p->OSig();
 pos->PSig(NULL);
 delete pos;
 }
}

```

## 17.4. Lista doblemente enlazada

En una **lista doblemente enlazada**, cada elemento contiene dos punteros, aparte del valor almacenado en el elemento; un puntero apunta al derecho o siguiente elemento de la lista (*sig*) y el otro puntero apunta al izquierdo o elemento anterior de la lista (*ant*). La Figura 17.1 muestra una lista doblemente enlazada circular y un nodo de dicha lista.

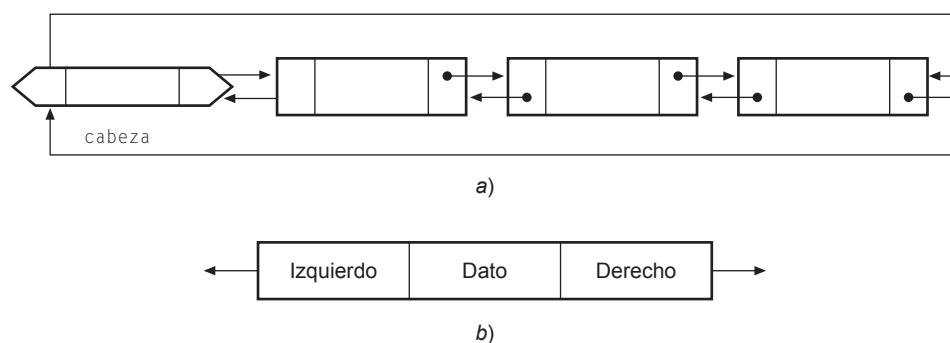
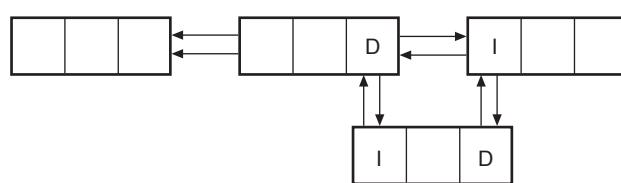
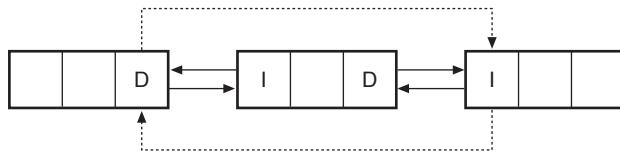


Figura 17.1. Lista doblemente enlazada circular y nodo.

Existe una operación de *insertar* y *eliminar* (borrar) en cada dirección.



Inserción de un nodo en una lista doblemente enlazada.



Eliminación de un nodo en una lista doblemente enlazada.

**EJEMPLO 17.12.** Declaración e implementación de la clase `NodoD` que contiene una información de tipo `elemento` y el siguiente y el anterior de tipo puntero o a `NodoD`.

La clase `NodoD` es muy similar a la clase `Nodo` del Ejemplo 17.2. De hecho, coincide completamente, excepto en que se añade un nuevo atributo `Ant`, como un puntero al `NodoD` y dos funciones miembros para poner el anterior y obtener el anterior. Hay que tener en cuenta, además, que se ha cambiado el nombre del nodo, por lo que cambia `Nodo` por `NodoD`.

```
#include <cstdlib>
#include <iostream>
using namespace std;
typedef int Telemento;

class NodoD {
protected:
 Telemento e;
 NodoD *Sig, *Ant ;
public:
 NodoD (Telemento x); // constructor
 Telemento OE(); // Obtener elemento
 void PE(Telemento e); // poner elemento
 NodoD * OSig(); // Obtener siguiente
 void PSig(NodoD *p); // poner siguiente
 NodoD * OAnt(); // Obtener anterior
 void PAnt(NodoD *p); // poner anterior
 ~NodoD(){};
 NodoD(){}
};

NodoD::NodoD(Telemento x)
{
 e = x;
 Sig = NULL;
 Ant = NULL;}
}

Telemento NodoD::OE()
{
 return e;
}

void NodoD::PE(Telemento x)
{
 e = x;
}

NodoD* NodoD::OSig()
{
```

```

 return Sig;
}

void NodoD:: PSig(NodoD *p)
{
 Sig = p;
}

NodoD* NodoD::OAnt()
{
 return Ant;
}

void NodoD:: PAnt(NodoD *p)
{
 Ant = p;
}

```

**EJEMPLO 17.13.** Clase lista doblemente enlazada con constructor destrutor, y funciones miembro para poner y obtener el puntero al primer nodo de la lista.

```

class ListaD //clase lista doblemente enlazada
{
protected:
 NodoD *p; // puntero al primer nodo protegido
public:
 ListaD(){p = NULL;} //constructor
 NodoD *Op() { return p; } //Obtener el puntero
 void Pp(NodoD * p1) {p = p1;} //Poner el puntero
 ~ListaD();{} //destructor
};

```

### 17.4.1. 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 y situar en el nodo `nuevo` el elemento `e` que se va a insertar del nuevo nodo `nuevo`.
- Hacer que el campo enlace `Sig` del nuevo nodo `nuevo` apunte a la cabeza (primer nodo `p`) de la lista original, y que el campo enlace `Ant` del nodo cabeza `p` apunte al nuevo nodo `nuevo` si es que existe. En caso de que no existe no hacer nada.
- Hacer que cabeza (puntero de la lista `p`) apunte al nuevo nodo que se ha creado.

**EJEMPLO 17.14.** Función miembro de la clase `ListaD` que añade un nuevo elemento a la lista doble como primer elemento.

```

void ListaD::InsertaP(Telemento e)
{
 NodoD *nuevo = new Nodo(e);

```

```

nuevo->Psig(p);
if(p) // si existe p que apunte a un nodo
 p->Pant(nuevo);
p = 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 al final de ella. El algoritmo de la nueva operación insertar *requiere* las siguientes etapas:

- Buscar la posición donde hay que insertar el dato, dejando los punteros a `NodoD`: `ant` apuntando al nodo inmediatamente anterior (siempre existe), y `pos` al que debe ser el siguiente en caso de que exista (`NULL` en caso de que no exista).
- Asignar memoria al nuevo nodo apuntado por el puntero `nuevo`, y situar el nuevo elemento como atributo del nuevo nodo `nuevo`.
- Hacer que el atributo `Sig` del nuevo nodo `nuevo` apunte al nodo `pos` 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 atributo `Ant` del nodo siguiente `ptrnodo` al que ocupa la posición del nuevo nodo `nuevo` que es `pos`, tiene que apuntar a `nuevo` si es que existe. En caso de que no exista no hacer nada.
- Hacer que el atributo `Sig` del puntero `ant` apunte al nuevo nodo `nuevo`. El atributo `Ant` del nuevo nodo `nuevo` ponerlo apuntando a `ant`.

**EJEMPLO 17.15.** Segmento de código de una función miembro de la clase `ListaD` que añade un nuevo elemento a la lista doble en una posición que no es la primera.

```

void ListaD::InsertaListadoble(Telemento e)
{
 NodoD *nuevo, *ant=NULL, *pos=p;
 // búsqueda de la posición donde colocar el dato e
 // se sabe que no es el primero de la lista doble

 aux = new NodoD(x);
 if (!pos) //último y no primero
 {
 ant->PSig(nuevo);
 nuevo->PAnt(ant);
 }
 else //no primero y no último
 {
 nuevo->PSig(pos);
 nuevo->PAnt(ant);
 ant->PSig(nuevo);
 pos->PAnt(nuevo);
 }
}

```

#### 17.4.2. 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 atributo `Sig` del nodo anterior (`ant`) tiene que apuntar al atributo `Sig` del nodo a eliminar, `pos` (esto en el caso de no ser el nodo primero de la lista). En caso de que sea el primero de la lista el atributo `p` de la lista debe apuntar al atributo `Sig` del nodo a eliminar `pos`.

- El atributo Ant del nodo siguiente a borrar tiene que apuntar al atributo Ant del nodo a eliminar, esto es 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 pos.

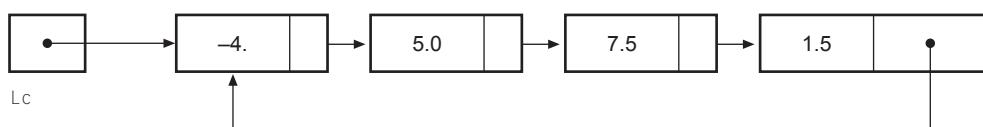
**EJEMPLO 17.16.** Segmento de código de una función miembro de la clase *ListaD* que borra un elemento en una lista doblemente enlazada.

```
void ListaD::Borrarelemento(Telemento x)
{
 NodoD *ant = NULL, *pos = p;

 // búsqueda del nodo a borrar omitida
 // NodoD a borrar en pos
if(!ant && ! pos->OSig()) // es el primero y último
{
 p = NULL; //la lista doble se queda vacía
}
else if (!ant) // primero y no último
{
 pos->OSig()->PAnt(NULL);
 p = pos->OSig();
 pos->PSig(NULL);
}
else if (!pos->OSig()) //ultimo y no primero
{
 ant->PSig(NULL);
 pos->PAnt(NULL);
}
else
{
 ant->PSig(pos->OSig());
 pos->OSig()->PAnt(ant);
 pos->PAnt(NULL);
 pos->PSig(NULL);
}
delete pos;
}
```

## 17.5. Listas circulares

En las listas lineales simples o en las dobles siempre hay un primer nodo y un último nodo que tiene el atributo de enlace a nulo (NULL). 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.



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 elemento en el atributo `e`.
- Si la lista está vacía enlazar el atributo `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 atributo `Sig` de nuevo nodo `nuevo` con el atributo `Sig` del nodo anterior `ant`. Enlazar el atributo `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 17.17.** Segmento de código de una función miembro de la clase `ListaS` que inserta un elemento en una lista circular simplemente enlazada.

Para implementar una lista circular simplemente enlazada, se puede usar la clase `ListaS`, y la clase `Nodo` de las listas simplemente enlazadas. El código que se presenta corresponde a una función miembro que añade a una lista circular simplemente enlazada un elemento `e`.

```
void ListaS::Insertar(Telemento e)
{
 Nodo *ant = NULL, nuevo;
 // Se busca dónde colocar el elemento x, dejando en ant la posición del
 // nodo inmediatamente anterior si es que existe.

 nuevo = new NodoD(e);
 if (p == NULL) //la lista está vacía y sólo contiene el nuevo elemento
 {
 nuevo->PSig(nuevo);
 p = nuevo;
 }
 else // lista no vacía, no se inserta como primer elemento.
 {
 nuevo->PSig (ant);
 ant->PSig(nuevo);
 }
}
```

#### Eliminación de un elemento en una lista circular simplemente enlazada

El algoritmo para eliminar un nodo de una lista circular es el siguiente:

- Buscar el nodo `ptrnodo` que contiene el dato quedándose con un puntero al anterior `ant`.
- Se enlaza el atributo `Sig` el nodo anterior `ant` con el atributo siguiente `Sig` del nodo a borrar. Si la lista contenía un solo nodo se pone a `NULL` el atributo `p` de la lista.
- En caso de que el nodo a eliminar sea el referenciado por el puntero de acceso a la lista, `p`, y contenga más de un nodo se modifica `p` para que tenga la el atributo `Sig` de `p` (si la lista se quedara vacía hacer que `p` tome el valor `NULL`).
- Por último, se libera la memoria ocupada por el nodo.

**EJEMPLO 17.18.** Segmento de código de una función miembro de la clase `ListaS` que borra un elemento en una lista circular simplemente enlazada.

```
void ListaS::Borrar(Telemento e)
{
 Nodo *ant = NULL, ptrnodo = p;
```

```

// búsqueda del elemento a borrar con éxito que es omitida

ant->Psig(ptrnodo->OS());
if (p == p->OSig()) // la lista se queda vacía
 p = NULL;
else if (ptrnodo == p) // si es el primero mover p
 p = ant->OSig();
}

```

## EJERCICIOS

- 17.1.** Escribir una función miembro de una clase lista simplemente enlazada que devuelva “verdadero” si está vacía y falso “falso” en otro caso; y otra función miembro que cree una lista vacía.
- 17.2.** Escribir una función miembro que devuelva el número de nodos de una lista simplemente enlazada.
- 17.3.** Escribir una función miembro de una clase lista simplemente enlazada que borre el primer nodo.
- 17.4.** Escribir una función miembro de una clase lista simplemente enlazada que muestre los elementos en el orden que se encuentran.
- 17.5.** Escribir una función miembro de una clase lista simplemente enlazada que devuelva el primer elemento de la clase lista si es que lo tiene.
- 17.6.** Escribir una función miembro de una clase lista doblemente enlazada que añada un elemento como primer nodo de la lista.
- 17.7.** Escribir una función miembro de una clase lista doblemente enlazada que devuelva el primer elemento de la lista.
- 17.8.** Escribir una función miembro de una clase lista doblemente enlazada que elimine el primer elemento de la lista si es que lo tiene.
- 17.9.** Escribir una función miembro de la clase lista doblemente enlazada que decida si está vacía.

## PROBLEMAS

- 17.1.** Escribir el código de un constructor de copia de una clase lista simplemente enlazada.
- 17.2.** Escribir el código de un destructor que elimine todos los nodos de un objeto de la clase lista simplemente enlazada.
- 17.3.** Escribir una función miembro de la clase lista simplemente enlazada, que inserte en una lista enlazada ordenada crecientemente, un elemento  $x$ .
- 17.4.** Escribir una función miembro de una lista simplemente enlazada, que elimine todas las apariciones de un elemento  $x$  que reciba como parámetro.
- 17.5.** Escriba un método de la clase lista simplemente enlazada que elimine un elemento  $x$  que reciba como parámetro de una lista enlazada ordenada.

- 17.6.** Escribir una función miembro de una clase lista simplemente enlazada que ordene la lista moviendo solamente punteros.
- 17.7.** Escribir una función que no sea miembro de la clase lista simplemente enlazada, que reciba como parámetro dos objetos de tipo lista que contengan dos listas ordenadas crecientemente, y las mezcle en otra lista que reciba como parámetro.
- 17.8.** Escribir el código de un constructor de copia de una clase lista doblemente enlazada.
- 17.9.** Escribir una función miembro de una clase lista doblemente enlazada que inserte un elemento  $x$  en la lista y que quede ordenada.
- 17.10.** Escribir una función miembro de una clase lista doblemente enlazada que borre un elemento  $x$  en la lista.
- 17.11.** Escriba una función que no sea miembro de la clase lista doblemente enlazada que copie una lista doble en otra.
- 17.12.** Escribir una clase lista simplemente enlazada circular que permita, decidir si la lista está vacía, ver cuál es el primero de la lista, añadir un elemento como primero de la lista, borrar el primer elemento de la lista y borrar la primera aparición de un elemento  $x$  de la lista.

## SOLUCIÓN DE LOS EJERCICIOS<sup>1</sup>

- 17.1.** Se supone declarada la clase `ListaS`, y la clase `Nodo` de los Ejemplos 17.3 y 17.4. En este caso la codificación de las funciones miembro, resulta sencilla y es la siguiente:

```
void ListaS::VaciaL()
{
 p = NULL;
}

bool ListaS::EsVaciaL()
{
 return !p;
}
```

- 17.2.** Para decidir el número de nodos de una lista, basta con recorrerla, y contarlos en un contador. Una codificación es la siguiente:

```
int ListaS::CuantosL()
{
 Nodo *aux = p;
 int c = 0;

 while (aux) // mientras queden datos en la lista.
 {
 c++;
 }
}
```

<sup>1</sup> En todos los ejercicios sobre listas simplemente enlazadas, se emplean la clases `Nodo` `ListaS`, implementadas en los Ejemplo 17.3 y 17.4 a la que se van añadiendo nuevas funciones miembros. En los ejercicios de listas doblemente enlazadas se emplean las clases `NodoD` y `ListaD` implementadas en los Ejemplos 17.11 y 17.12.

```

 aux = aux->OSig();
 }
 return c;
}

```

- 17.3.** En esta función miembro hay que puentear el primer nodo de la lista, y posteriormente borrarlo.

```

void ListaS::BorraPL()
{
 Nodo *paux;

 if(p) // si tiene datos, borrar el primero
 {
 paux = p; // nodo a borrar
 p = p->OSig(); // puentejar el nodo a borrar
 paux->PSig(NULL); // poner el siguiente a NULL. No necesario hacerlo
 delete paux;
 }
 else; // error la lista esta vacía y no se puede borrar.
}

```

- 17.4.** Se recorre los elementos de la lista con un puntero o *aux* a la clase *Nodo*, inicializado al primer elemento de la lista *p*, y se muestran sus contenidos.

```

void ListaS::MostrarL()
{
 Nodo *aux = p;

 while (aux)
 {
 cout << aux->OE()<< endl; // mostrar el dato
 aux = aux->OSig(); // avanzar en la lista
 }
}

```

- 17.5.** Si el puntero al primer nodo no es *NULL* entonces se retorna el elemento, y en otro caso no se hace nada.

```

Telemento ListaS::PrimeroPL()
{
 if (p)
 return p->OE();
 else ; //error

}

```

- 17.6.** Se crea un nuevo nodo de tipo *NodoD*, y se pone el atributo *Sig* a *p*. Si la lista no está vacía se pone el atributo *Ant* del nodo apuntado por *p* a *aux*. Siempre el primer nodo de la lista es el añadido.

```

void ListaD::AnadePLD(Telemento e)
{
 NodoD *aux;

```

```
aux = new NodoD(e);
if(p)
{
 aux->PSig(p);
 p->PAnt(aux);
}
p = aux;
}
```

- 17.7. Si la lista no está vacía el primer elemento es el que se extrae llamando a la función miembro `OE` de la clase `NodoD`.

```
Telemento ListaD::PrimerD()
{
 if (p)
 return p->OE();
 else // error
}
```

- 17.8. Si hay datos en la lista, se avanza el atributo que apunta al primer elemento de la lista doble , y se borra. Si la lista tiene más de un dato, hay que hacer que el atributo `Ant` del nodo apuntado por `p` apunte a `NULL`.

```
void ListaD::BorraPLD()
{
 NodoD *paux;

 if(p)
 {
 paux = p;
 p = p->OSig();
 if (p)
 p->PAnt(NULL);
 delete paux;
 }
 else; // error
}
```

- 17.9. Una codificación es la siguiente:

```
bool ListaD::EsVaciaLD()
{
 return !p;
}
```

## SOLUCIÓN DE LOS PROBLEMAS<sup>2</sup>

- 17.1.** El constructor de copia se encarga de copiar la lista enlazada que se recibe como parámetro por valor en cualquier función que use un objeto de la clase `ListaS`. La codificación que se presenta, añade un nodo ficticio como primer elemento de la lista enlazada que será la copia (usa la función miembro definida en el Ejemplo 17.5), para, posteriormente, ir recorriendo y copiando los elementos de la lista que se recibe como parámetro `p2`. Una vez terminada la copia, usa la función `Borrar el primero de la lista` codificada en el Ejercicio resuelto 17.3 para eliminar el nodo ficticio que se añadió.

**La codificación de este problema se encuentra en la página Web del libro.**

- 17.2.** El destructor va recorriendo todos los nodos de la lista y eliminándolos usando un puntero a `Nodo`.

**La codificación de este problema se encuentra en la página Web del libro.**

- 17.3.** La función miembro `AnadeOI` recibe como parámetro un elemento `x` 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 dónde colocar el dato. Una vez encontrado el sitio se realiza la inserción de acuerdo con el algoritmo.

**La codificación de este problema se encuentra en la página Web del libro.**

- 17.4.** Para borrar todas las apariciones del elemento `x` en la lista, hay que recorrer la lista hasta el final. Este recorrido se realiza con un puntero a `Nodo` `pos`, inicializado al primer nodo de la lista que es apuntado por el atributo `p`, y se itera mientras queden datos en la lista (`while (pos)`); y cada vez que se encuentra el elemento se elimina, teniendo en cuenta que sea o no el primer elemento de la lista. Una vez borrado, como hay que seguir buscando nuevas apariciones para eliminarlas, es necesario, poner el puntero que recorre la lista enlazada `pos` en el nodo siguiente al borrado.

**La codificación de este problema se encuentra en la página Web del libro.**

- 17.5.** La función miembro `BorrarOI` se encarga de buscar la primera aparición de dato `x` 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 `pos` y con el puntero `ant` (anterior). Posteriormente, realiza el borrado teniendo en cuenta que sea el primer o de la lista o que no lo sea.

**La codificación de este problema se encuentra en la página Web del libro.**

- 17.6.** La función miembro `ordenar`, realiza la ordenación de la lista con la ayuda de cuatro punteros:

- `pOrdenado` es un puntero que apunta a un nodo de la lista de tal manera que si, por ejemplo, hay 3 nodos en la lista anteriores a él esos tres nodos están ordenados y además sus contenidos son los más pequeños de toda la lista. Este puntero comienza apuntando al primero de la lista que se encuentra en el atributo `p`, y en cada iteración, avanza una posición en la lista, por lo que cuando llegue al final de la lista, ésta estará ordenada.
- `pMenor` es un puntero que recorre los elementos de la lista de uno en uno y comenzando por el nodo siguiente al apuntado por `pOrdenado`. Si el contenido apuntado por `pMenor` es más pequeño que el apuntado por `pOrdenado`, entonces moviendo punteros, se elimina de la posición que ocupa en la lista, y se coloca justo delante del nodo apuntado por `pOrdenado`, reubicando el puntero `pOrdenado` que ahora apunta `pMenor`, y el puntero `pMenor` que apunta al siguiente de donde estaba previamente.
- `pAntOrdenado`, apunta al nodo inmediatamente anterior a `pOrdenado`, para poder realizar inserciones.
- `AntpMenor`, apunta al nodo inmediatamente anterior a `pMenor`, para poder eliminarlo de la lista.

**La codificación de este problema se encuentra en la página Web del libro.**

<sup>2</sup> En todos los problemas sobre listas simplemente enlazadas, se emplean la clases `Nodo` y `ListaS`, implementadas en los Ejemplos 17.3 y 17.4 a la que se van añadiendo nuevas funciones miembros. En los problemas de listas doblemente enlazadas se emplean las clases `NodoD` y `ListaD` implementadas en los Ejemplos 17.12 y 17.13.

17.7. La función *mezclar* recibe como parámetros dos objetos *l1* y *l2* de tipo *ListaS*, ordenados crecientemente, y los mezcla ordenados en el objeto *l3*. La mezcla se realiza de la siguiente forma:

- Se inicializan los punteros a *Nodo*, *aux1* y *aux2* a los dos primeros nodos de las listas *l1* y *l2*, para ir avanzando con dos punteros *aux1* y *aux2* por las listas *l1* y *l2*. Un puntero a *Nodo* *aux3* apuntará siempre al último nodo añadido a la lista mezcla de las dos *l3*.
- Un primer bucle *while* avanzará o bien por *aux1* o bien por *aux2* insertando en la lista *mezcla l3*, dependiendo de que el dato más pequeño esté en *aux1* o en *aux2*, hasta que una de las dos listas se terminen.
- Dos bucles *while* posteriores se encargan de terminar de añadir a la lista *mezcla l3* los elementos que queden o bien de *l1* o bien de *l2*.
- Hay que tener en cuenta que la inserción de un nodo en la lista *l3*, puede ser al comienzo de la lista o bien al final.

**La codificación de este problema se encuentra en la página Web del libro.**

17.8. El constructor de copia se encarga de copiar la lista doblemente que se recibe como parámetro por valor en cualquier función que use un objeto de la clase *ListaD*. La codificación que se presenta, es muy similar a la del constructor de copia de *ListaS*. Sólo se diferencia en que usa la clase *NodoD*, y que, además, trata el atributo *Ant* de cada nodo de la lista doblemente enlazada. Al igual que su semejante en listas, añade un nodo ficticio como primer elemento de la lista doblemente enlazada que será la copia (usa la función *miembr* o definida en el Ejercicio resuelto 17.6), para posteriormente ir recorriendo y copiando los elementos de la lista que recibe como parámetro *p2*. Una vez terminada la copia, usa la función *Borrar el primero de la lista* codificada en el Ejercicio resuelto 17.8 para eliminar el nodo ficticio que se añadió.

**La codificación de este problema se encuentra en la página Web del libro.**

17.9. La función *miembro* de la clase *ListaD* *InsertarOI* inserta el dato *x* en una lista enlazada ordenada crecientemente, como primer elemento, en caso de igualdad de elementos almacenados. La inserción, comienza con una búsqueda, que termina, cuando la lista se ha terminado, o cuando se ha encontrado la posición. En los puntero a *NodoD* *ant* y *pos* se tienen respectivamente la dirección del nodo inmediatamente anterior y el inmediatamente posterior en caso de que existan, o *NULL* en otro caso. Una vez situados los dos punteros, y creado el nuevo nodo *aux* que almacena el dato *x*, la inserción contempla los cuatro casos:

- La lista está vacía. Basta con asignar a *p* el puntero *aux*.
- Se inserta como primer elemento y no último. Hay que poner: el siguiente de *aux* en *p*; el anterior de *p* en *aux*; y *p* a *aux*.
- Se inserta como último y no primero. Hay que poner: el siguiente de *ant* en *aux*; y el anterior de *aux* en *ant*.
- Se inserta en el centro de la lista. Hay que poner: el siguiente de *aux* en *pos*; el anterior de *aux* en *ant*; el siguiente de *ant* en *aux*; y el anterior de *pos* en *aux*.

**La codificación de este problema se encuentra en la página Web del libro.**

17.10. El borrado de *x*, va precedido de la búsqueda en una lista ordenada crecientemente. Una vez terminada la búsqueda y colocado de la forma adecuada los punteros a *NodoD*, *ant* y *pos*, se decide si la parada de la búsqueda da como resultado que hay que borrar el elemento. Esto ocurre cuando *enc* vale verdadero y en *pos* se encuentra almacenado *x*. En el borrado del nodo apuntado por *pos*, se distinguen cuatro casos para el movimiento de punteros:

- Es el primero y último de la lista: poner *p* a *NULL*, ya que la lista se queda vacía.
- Es el primero y no último: poner *p* al siguiente de *pos*; y el anterior de *p* a *NULL*.
- Es el último y no primero: poner el siguiente de *ant* a *NULL*.
- No primero y no último: poner el siguiente de *ant* al siguiente de *pos*, y poner el anterior del siguiente de *pos* a *ant*.

Una vez que se han realizado los enlaces, se procede a la eliminación del nodo apuntado por *pos*.

**La codificación de este problema se encuentra en la página Web del libro.**

- 17.11.** Para copiar la lista doblemente enlazada apuntada por `ld`, en la lista doblemente enlazada `ld1`, se usan las funciones miembro de la clase `ListaD`, `PrimerD`, `AnadePLD`, `EsVaciaLD`, `BorraPLD`, codificadas en los Ejercicios resueltos: 17.7, 17.8, 17.9, 17.10. Se realiza de la siguiente forma, se copia en primer lugar la lista doblemente enlazada `ld` en una `ld2` auxiliar como si fuera una pila. Es decir, los elementos quedan justo en el orden inverso al que se encontraban en la lista original. Posteriormente, se vuelve a copiar, de la misma forma, la lista doblemente enlazada `ld2` en `ld1`, con lo que la lista queda en su orden natural.

La codificación de este problema se encuentra en la página Web del libro.

- 17.12.** La clase lista simplemente enlazada circular usa la clase `Nodo` definido en el Ejemplo 17.3. Las funciones miembro o: `Listac(); void VaciaL(); Listac (const Listac &p2); ~Listac(); Nodo * Op(); void Pp(Nodo *p1); y bool EsVaciaL();` coinciden con las codificadas en la clase `ListaS` en los ejercicios y problemas del presente capítulo, por lo que sólo se incluyen el código de las restantes funciones miembros pedidas.

La lista circular se implementa de tal manera que el atributo `p` de la lista, siempre apunta al último elemento de la lista, y el primero de la lista siempre se encuentra en el nodo apuntado por el siguiente de `p`, en caso de que lo tenga.

- De esta forma la codificación de la función miembro `Primeropl()`, que retorna el primero de la lista es tan sencilla como `return p->OSig()->OE();`.
- `AnadePL()`, añade un elemento `x` como primero de la lista, es decir, entre el nodo apuntado por el atributo `p`, y el apuntado por `p->OSig()`. En la codificación hay que tener en cuenta que la lista se encuentre vacía ya que en este caso el primer elemento se debe quedar apuntado por el propio atributo `p->OSig()` de la lista circular.
- La función miembro `BorraPL()` elimina el nodo apuntado por `p->OSig()`, que es el primero de la lista. En la codificación se tiene en cuenta que la lista se quede vacía al eliminarlo (la lista tenía antes del borrado un solo nodo).
- Para visualizar el contenido de la lista circular, hay que tener en cuenta que los nodos están en un ciclo, por lo que el bucle que controla la salida de datos debe parar cuando “se haya dado la vuelta a la lista circular”. Esta condición se traduce en el control del bucle `do while` en la condición (`aux != p->OSig()`). Hay que observar que en el cuerpo del bucle, `aux` es un puntero que apunta al primer elemento que debe ser escrito, por tanto, hay que mostralo, y avanzar el puntero `aux` al siguiente de la lista.
- El borrado de la primera aparición de un elemento es realizado por la función miembro `BorrarL`. El código de la función busca la primera aparición del elemento, y una vez encontrado, lo elimina, teniendo en cuenta que la lista se puede quedar vacía, o que el elemento que se elimine sea el último de la lista, que es el apuntado por el atributo `p`.
- Además de las macros de las funciones de generación de números aleatorios se presenta una función no miembro de la clase `Listac`, que genera aleatoriamente una lista, y un programa principal que llama a la función de generación aleatoria de los elementos de la lista, y la función miembro que visualiza la lista circular.

La codificación de este problema se encuentra en la página Web del libro.

## EJERCICIOS PROPUESTOS

- 17.1.** 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.
- 17.2.** Se tiene una lista de simple enlace, el campo dato es un registro (estructura) con los campos 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.
- 17.3.** Escribir un algoritmo que lea una lista de enteros del teclado, cree una lista enlazada con ellos e imprima el resultado.
- 17.4.** Escribir un algoritmo que acepte una lista enlazada, la recorra y devuelva el dato del nodo con el valor menor.
- 17.5.** Escribir un algoritmo que acepte una lista enlazada, la recorra y devuelva el dato del nodo con el valor mayor.
- 17.6.** Escribir una función que genere una lista circular con los datos 1, 2, 3, ..., 20.

## PROBLEMAS PROPUESTOS

- 17.1.** Escribir una función miembro de la clase *Lista doblemente enlazada* que cree una lista doblemente enlazada de palabras introducidas por teclado. La función debe tener un argumento, un puntero al *Nodo* de la lista doble *Pd* en el que se devuelva la dirección del nodo que está en la posición intermedia.
- 17.2.** Escribir un programa que cree un array de listas enlazadas.
- 17.3.** Una lista circular de cadenas está ordenada alfabéticamente. El atributo *p* del objeto *Lista circular* tiene la dirección del nodo alfabéticamente mayor, que a su vez apunta al nodo alfabéticamente menor. Escribir una función para añadir una nueva palabra, en el orden que le corresponda, a la lista.
- 17.4.** Se dispone de una lista enlazada con la que los datos de cada elemento contiene un entero. Escribir una función que decida si la lista está ordenada.
- 17.5.** 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.
- 17.6.** Escribir un programa que lea una lista de estudiantes de un archivo y cree una lista enlazada. Cada entrada de la lista enlazada ha de tener el nombre del estudiante, un puntero al siguiente estudiante y un puntero a una lista enlazada de calificaciones.
- 17.7.** 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 atributos el coeficiente del término y el exponente. Por ejemplo, el polinomio  $3x^4 - 4x^2 + 11$  se representa en la siguiente figura:
- 
- 17.8.** 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, ..., 5.0.
- 17.9.** 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 tener la dirección del último término del polinomio, el cuál apuntará al primer término.
- 17.10.** Escribir una función que tome una lista enlazada de enteros e invierta el orden de sus nodos.
- 17.11.** 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.
- 17.12.** 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, implementados con listas simplemente enlazadas. Posteriormente, encontrar las palabras comunes y mostrarlas por pantalla.
- 17.13.** Se dispone de una lista doblemente enlazada ordenada crecientemente con claves repetidas. Realizar una función de inserción de una clave en la lista de forma tal que si la clave ya se encuentra en la lista se inserte al final de todas las que tienen la misma clave.
- 17.14.** 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.
- 17.15.** 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.
- 17.16.** 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.
- 17.17.** Un vector disperso es aquel que tiene muchos elemen-

tos 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.

**17.18.** Escribir una función que permita insertar un elemento en una posición determinada de una lista doblemente enlazada.

**17.19.** Escriba una función que tenga como argumento una lista circular de números enteros. La función debe de volver el dato del nodo con mayor valor.

## CAPÍTULO 18

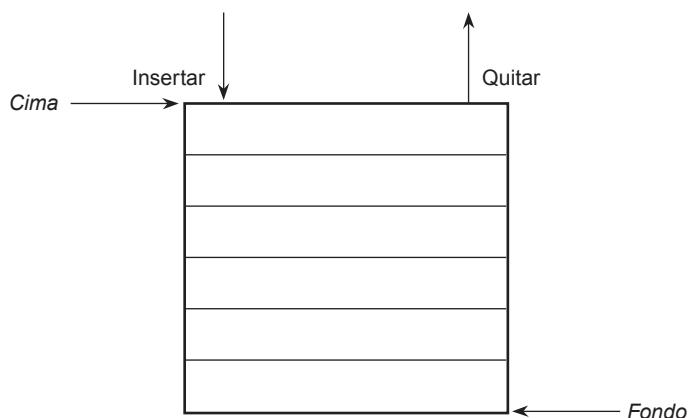
# Pilas y colas

## Introducción

En este capítulo se estudian en detalle las estructuras de datos pilas y colas que son, probablemente, las utilizadas más frecuentemente en los programas más usuales. 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.

### 18.1. 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 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.



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 cabe más elementos en la pila). Otro método de diseño y construcción de pilas puede realizarse mediante plantillas con la biblioteca **STL**. El mejor sistema para definir una pila es con una clase (*class*) o bien con una plantilla (*template*).

### Especificación de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes:

|               |                                                    |
|---------------|----------------------------------------------------|
| AnadeP (push) | Insertar un dato en la pila.                       |
| BorrarP (pop) | Sacar (quitar) un dato de la pila.                 |
| EsVaciaP      | Comprobar si la pila no tiene elementos.           |
| VaciaP        | Crea la Pila vacía.                                |
| EstallenaP    | Comprobar si la pila está llena de elementos.      |
| Primerop      | Extrae el primer elemento de la pila sin borrarlo. |

### La clase pila implementado con arrays (arrays)

En C++ para definir una pila con arrays se utiliza una clase. Los atributos de la clase *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 *MaxTamaPila* elementos, el índice o puntero de la pila, estarán comprendidos en el rango 0 a *MaxTamaPila*-1 elementos, de modo que *en una pila llena* el puntero de la pila apunta a *MaxTamaPila*-1 y *en una pila vacía* el puntero de la pila apunta a -1, ya que 0, teóricamente, es índice del primer elemento.

#### EJEMPLO 18.1. Clase pila implementada con array.

Se define una constante *MaxTamaPila* cuyo valor es 100 que será el valor máximo de elementos que podrá contener la pila. Se define en un *typedef* el tipo de datos que almacenará *Pila*, que en este caso serán enteros. *Pila* es una clase cuyos atributos son la *cima* que apunta siempre al último elemento añadido a la pila y un array *A* cuyos índices variarán entre 0 y *MaxTamaPila*-1. Todas la funciones miembro de la clase, son públicas, excepto *EstallenaP* que es privada.

- *VaciaP*. Crea la pila vacía poniendo la cima en el valor -1.
- *Pila*. Es el constructor de la *Pila*. Coincide con *VaciaP*.
- *EsvaciaP*. Decide si la pila vacía. Ocurre cuando su *cima* valga -1.
- *EstallenaP*. En la implementación de una pila con array, hay que tenerla declarada como privada 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. Primeramente 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`.

```
#include <cstdlib>
#include <iostream>
using namespace std;

#define MaxTamaPila 100

typedef int TipoDato;
class Pila
{
protected:
 TipoDato A[MaxTamaPila];
 int cima;

public:
 Pila(){ cima = -1; } //constructor
 ~Pila(){} //destructor
 void VaciaP();
 void AnadeP(TipoDato elemento);
 void BorrarP();
 TipoDato PrimeroP();
 bool EsVaciaP();
 void Push(TipoDato elemento);
 TipoDato Pop();
private:
 bool EstallenaP();
};

void Pila:: VaciaP()
{
 cima = -1;
}

void Pila::AnadeP(TipoDato elemento)
{
 if (EstallenaP())
 {
 cout << " Desbordamiento pila";
 exit (1);
 }
 cima++;
 A[cima] = elemento;
}

void Pila::Push (TipoDato elemento)
{
 AnadeP (elemento);
}
```

```
 TipoDato Pila::Pop()
 {
 TipoDato Aux;

 if (EsVaciaP())
 {
 cout << "Se intenta sacar un elemento en pila vacía";
 exit (1);
 }
 Aux = A[cima];
 cima--;
 return Aux;
 }

 TipoDato Pila::PrimeroP()
 {
 if (EsVaciaP())
 {
 cout << "Se intenta sacar un elemento en pila vacía";
 exit (1);
 }
 return A[cima];
 }

 void Pila::BorrarP()
 {
 if (EsVaciaP())
 {
 cout << "Se intenta sacar un elemento en pila vacía";
 exit (1);
 }
 cima--;
 }

 bool Pila::EsVaciaP()
 {
 return cima == -1;
 }

 bool Pila::EstallenaP()
 {
 return cima == MaxTamaPila-1;
 }

 int main(int argc, char *argv[])
 {
 Pila P;

 P.AnadeP(5);
 P.AnadeP(6);
 cout << P.Pop() << endl;;
 cout << P.Pop() << endl;
 system("PAUSE");
 return EXIT_SUCCESS;
 }
```

### La clase pila implementada con punteros

Para implementar una pila con punteros basta con usar una lista simplemente enlazada, con lo que la pila estará vacía si la lista apunta a `NULL`. La pila teóricamente nunca estará llena. La pila se declara como una clase que tiene un atributo protegido que es un puntero a nodo, este puntero señala el extremo de la lista enlazada por el que se efectúan las operaciones. Siempre que se quiera poner un elemento se hace por el mismo extremo que se extrae.

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 quiere 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).

#### EJEMPLO 18.2. Clase Pila implementada con punteros.

Se define en primer lugar la clase `Nodo`, ya implementada en el Ejercicio resuelto 17.3 y utilizada en las listas simplemente enlazadas. La clase `Pila` tiene por atributo protegido un puntero a la clase `Nodo`. Las funciones miembros que se implementan de la clase `Pila` son:

- Constructor `Pila`. Crea la pila vacía poniendo el atributo `p` a `NULL`.
- Constructor de copia `Pila` utilizado para la transmisión de parámetros por valor de una pila a una función.
- Destructor `~Pila`, cuyo código elimina todos los nodos de la lista.
- `VaciaP`. Crea la pila vacía poniendo el atributo `p` a `NULL`.
- `EsvaciaP`. Decide si la pila está vacía. Esto ocurre cuando el atributo `p` valga `NULL`.
- `AnadeP`. Añade un elemento a la pila. Para hacerlo es preciso 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. Primeramente 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`.

```
#include <cstdlib>
#include <iostream>
using namespace std;

typedef int Telemento;

class Nodo {
protected:
 Telemento e;
 Nodo *Sig;
public:
 Nodo(){}
 ~Nodo(){}
 Nodo (Telemento x){ e = x; Sig = NULL;} // constructor
 Telemento OE(){ return e;} // Obtener elemento
 void PE(Telemento x){ e = x;} // poner elemento
 Nodo * OSig(){ return Sig;} // Obtener siguiente
 void PSig(Nodo *p){ Sig = p;} // poner siguiente
};

class Pila
```

```
{
protected:
 Nodo *p;

public:
 Pila(){p = NULL;}
 void VaciaP(){ p = NULL;}
 bool EsvaciaP(){ return !p;}
 Telemento PrimeroP(){ if (p) return p->OE();}
 void AnadeP(Telemento e);
 void Push(Telemento e){ AnadeP(e);}
 void BorrarP();
 Pila::Pila (const Pila &p2);
 ~Pila();
 Telemento Pop();
};

void Pila::AnadeP(Telemento e)
{
 Nodo *aux;

 aux = new Nodo(e);
 if(p)
 aux->PSig(p);
 p = aux;
}

void Pila::BorrarP()
{
 Nodo *paux;

 if(p)
 {
 paux = p;
 p = p->OSig();
 delete paux;
 }
 else; // error
}

Pila::Pila (const Pila &p2)
{
 Nodo* a = p2.p, *af, *aux;

 p = NULL; // Se añade Nodo Cabecera
 AnadeP(-1);
 af = p;
 while (a != NULL)
 {
 aux = new Nodo(a->OE());
 af->PSig(aux);
 af = aux;
 a = a->OSig();
 }
}
```

```
BorrarP(); // Se borra nodo Cabecera
}

Pila:: ~Pila()
{
 Nodo *paux;

 while(p != 0)
 {
 paux = p;
 p = p->Osig();
 delete paux;
 }
}

Telemento Pila::Pop()
{
 Telemento e;

 e = PrimeroP();
 BorrarP();
 return e;
}

int main(int argc, char *argv[])
{
 Pila p;

 p.VaciaP();
 p.AnadeP(4);
 p.AnadeP(5);
 while (!p.EsvaciaP())
 {
 cout << p.PrimeroP() << endl;
 p.BorrarP();
 }
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

## 18.2. Concepto de cola

Una **cola** es una estructura de datos lineal donde 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.

### La clase cola implementada con arrays

La definición de una clase **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 *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 *TipoDatos*. 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*. Éstos son los punteros de cabecera y cola o final respectivamente. Las operaciones típicas de la cola son

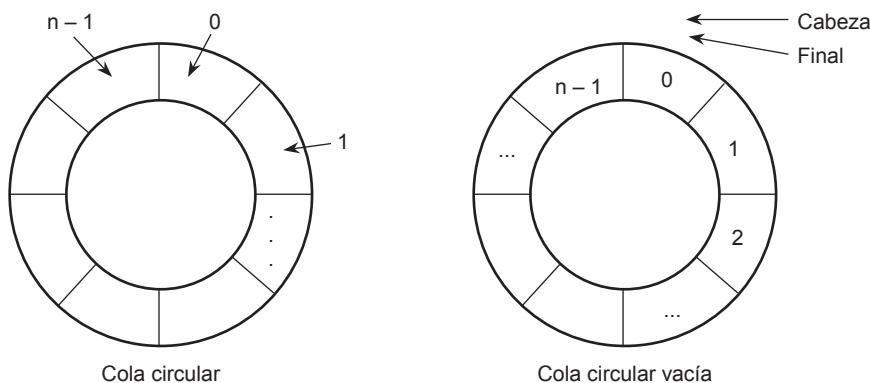
|                   |                                       |
|-------------------|---------------------------------------|
| <i>AnadeC</i>     | Añade un elemento a la cola.          |
| <i>BorrarC</i>    | Borra el primer elemento de la cola.  |
| <i>VaciaC</i>     | Deja la cola sin ningún elemento.     |
| <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. |

Las implementaciones clásicas de una cola son:

**Retroceso.** Consiste en mantener fijo a uno el valor de *frente*, realizando un desplazamiento de una posición para todas las componentes ocupadas cada vez que se efectúa una supresión.

**Reestructuración.** Cuando *final* llega al máximo de elementos, se desplazan las componentes ocupadas hacia atrás las posiciones necesarias para que el principio coincida con la primera posición del array .

**Mediante un array circular.** Un array circular es aquél en el que 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*:

$$\begin{array}{ll} \text{Mover final adelante} & = (\text{final}+1)\% \text{MaxTamC} & \text{Siguiente de final} \\ \text{Mover frente adelante} & = (\text{frente}+1) \% \text{MaxTamC} & \text{Siguiente de frente} \end{array}$$

### EJEMPLO 18.3. Implementación de la clase Cola con un array circular.

La clase *Cola* contiene los atributos protegidos *frente*, *final* y el array *A* declarado de una longitud máxima. Todas las funciones miembro serán públicas, excepto el método *Estallenac* que será privado de la clase. En la implementación de la clase *Cola* con un array circular hay que tener en cuenta que *frente* va a apuntar siempre a una posición

anterior donde se encuentra el primer elemento de la cola y `final` va a apuntar siempre a la posición donde se encuentra el último de la cola. Por 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`.

```
#include <cstdlib>
#include <iostream>
#define MaxTamC 100
using namespace std;

typedef int TipoDato;
class Cola
{
protected:
 int frente, final;
 TipoDato A[MaxTamC];

public:
 Cola(); //constructor
 ~Cola(); //destructor
 void VaciaC();
 void AnadeC(TipoDato e);
 void BorrarC();
 TipoDato PrimeroC();
 bool EsVaciaC();

private:
 bool EstallenaC();
};

Cola::Cola()
{
 frente = 0;
 final = 0;
}

Cola::~Cola(){}

void Cola::VaciaC()
{
 frente = 0;
 final = 0;
}

void Cola::AnadeC(TipoDato e)
{
 if (EstallenaC())
 {

```

```

 cout << "desbordamiento cola";
 exit (1);
 }
 final = (final + 1) % MaxTamC;
 A[final] = e;
}

TipoDato Cola::PrimeroC()
{
 if (EsVaciaC())
 {
 cout << "Elemento frente de una cola vacía";
 exit (1);
 }
 return (A[(frente+1) % MaxTamC]);
}

bool Cola::EsVaciaC()
{
 return (frente == final);
}

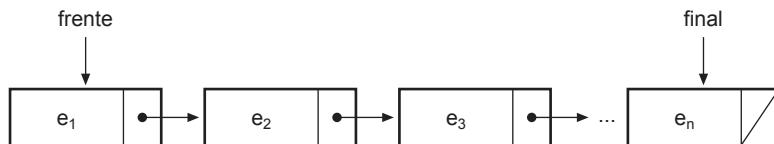
bool Cola::EstallenaC()
{
 return (frente == (final+1) % MaxTamC);
}

void Cola::BorrarC()
{
 if (EsVaciaC())
 {
 cout << "Eliminación de una cola vacía";
 exit (1);
 }
 frente = (frente + 1) % MaxTamC;
}

```

### La clase cola implementada con una lista enlazada

Cuando la cola se implementa utilizando variables dinámicas, la memoria utilizada se ajusta en todo momento al número de elementos de la cola, pero se consume algo de memoria extra para realizar el encadenamiento entre los elementos de la cola. Se utilizan dos punteros para acceder a la cola, `frente` y `final`, que son los extremos por donde salen los elementos y por donde se insertan respectivamente.



**EJEMPLO 18.4.** *Implementación de la clase Cola con una lista enlazada con frente y final.*

Se usa la clase `Nodo` implementada y explicada en el Ejemplo resuelto 17.3 y utilizada en las listas simplemente enlazadas. Esta clase `Nodo` es incluida en la implementación. La clase `Cola` tiene dos tipos atributos protegidos `Frente` y

`Final` que son punteros a la clase `Nodo` y que apuntarán al primer nodo de la `Cola` y al último nodo de la `Cola` respectivamente. Las funciones miembro de la clase son:

- `Cola`. Es el constructor por defecto que pone los atributos `Frente` y `Final` a `NULL`.
- `Cola (const Cola &p2)`. Se encarga de hacer una copia de la `Cola` `p2`; para ello crea la cola `p2` vacía, y posteriormente recorre todos los nodos de la cola `p2` y se los añade a la actual. Es el constructor de copia y se activa cada vez que hay una transmisión de parámetros por valor.
- `~Cola()`. Es el destructor de la `Cola`. Se encarga de mandar a la memoria disponible todos los nodos de la `Cola`.
- `VaciaC`. Crea una cola vacía, para lo cual basta con poner el `Frente` y el `Final` a `NULL`. Realiza el mismo trabajo que el constructor por defecto.
- `EsVaciaC`. Decide si una cola está vacía. Es decir si `Frente` y `Final` valen `NULL`.
- `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.

```
#include <cstdlib>
#include <iostream>
using namespace std;

typedef int Telemento;

class Nodo {
protected:
 Telemento e;
 Nodo *Sig;

public:
 Nodo(){}
 ~Nodo(){}
 Nodo (Telemento x){ e = x; Sig = NULL;} // constructor
 Telemento OE(){ return e;} // Obtener elemento
 void PE(Telemento x){ e = x;} // poner elemento
 Nodo * OSig(){ return Sig;} // Obtener siguiente
 void PSig(Nodo *p){ Sig = p;} // poner siguiente
};

class Cola
{
protected:
 Nodo *Frente, *Final;

public:
 Cola(); // constructor por defecto
 Cola (const Cola &p2); // para los de parámetros por valor
 ~Cola(); //destructor
 void VaciaC();
 void AnadeC(Telemento e);
 Telemento PrimeroC();
 void BorrarC();
 bool EsvaciaC();
};
```

```

Cola::Cola()
{
 Frente = NULL;
 Final = NULL;
}

Cola::Cola (const Cola &p2)
{
 Telemento e;
 Nodo* a = p2.Frente;
 Frente = NULL;
 Final = NULL;
 while (a != NULL)
 {
 AnadeC (a -> OE());
 a = a->OSig();
 }
}

Cola:: ~Cola()
{
 Nodo *paux;
 while(Frente != 0)
 {
 paux = Frente;
 Frente = Frente->OSig();
 delete paux;
 }
}

void Cola::VaciaC()
{
 Frente = NULL;
 Final = NULL;
}

void Cola::AnadeC(Telemento e)
{
 Nodo *aux;

 aux = new Nodo(e); // nuevo nodo
 if(Final) // cola no vacía se coloca después de Final
 Final->PSig(aux);
 Else // la cola está vacía es Frente y Final
 Frente = aux;
 Final = aux;
}

Telemento Cola::PrimeroC()
{
 if (Frente)
 return Frente->OE();
}

```

```

bool Cola::EsvaciaC()
{
 return !Frente;
}

void Cola::BorrarC()
{
 Nodo *paux;

 if(Frente)
 {
 paux = Frente; // nodo a borrar
 Frente = Frente->OSig();
 delete paux;
 }
 else; // error
 if (! Frente) // se queda vacía
 Final = NULL;
}

```

## EJERCICIOS

**18.1.** Escribir un programa que usando la clase *Pila*, lea datos de la entrada (*-I* fin de datos), los almacene en una pila y posteriormente visualice la pila.

**18.2.** ¿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;
P.VaciaP();
P.AnadeP(x);
P.BorrarP();
P.AnadeP(32);
y = P.PrimeroP();
P.BorrarP();
P.AnadeP(y);
do
{
 cout <<" " << P.PrimeroP()<< endl;
 P.BorrarP();
}
while (!P.EsvaciaP());

```

**18.3.** Codificar el destructor de la clase *Pila* implementada con punteros del Ejemplo 18.2, para que libere toda la memoria apuntada por la pila.

**18.4.** Escribir una función que no sea miembro de la clase *Pila*, que copie una pila en otra.

**18.5.** Escribir una función que no sea miembro de la clase *Pila*, que muestre el contenido de una pila que reciba como parámetro.

**18.6.** Considerar una cola de nombres representada por una array circular con 6 posiciones, y los elementos de la Cola: Mar, Sella, Centurión. Escribir los elementos de la cola y los valores de los nodos 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.

**18.7.** Escribir una función no miembro de la clase `Cola` que visualice los elementos de una cola.

**18.8.** Escribir una función que reciba una cola como parámetro, e informe del número de elementos que contiene la cola.

## PROBLEMAS

**18.1.** Implementar una clase `pila` con un array dinámico.

**18.2.** Implementar una clase `pila` con las funciones miembro “`pop`” y “`push`”. Ésta llena con su constructor y destructor, usando un array dinámico y tipos genéricos.

**18.3.** Escribir las funciones no miembro de la clase `Pila MayorPila` y `MenorPila`, que calculan el elemento mayor, menor de una pila de enteros.

**18.4.** Escribir la función no miembro de la clase `pila`, `MediaPila` que calcule la media de una pila de enteros.

**18.5.** Escribir una función no miembro de la clase `Pila`, que decida si dos pilas son iguales.

**18.6.** 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. Por ejemplo, `ab&ba` sí es de la forma indicada, pero no lo son `ab&ab` ni `o ab&xba`.

**18.7.** Implementar la clase `Cola` con una lista circular simplemente enlazada.

**18.8.** Implementar una función no miembro de la clase `Cola` de enteros que reciba una cola como parámetro y retorne el elemento mayor y el menor de una instancia de la clase `Cola`.

**18.9.** Escribir una función que reciba dos objetos de la clase `Cola` como parámetros y decida si son iguales.

**18.10.** Escribir una función que reciba como parámetro una instancia de la clase `Cola` y un elemento, y elimine todos los elementos de la instancia que sean mayores que el que recibe como parámetro.

**18.11.** Escribir 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” sí que lo es.

**18.12.** Escribir un programa para gestionar una cola genérica implementada con un array dinámico circular con control de excepciones.

## SOLUCIÓN DE LOS EJERCICIOS

- 18.1. Se usa la implementación de la clase `pila` realizada en los Ejemplos 18.1 o 18.2. Mediante un bucle `do while` se leen los datos y en otro bucle `while` se visualizan los resultados.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

- 18.2. 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

- 18.3. El destructor recorre todos los nodos de la pila liberando la memoria al tiempo que mueve el atributo que apunta al primer nodo.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

- 18.4. La función no miembro de la clase `Pila`, `Copiar` recibe la pila `p` como parámetro por valor, y retorna como resultado una copia de la pila en el parámetro por referencia `p1`. Se vuelca, en primer lugar, el contenido de la pila `p` en la pila auxiliar `p2`, para posteriormente hacer lo mismo con la pila `p2` sobre la pila `p1`.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

- 18.5. Mediante un bucle `while` se itera mientras queden datos en la pila, extrayendo el primer elemento, borrándolo y visualizándolo. Como el parámetro es por valor, la Pila `P` no se destruyen fuera de la función no miembro de la clase `Pila`.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

- 18.6. 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.

- 18.7. La función recibe como parámetro una cola y la presenta en pantalla. Para ello mediante un bucle `while` extrae elementos de la cola y los escribe. El código incluye un contador, para que cada vez que se hayan escrito 10 elementos saltar de línea, y favorecer de esta forma la visualización de los datos.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

- 18.8. Para resolver el problema, usa un contador que se inicializa a cero, y se incremente en una unidad, cada vez que un bucle `while` extrae un elemento de la cola.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

## SOLUCIÓN DE LOS PROBLEMAS

- 18.1. Una pila puede implementarse con un array dinámico, simplemente cambiando el atributo *A* del array, por un puntero al tipo de dato. Ahora bien, para que la pila funcione, el tamaño máximo de la pila debe ser definido al declararlo, por lo que el constructor de la *Pila*, debe hacer la reserva de memoria correspondiente. La clase *Pila*, se implementa como el Ejemplo 18.1, pero cambian solamente, el tipo del atributo *A*, y el constructor *Pila*, por lo que sólo se incluye la declaración de la clase y la implementación del constructor.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.2. La implementación con array dinámico es similar a la realizada en el problema anterior. Para permitir la generalidad de tipos, hay que añadir template. Se incluye además un programa principal como ejemplo de llamada.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.3. La función no miembro de la clase *Pila 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 y teniendo el mayor de todos. *MenorPila* calcula el elemento Menor. Se realiza de manera análoga a la función *MayorPila*.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.4. La función no miembro de la clase *pila MediaPila* 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 entre *Total* y *k*, convirtiendo previamente *Total* a float, para obligar a que el cociente sea real.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.5. Dos pilas son iguales si tienen el mismo número de elementos y además coinciden en el orden de colocación. Por tanto, basta con un bucle **mientras** controlado por la existencia de 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.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.6. 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 retorna el valor de son iguales las dos pilas. La función *SonIgualesPilas* se ha codificado en el Problema 18.5. Se codifican además la función *DaVueltaPila*, que recibe una pila como parámetro y la da la vuelta.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.7. Para comprobar el equilibrio de paréntesis y corchetes, hay que decidir si tienen el mismo número de abiertos que cerrados y además si aparecen en el orden correspondiente. Se usa una pila por la que pasan sólo los paréntesis abiertos y los corchetes abiertos en el orden en que aparecen. Cada vez que nos 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 tanto, si con un bucle *while* controlado por el fin de línea y por el valor verdadero de una variable *sw* de tipo lógico (previamente inicializada a verdadero), se lean 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.

**La codificación de este problema se encuentra en la página Web del libro.**

- 18.8. Una Cola es una clase que contiene un atributo que es un puntero al nodo de la lista simplemente enlazada denominado *Nodo*. Todas las funciones miembro tienen la misma estructura que en el Ejemplo 18.4. La clase *Cola* tiene un solo atribu-

to protegido  $C$  que es un puntero a la clase  $Nodo$ . La implementación se realiza de tal manera que si la  $Cola$  no está vacía el atributo  $C$  apunta al último nodo de la cola y el siguiente de  $C$  apunta siempre al primer elemento de la cola. Los métodos de la clase  $Cola$  son:

- $Cola$ . Es el constructor por defecto que pone el atributo  $C$   $NULL$ .
- $Cola$  ( $const Cola \&p2$ ). Se encarga de hacer una copia de la  $p2$  en la  $Cola$ , para ello crea la cola vacía, y posteriormente recorre todos los nodos de la cola  $p2$  y se los añade a  $p2$ .
- $\sim Cola()$ . Es el destructor de la  $Cola$ . Se encarga de enviar a la memoria disponible todos los nodos de la  $Cola$ .
- $VaciaC$ . Crea una cola vacía, para lo cual basta con poner  $C$  a  $NULL$ . Realiza el mismo trabajo que el constructor por defecto.
- $EsVaciaC$ . Decide si una cola está vacía. Es decir si  $C$  vale  $NULL$ .
- $PrimerC$ . Extrae el primer elemento de la cola que se encuentra en el atributo correspondiente del puntero  $C->OSig()$ . 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  $C$ . Si la cola está vacía antes de añadir, hay que hacer que el nuevo nodo se apunte así mismo y  $C$  apunte al nuevo nodo. En caso de que la cola no esté vacía, el nuevo nodo debe añadirse, entre los punteros siguiente de  $C$  y el siguiente del siguiente de  $C$ . En todo caso siempre  $C$  debe apuntar 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 eliminando el nodo apuntado por el siguiente de  $C$  y estableciendo los enlaces correspondientes. Siempre hay que considerar que la cola se queda vacía.

**La codificación de este problema se encuentra en la página Web del libro.**

**18.9.** Si se usan los métodos de gestión de colas, lo único que hay que hacer es inicializar  $Mayor$  y  $menor$  al primer elemento de la cola, y mediante un bucle voraz controlado por se vacía la cola ir actualizando las variables  $Mayor$  y  $menor$ .

**La codificación de este problema se encuentra en la página Web del libro.**

**18.10.** Dos colas  $C1$  y  $C2$  son iguales, si tienen el mismo número de elementos y además están colocados en el mismo orden. Por lo que la codificación es tan sencilla como ir extrayendo elementos de las dos colas y comprobando que siguen siendo iguales. Se implementa con un bucle voraz que inicializa una variable  $bool sw$  a  $true$  e itera mientras queden datos en las dos colas y el  $sw$  siga manteniéndose a  $true$ . Una vez que se ha terminado de extraer los datos de las colas, para que sean iguales deben estar las dos colas vacías, y además la variable  $sw$  estar a  $true$ .

**La codificación de este problema se encuentra en la página Web del libro.**

**18.11.** La función recibe como parámetro una cola por referencia  $C$  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. Una vez terminado el proceso se copia la cola  $C1$  en la propia cola  $C$ .

**La codificación de este problema se encuentra en la página Web del libro.**

**18.12.** Un ejemplo de una frase palíndroma es: “dabale arroz a la zorra el abad”. Para resolver el problema basta con seguir la siguiente estrategia: añadir cada carácter de la frase que no sea blanco a una pila y a la vez a una cola. La extracción simultánea de caracteres de ambas y su comparación determina si la frase es o no palíndroma. La solución está dada en la función  $palindroma$ . Se usa además una función,  $SonIguales$ , que decide si una Pila y una Cola que recibe como parámetros constan de los mismos caracteres.

**La codificación de este problema se encuentra en la página Web del libro.**

**18.13.** La implementación que se realiza está basada en la cola circular explicada en la teoría. Se han omitido algunas primitivas de gestión de la cola, dejándose solamente el constructor, destructor,  $AnadeC$  que añade un elemento a la cola y  $BorrarC$  que borra un elemento de la cola y lo retorna. La zona protegida tiene declarado un puntero  $A$  al tipo genérico  $TipoDato$  que es declarado en una plantilla de clases. A este puntero genérico  $A$  se le reserva memoria en el momento que se

llama al constructor, y en el caso de que se produzca algún error en la memoria se lanza la correspondiente excepción. Para poder lanzar las excepciones de los métodos que añaden y borrar un dato de la cola, se define las funciones miembro ocultas en una zona privada `EsvaciaC` y `EstallenaC` que deciden si la cola está vacía o llena. La función `miembro` o `AnadeC` añade un dato a la cola si hay espacio, o bien lanza una excepción en el caso de que no pueda hacerlo por estar la cola llena. La función `miembro` o `BorraC` elimina el primer elemento de la cola si lo hay, y en otro caso lanza la excepción correspondiente. En el programa principal se declara una cola de 2 datos y se recogen las excepciones correspondientes en el caso de que las hubiera.

La codificación de este problema se encuentra en la página Web del libro.

## EJERCICIOS PROPUESTOS

- 18.1. 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.
- 18.2. 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 pares de la pila en la cola.
- 18.3. Escribir un programa en el que se generen 100 números aleatorios en el rango  $-25 \dots +25$  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.
- 18.4. Escribir una función que invertira el contenido de una cola usando los métodos de una clase `Cola`.

## PROBLEMAS PROPUESTOS

- 18.1. 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()`.
- 18.2. 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:  
 $\text{frente} = 5 \quad \text{final} = 7 \quad \text{Bicola: A,C,E}$   
 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.
- 18.3. Con un archivo de texto se quieren realizar las siguientes acciones: formar una lista de colas, de tal forma que en 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`, así sucesivamente.
  - 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.
- 18.4. 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 escribir el contenido de las  $n$  pilas en pantalla.

- 18.5.** Se trata de crear una cola de mensajes que sirve 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 archivo binario del que pueda recogerlos en la siguiente ejecución.

- 18.6.** Escriba una función que reciba una expresión en notación postfija y la evalúe.
- 18.7.** Una variante del conocido problema de José es el siguiente. Determinar los últimos datos que quedan de una lista inicial de  $n > 5$  números sometida al siguiente algoritmo: se inicializa  $n1$  a 2. Se retiran de la lista los números que ocupan las posiciones  $2, 2+n1, 2+2*n1, 2 + 3 * n1, \dots$ , etc. Se incrementa  $n1$  en una unidad. Si quedan en la lista menos de  $n1$  elementos se para, en otro caso se itera. Escribir un programa que resuelva el problema.



## CAPÍTULO 19

# Recursividad

## Introducción

La **recursividad** (*recursión*) es la propiedad que posee una función de permitir que dicha función pueda llamarse así 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 la programación. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa; 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.

### 19.1. La naturaleza de la recursividad

Una función *recursiva* es aquella que se llama a sí mismo 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 tiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es *recursiva*.

**EJEMPLO 19.1.** Función recursiva que calcula el factorial de un número  $n > 0$ .

Es conocido que  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4! = 120$ . Es decir  $n! = n \cdot (n - 1)!$   
Así pues, la definición recursiva de la función factorial es la siguiente:

Factorial( $n$ ) =  $n * \text{Factorial}(n-1)$  si  $n > 0$   
Factorial( $n$ ) = 1 si  $n = 0$ .

Una codificación de la función es:

```
int Factorial(int n)
{
 if (n == 0)
 return 1;
 else
 return (n * Factorial (n - 1));
}
```

## 19.2. Funciones recursivas

Una función **recursiva** es una función que se invoca a sí mismo 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. En consecuencia, la definición recursiva debe incluir un **componente base (condición de salida)** en el que `f(n)` se defina directamente (es decir, no recursivamente) para uno o más valores de `n`. Cualquier algoritmo que genere una secuencia de este tipo puede no terminar nunca. Toda función recursiva usa una pila de llamada. Cada vez que se llama a una función recursiva todos los valores de los parámetros formales y variables locales son almacenados en una pila. Cuando termina de ejecutarse una función recursiva se retorna al nivel inmediatamente anterior, justo en el punto de programa siguiente al que produjo la llamada. En la pila se recuperan los valores tanto de los parámetros como de las variables locales, y se continúa con la ejecución de la siguiente instrucción a la llamada recursiva.

**EJEMPLO 19.2.** Función recursiva directa que cuenta, visualizando los resultados y mostrando el funcionamiento de la pila interna.

El siguiente programa permite observar el funcionamiento de la recursividad. La función `contar` decremente el parámetro y lo muestra y si el valor es mayor que cero, se llama a sí misma donde decremente el parámetro una vez más en caso de que sea positivo, y en caso de que no sea positivo no hace nada, que es la parada de la función. Cuando el parámetro alcanza el valor de cero la función ya no se llama a sí misma y se producen sucesivos retornos al punto siguiente a donde se efectuaron las llamadas, donde se recupera de la pila el valor del parámetro y se muestra.

```
#include <cstdlib>
#include <iostream>

using namespace std;

void contar(int n)
{
 n--;
 cout << " " << n << endl;
 if (n > 0)
 contar(n);
 cout << " " << n << endl;
}

int main(int argc, char *argv[])
{
 int n;

 do
 {
 cout << " Escriba un numero entero positivo: ";
 cin >> n;
 } while (n <= 0);
 contar(n);
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

**EJEMPLO 19.3.** Funciones recursivas indirectas que visualizan el alfabeto.

El programa principal llama a la función recursiva `A()` con el argumento 'Z' (la última letra del alfabeto). La función `A` examina su parámetro `c`. Si `c` está en el orden alfabético después que 'A', la función llama a `B()`, que inmediatamen-

te llama a `A()`, pasándole un parámetro predecesor de `c`. Esta acción hace que `A()` vuelva a examinar `c`, y nuevamente una llamada a `B()`, hasta que `c` sea igual a '`A`'. En este momento, la recursión termina ejecutando `cout << c;` veintiséis veces y visualizando el alfabeto, carácter a carácter.

```
#include <cstdlib>
#include <iostream>

using namespace std;
void A(char c);
void B(char c);

void A(char c)
{
 if (c > 'A')
 B(c);
 cout << c;
}

void B(char c)
{
 A(--c);
}

int main(int argc, *argv[])
{
 A('Z');
 cout << endl;
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

**EJEMPLO 19.4.** *Se implementan dos funciones con recursividad indirecta.*

La función `f` retorna el valor de su parámetros si éste es menor o igual que `3`. En otro caso llama a la función `g` decrementando el parámetro en dos unidades, y retornado el valor obtenido más el propio parámetro. Por su parte la función `g` retorna el valor de su parámetro si éste es menor o igual que `2`. En otro caso retorna el valor obtenido de la llamada a la función `f` decrementando el parámetro en dos unidades más el propio parámetro.

```
float f(float y);
float g(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);
}
```

## 19.3. 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 utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas a funciones. La recursión inicia repetidamente al mecanismo de llamadas a funciones y, en consecuencia, se necesita un tiempo y espacio suplementario para realizar cada llamada. Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. 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 19.5. Versión recursiva e iterativa de la función de Fibonacci.

La función de Fibonacci se define recursivamente de la siguiente forma:

$\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$  si  $n > 1$ ;  
 $\text{Fibonacci}(n) = n$  en otro caso.

Para realizar la programación iterativa, hay que observar que la secuencia de números de Fibonacci es: 0, 1, 1, 2, 3, 5, 8, 13 ... Esta secuencia se obtiene partiendo de los números 0, 1 y a partir de ellos cada número se obtiene sumando los dos anteriores:  $a_n = a_{n-1} + a_{n-2}$ . Si se definen dos variables locales  $a1$  y  $a2$ , convenientemente inicializadas a 0 y a 1 respectivamente, el siguiente valor de la serie se obtiene sumando  $a1$  y  $a2$  en otra variable local  $a3$ . Si ahora se hace  $a1 = a2$  y  $a2 = a3$ , se tienen en estas dos variables los dos siguientes números de la serie, por lo que al iterar se van obteniendo todos los sucesivos valores.

```
long Fibonacci(int n) // recursiva
{
 if (n == 0 || n == 1)
 return (n);
 else
 return(Fibonacci(n - 1) + Fibonacci(n - 2));
}

long FibonacciIterativa(int n)
{
 long a1 = 0, a2 = 1, a3, i;
 for (i = 2; i <= n; i++)
 {
 a3 = a1 + a2;
 a1 = a2;
 a2 = a3;
 }
 return a3;
}
```

## 19.4. 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. Una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida. El flujo de control de una función recursiva requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (*condición de salida o caso base*).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

**EJEMPLO 19.6.** *Función recursiva que calcula la suma de los cuadrados de los N primeros números positivos.*

La función `sumacuadrados` implementada recursivamente, requiere la definición previa de la suma de los cuadrados primeros  $N$  enteros matemáticamente en forma recursiva tal como se muestra a continuación:

$$\text{suma}(N) = \begin{cases} 1 & \text{si } N = 1 \\ N^2 + \text{suma}(N-1) & \text{en caso contrario} \end{cases}$$

```
int sumacuadrados(int n)
{
 if (n == 1) //test para parar o continuar (condición de salida)
 return 1; //caso final. Se detiene la recursión
 else
 return n*n + sumacuadrados (n - 1); //caso recursivo. la recursión
 //continúa con llamada recursiva
}
```

Esta recursión para siempre que se llame a la función con un valor de  $n$  positivo, pero se produce una recursividad infinita en caso de que se llame con un valor de  $n$  negativo o nulo.

**EJEMPLO 19.7.** *Función recursiva que calcula el producto de dos números naturales, usando sumas.*

El producto de dos números naturales  $a$  y  $b$  se puede definir recursivamente de la siguiente forma:  $\text{Producto}(a, b) = 0$  si  $b = 0$ ;  $\text{Producto}(a, b) = a + \text{Producto}(a, b - 1)$  si  $b > 0$ . Esta función tendrá una llamada recursiva infinita si es llamada con el valor del segundo parámetro negativo.

```
int Producto(int a, int b)
{
 if (b == 0)
 return 0;
 else
 return a + Producto(a, b - 1);
}
```

## EJERCICIOS

**19.1.** *Explique porqué la siguiente función puede producir un valor incorrecto cuando se ejecute:*

```
long factorial (long n)
{
 if (n == 0 || n == 1)
 return 1;
 else
 return n * factorial (--n);
}
```

- 19.2.** ¿Cuál es la secuencia numérica generada por la función recursiva *f* en el listado siguiente?

```
long f(int n)
{
 if (n == 0 || n == 1)
 return 1;
 else
 return 3 * f(n - 2) + 2 * f(n - 1);
}
```

- 19.3.** Cuál es la secuencia numérica generada por la función recursiva siguiente?

```
int f(int n)
{
 if (n == 0)
 return 1;
 else if (n == 1)
 return 2;
 else
 return 2 * f(n - 2) + f(n - 1);
}
```

- 19.4.** Escribir una función que calcule el elemento mayor de un array de *n* enteros recursivamente.

- 19.5.** Escribir una función recursiva que realice la búsqueda binaria de una clave en un vector ordenado creciente y recursivamente.

## PROBLEMAS

- 19.1.** Escribir una función que calcule la potencia  $a^n$  recursivamente, siendo *n* positivo.

- 19.2.** Escribir una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

$$\begin{array}{ll} A(m, n) = n + 1 & \text{si } m = 0 \\ A(m, n) = A(m - 1, 1) & \text{si } n = 0 \\ A(m, n) = A(m - 1, A(m, n - 1)) & \text{si } m > 0, \text{ y } n > 0 \end{array}$$

- 19.3.** Escribir una función recursiva que calcule el cociente de la división entera de *n* entre *m*, siendo *m* y *n* dos números enteros positivos recursivamente.

- 19.4.** Escribir un programa que mediante una función recursiva calcule la suma de los *n* primeros números pares, siendo *n* un número positivo.

- 19.5.** Escribir un programa en C++ que mediante recursividad indirecta decida si un número natural positivo es par o impar.

- 19.6.** Escribir una función recursiva para calcular el máximo común divisor de dos números naturales positivos.

- 19.7.** Escribir una función recursiva que lea números enteros positivos ordenados decrecientemente del teclado, elimine los repetidos y los escriba al revés, es decir, ordenado crecientemente. El fin de datos viene dado por el número especial 0.

- 19.8.** Escribir 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  $1.0e^{-8}$ .

- 19.9.** Escribir una función recursiva que sume los dígitos de un número natural.

- 19.10.** Escribir una función recursiva, que calcule la suma de los elementos de un vector de reales  $v$  que sean mayores que un valor  $b$ .

- 19.11.** Escribir una clase `ListaS` que con la ayuda de una clase `Nodo`, permita la implementación de listas enlazadas recursivas con objetos. Incluir en la clase `ListaS` funciones miembro que permitan insertar recursivamente, y borrar recursivamente, en una lista enlazada ordenada crecientemente, así como mostrar recursivamente una lista.

- 19.12.** Añada a la clase `ListaS` del Problema 19.11 que implementa una lista enlazada ordenada recursivamente con objetos funciones miembro que permitan la inserción y borrado iterativo.

## SOLUCIÓN DE LOS EJERCICIOS

- 19.1.** La función factorial escrita produce un bucle infinito si se le llama por primera vez con un valor de  $n$  negativo. Además si se supone que la función ha sido escrita para calcular el factorial de un número  $o$ , hay que observar que no produce el resultado deseado. La ejecución de la función cuando  $n$  es mayor que 1 comienza con una llamada recursiva que decremente el valor de  $n$  en una unidad, por lo que cuando se ejecuta el producto no multiplica por  $n$ , sino por  $n - 1$ . Es decir, la función en lugar de calcular el factorial de un número  $n$  calcula el factorial de  $n-1$ , si  $n$  es mayor o igual que 2. Si se ejecuta el siguiente programa se obtiene el resultado que visualiza lo explicado.

La codificación de este ejercicio se encuentra en la página Web del libro.

- 19.2.** Para el caso de que  $n$  valga 0 o 1 la función retorna siempre 1. En otro caso retorna el valor de  $3 * f(n - 2) + 2 * f(n - 1)$ . Es decir, multiplica el valor anterior de la función por 2, y se lo suma al triple del valor obtenido por la función en el valor anterior del anterior. Por tanto, la serie es la siguiente: 1, 1, 5, 13, 41, 121, 365, 1093, 3281, 9841. Por ejemplo,  $41 = 5*3 + 13*2$ .

- 19.3.** Si  $n$  vale 0, retorna el valor de 1, si  $n$  vale 2, retorna el valor de 2, si  $n$  vale 3, retorna el valor de  $2*1 + 2 = 4$ , si  $n$  vale 4, retorna el valor de  $2*2 + 4 = 8$ . La secuencia de ejecución es: 1, 2, 4, 8, 16, 32, 64, 128, 256, ...

- 19.4.** Se define en primer lugar la función `int max(int x, int y);` que devuelve el mayor de dos enteros  $x$  e  $y$ . Se define posteriormente la función `int maxarray(int a[], int n)` que utiliza la recursión para devolver el elemento mayor de  $a$ .

Condición de parada:  $n == 1$

Incremento recursivo:  $\text{maxarray} = \text{max}(\text{maxarray}(a[0] \dots a[n-2]), a[n-1])$

La codificación de este ejercicio se encuentra en la página Web del libro.

- 19.5.** La función `busquedaBR`, recibe como parámetro un vector, la clave y el rango inferior y superior donde se debe realizar la búsqueda. La función retorna una posición del vector donde se encuentra la clave, o bien el valor de -1 en caso de que no se encuentre. La búsqueda se basa en calcular el centro de la lista que se encuentra a igual distancia de la parte inferior y la superior. La búsqueda termina con éxito, si en centro se encuentra la clave. Se realiza una llamada recursiva a

la izquierda de centro en el caso de que la clave sea mayor que el elemento que ocupa la posición y la llamada recursiva se realiza a la derecha en otro caso. Hay que tener en cuenta que clave puede no encontrarse en el vector. Esta posibilidad se detecta cuando el rango inferior excede al rango superior.

**La codificación de este ejercicio se encuentra en la página Web del libro.**

## SOLUCIÓN DE LOS PROBLEMAS

- 19.1. La potencia de  $a^n$  se puede definir recursivamente de la siguiente forma:  $a^n = 1$  si  $n = 0$  y es  $a * n^{n-1}$  en otro caso. Hay que observar que sólo es válido para valores de  $n$  positivo.

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.2. La función de Ackerman es un ejemplo de función recursiva anidada, donde la llamada recursiva utiliza como parámetro el resultado de una llamada recursiva.

**La codificación de este problema se encuentra en la página Web del libro.**

Los sucesivos valores de la serie son:

| $A(m, n)$ | $n = 0$ | $n = 1$ | $n = 2$       | $n = 3$ | $n = 4$ | $n = 5$ | $n = 6$ |
|-----------|---------|---------|---------------|---------|---------|---------|---------|
| $m = 0$   | 1       | 2       | 3             | 4       | 5       | 6       | 7       |
| $m = 1$   | 2       | 3       | 4             | 5       | 6       | 7       | 8       |
| $m = 2$   | 3       | 5       | 7             | 9       | 11      | 13      | 15      |
| $m = 3$   | 5       | 13      | 29            | 61      | 125     | 253     | 509     |
| $m = 4$   | 13      | 65533   | $2^{65536}-3$ |         |         |         |         |

- 19.3. El cociente de la división enter a de n entre m, siendo ambos números enteros positivos se calcula recursivamente de la siguiente forma si  $n < m$  entonces  $\text{cociente}(n, m) = 0$ , si  $n \geq m$  entonces  $\text{cociente}(n, m) = 1 + \text{cociente}(n - m, m)$ , por lo que la codificación de la función es:

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.4. La suma de los n primeros números pares, siendo n positivo, viene dada por la expresión:

$$S = 2 * 1 + 2 * 2 + \dots + 2 * (n - 1) + 2 * n = \sum_{i=1}^n 2 * i = (n + 1)n$$

El programa solicita al usuario el valor de n validando la entrada y ejecuta la función sumapares, que recibe como argumento el valor de i. Si este valor es mayor que 1 obtiene la suma ejecutando la sentencia  $2 * i + \text{sumapares}(i-1)$ . En otro caso, la suma del primer número par es siempre 2.

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.5. Se programan dos funciones mutuamente recursivas que deciden si un número es par, para lo que se tiene en cuenta que:

```
par(n) = impar(n - 1) si n > 1.
par(0) = true
impar(n) = par(n - 1) si n > 1.
impar(0) = false.
```

Se incluye, además, un programa que realiza una llamada a la función par.

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.6.** El máximo común divisor tiene una serie de axiomas que la definen de la siguiente forma:  $Mcd(m, n) = Mdc(n, m)$  si  $m < n$ ;  $Mcd(m, n) = Mdc(n, m \ mod \ n)$  si  $m \geq n$  y  $m \ mod \ n \neq 0$ ;  $Mcd(m, n) = n$  si  $n \leq m$  y  $n \ mod \ m = 0$ . Por lo que una codificación es la siguiente:

```
int Mcd(int m, int n)
{
 if ((n <= m) && (m % n == 0))
 return n;
 else if (m < n)
 return Mcd(n, m);
 else
 return Mcd(n, m % n);
}
```

- 19.7.** 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.

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.8.** La función `loge()`, calcula iterativamente la suma de la serie indicada de la siguiente forma: la variable `delta` contiene 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` contiene 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` contiene 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  $1.0 \ e^{-8}$ , 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  $1.0 \ e^{-8}$  se retorna el valor de `0`, ya que se ha terminado de sumar la serie.

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.9.** La suma de los dígitos de un número natural positivo se puede calcular recursivamente mediante la expresión.

$$\text{SumaRecursiva}(n) = \begin{cases} n \bmod 10 + \text{SumaRecursiva}(n \bmod 10) & \text{si } n > 0 \\ 0 & \text{en otro caso} \end{cases}$$

**La codificación de este problema se encuentra en la página Web del libro.**

- 19.10.** Se programa la función `suma` 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 - 1`. De esta forma si `n` vale `1` el problema tiene una solución trivial: si `v[0] <= b` devuelve `0` y si `v[0] > b` devuelve `v[0]`. Si `n` es mayor que `1`, entonces, debe calcularse recursivamente la suma desde la posición `0` hasta la `n - 2` y después sumar `v[n - 1]` en el caso de que `v[n - 1]` sea mayor que `b`.

**La codificación de este problema se encuentra en la página Web del libro.**

**19.11.** Se declara en primer lugar el tipo elemento como un enter o en un `typedef`. Posteriormente, se avisa que existe la clase `Nodo`, que será codificada posteriormente. De esta forma la clase `ListaS`, puede tener un atributo protegido `p` que es un puntero a la clase `Nodo`. En la clase `ListaS`, se declaran los constructores y los destructores de la forma estándar, y además sólo tres funciones miembro que permiten: insertar un elemento en una lista enlazada ordenada crecientemente de forma recursiva; borrar un elemento en una lista enlazada ordenada crecientemente recursivamente; mostrar una lista enlazada recursivamente. La clase `Nodo` se declara con dos atributos que son: el elemento `e` y un atributo `Sig` que es de tipo `ListaS`. Esta declaración cruzada entre las dos clases `Nodo` y `ListaS`, permite implementar de una forma sencilla objetos recursivos. Las funciones miembro de la clase `Nodo` se codifican de la forma estándar, tal y como se ha hecho en los capítulos 17 y 18 de Listas enlazadas, Pilas y Colas. Sólo se incluye una pequeña diferencia en la función miembro `OSig()`, que retorna una referencia a una `ListaS`, en la que se devuelve el atributo `Sig` en lugar de una copia del atributo `Sig`, como se hizo en los capítulos indicados. Esta pequeña modificación permite modificar el propio atributo `Sig` de un `Nodo`, y facilitar la implementación recursiva de las funciones miembro de la clase `ListaS`. La implementación de las funciones recursivas de la clase `ListaS` son:

- `ListaS & ListaS:: Inserta0rec( Telemento x )`. Si la lista está vacía entonces la inserción de un nuevo nodo consiste en crear un nodo, con la información correspondiente y ponerlo como primer elemento de la lista. Si la lista no está vacía, hay dos posibilidades, que el elemento a insertar sea mayor que el elemento que se encuentra a en la posición actual, en cuyo caso basta con insertarlo recursivamente en su nodo siguiente, o que no lo sea, en cuyo caso hay que: crear un nuevo nodo con la información; enlazar su atributo `Sig` con `*this`; y decir que el nuevo primer elemento de la lista es el propio nodo. Una vez terminado todo el proceso se retorna `*this`.
- `ListaS & ListaS:: Borrar0rec( Telemento x )`. Si la lista está vacía, entonces nunca se encuentra el elemento en la lista. En otro caso hay tres posibilidades: que la información contenida en el nodo actual de la lista sea mayor que el elemento a borrar en cuyo caso nunca se puede encontrar el elemento en la lista; que la información el nodo actual de la lista sea menor que el elemento a borrar en cuyo caso basta con llamar recursivamente a borrar en el nodo siguiente; que el nodo actual tenga una información que coincida con el elemento a borrar, en cuyo caso se enlaza el puntero de la lista con el atributo `Sig` del nodo actual, y se destruye el nodo actual. Una vez terminado todo el proceso se retorna `*this`.
- `void ListaS::muestrarec()`. Si hay datos, muestra la información actual, y posteriormente muestra recursivamente el siguiente.

En la implementación se incluye, además, un programa principal que inserta y borra en listas ordenadas, así como un resultado de ejecución.

**La codificación de este problema se encuentra en la página Web del libro.**

**19.12.** La inserción iterativa, en la lista declarada en el Problema 19.11, comienza con la búsqueda de la posición donde colocar el elemento, dejando el objeto `ant` y `act` en listas que apuntan al nodo anterior y actual donde debe ser añadido el nuevo elemento. Posteriormente, se procede a la inserción del nuevo nodo teniendo en cuenta que puede ser el primer elemento de la lista, o no serlo. El borrado iterativo comienza con la búsqueda del elemento a borrar, dejando el objeto `ant` y `act` en listas que apuntan al nodo anterior y actual donde debe ser borrado el nodo. Posteriormente si la búsqueda ha terminado con éxito se procede al borrado del nodo, puenteadolo, teniendo en cuenta que puede que sea o no el primer elemento de la lista.

**La codificación de este problema se encuentra en la página Web del libro.**

## EJERCICIOS PROPUESTOS

- 19.1. Escribir 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.
```

- 19.2. ¿Cuál es la secuencia numérica generada por la función recursiva `f` en el listado siguiente?

```
int f(int x)
{
 if (x <= 0)
 return 2;
 else
 return(n + 2 * f(n - 2));
}
```

- 19.3. ¿Cuál es la secuencia numérica generada por la función recursiva `f` en el listado siguiente?

```
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));
}
```

## PROBLEMAS PROPUESTOS

- 19.1. Escriba una función recursiva de prototipo `int consonantes(const char * cd)` para calcular el número de consonantes de una cadena.

- 19.2. Escriba una función recursiva que calcula el producto escalar de dos vectores de  $n$  elementos recursivamente.

- 19.3. Escriba una función recursiva que sume los  $n$  primeros números naturales que sean múltiplo de 3.

- 19.4. Escriba un programa que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe llamar a una función que calcule el mayor de todos los enteros introducidos recursivamente.

- 19.5. Leer un número entero positivo  $n < 10$ . Calcular el desarrollo del polinomio  $(x + 1)^n$ . Imprimir cada potencia  $x^i$  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:

$$C(n, 0) = 1$$

$$C(n, n) = 1$$

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$

- 19.6. Dadas las letras del abecedario a, b, c, d, e, f, g, h, i, j y dos números enteros  $0 < n < = m < = 10$ , escriba un programa que calcule las variaciones de los  $m$  primeros elementos tomados de  $n$  en  $n$ .

- 19.7. Añada a la clase `ListaS` del Problema resuelto 19.11 funciones miembro que permitan la inserción recursiva y el borrado de recursivo de un nodo que ocupe la posición  $pos$ .



## CAPÍTULO 20

# Árboles

## Introducción

Los árboles son estructuras de datos *no lineales* al contrario que los arrays y las listas enlazadas que constituyen *estructuras lineales*.

Los árboles son muy utilizados para representar fórmulas algebraicas, como método eficiente para búsquedas grandes y complejas, listas dinámicas, etc. 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, proceso de textos 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 de dato estructurado que representa una estructura jerárquica entre sus elementos. La definición de un árbol viene dada recursivamente de la siguiente forma:

Un **árbol** es un conjunto de uno o nodos tales que:

1. Hay un nodo diseñado especialmente llamado **raíz**.
2. Los nodos restantes se dividen en  $n \geq 0$  conjuntos disjuntos tales que  $T_1 \dots T_n$ , son un árbol. A  $T_1, T_2, \dots T_n$  se les denomina **subárboles** del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz**. La Figura 20.1 muestra un árbol, general.

### Terminología árboles

- El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
- Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas**.
- Los nodos terminales, (nodos de los cuales no se deduce ningún nodo), se denominan **hojas**.
- Los nodos que no son hojas se denominan **nodos internos** o **nodos no terminales**.
- Si en un árbol una rama va de un modo  $n_1$  a un nodo  $n_2$ , se dice que  $n_1$  es el **padre** de  $n_2$  y que  $n_2$  es un **hijo** de  $n_1$ .
- $n_1$  se llama **ascendiente** de  $n_2$  si  $n_1$  es el padre de  $n_2$  o si  $n_1$  es el padre de un ascendiente de  $n_2$ .
- $n_2$  se llama **descendiente** de  $n_1$  si  $n_1$  es un ascendiente de  $n_2$ .

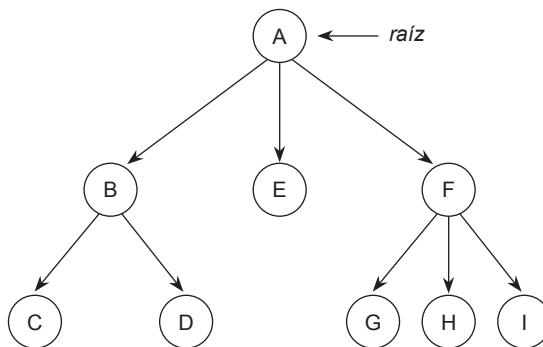
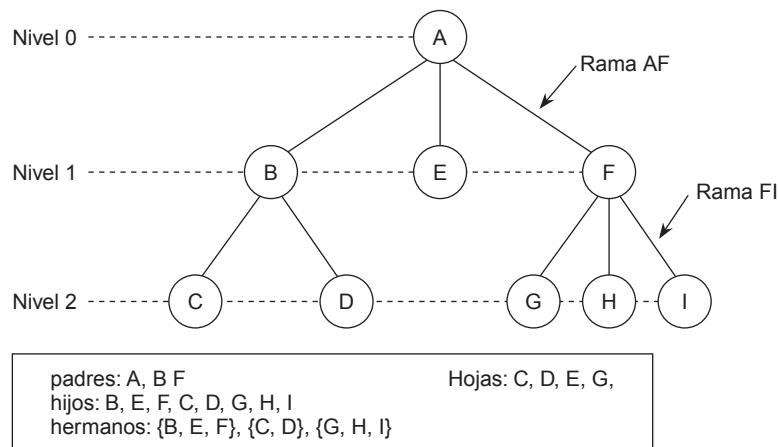


Figura 20.1. Árbol.

- Un **camino** de  $n_1$  a  $n_2$  es una secuencia de arcos contiguos que van de  $n_1$  a  $n_2$ .
- La **longitud de un camino** es el número de arcos que contiene (en otras palabras, el número de nodos –1).
- El **nivel** de un nodo es la longitud del camino que lo conecta al raíz.
- La **profundidad o altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja.
- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es a su vez un árbol.
- Sea SA un subárbol de un árbol A: si para cada nodo  $n$  de SA, SA contiene también todos los descendientes de  $n$  en A. SA se llama un **subárbol completo** de A.
- Un árbol está **equilibrado** cuando, dado un número máximo  $K$  de hijos de cada nodo y la altura del árbol  $h$ , cada nodo de nivel  $k < h - 1$  tiene exactamente  $K$  hijos.

**EJEMPLO 20.1.** Árbol que ilustra algunas definiciones.



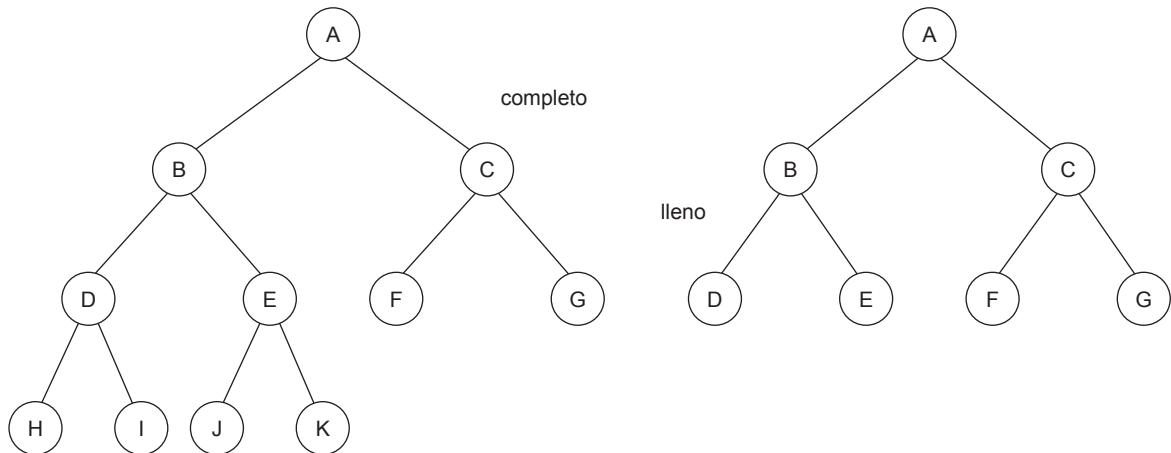
## 20.2. Árboles binarios

Un **árbol binario** es aquél en que cada nodo tiene como máximo grado dos. Por tanto, un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*.

- Un árbol binario está *perfectamente equilibrado*, si los subárboles de todos los nodos tienen la misma altura.
- Un árbol binario se dice que es *completo* si todos los nodos *interiores*, es decir aquéllos 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.** *Árbol binario completo y lleno.*



## 20.3. Estructura y representación de un árbol binario

La estructura de un árbol binario es aquella que en cada nodo se almacena un dato y su hijo izquierdo e hijo derecho. En C++ puede representarse de la siguiente forma.

1. La clase nodo se encuentra anidada dentro de la clase árbol.

```
class arbol
{
 class nodo
 {
 public:
 char *datos;
 private:
 nodo *derecho;
 nodo *izquierdo;
 friend class arbol;
 };
public:
 nodo *raiz; // raíz del árbol (raíz)
 arbol() {raiz = NULL; }
 // ... otras funciones miembro
};
```

2. La clase `ArbolBin` no tiene dentro la clase `NodoArbol`, pero es declarada como clase amiga. Se realiza además con tipos genéricos.

```
template <class T>
class ArbolBin; // aviso a la clase NodoArbol.

// declara un objeto nodo árbol de un árbol binario
template <class T>
```

```

class NodoArbol
{
 private:
 // apunta a los hijos izquierdo y derecho del nodo
 NodoArbol <T> *izquierdo;
 NodoArbol <T> *derecho;
 T datos;
 public:

 // constructor
 NodoArbol (T item, NodoArbol <T> *ptri,NodoArbol <T> *ptrd)
 {
 datos = item;
 izquierdo = ptri;
 derecho = ptrd;
 }

 // métodos de acceso a los atributos puntero
 NodoArbol <T>* OIzquierdo() {return izquierdo;}
 NodoArbol <T>* ODerecho(void) {return derecho;};
 void PIzquierdo(NodoArbol <T>* Izq) {izquierdo = Izq;}
 void PDerecho(NodoArbol <T>* Drc) {derecho = Drc;}

 // métodos de acceso al atributo dato
 T Odatos() { return datos;}
 void Pdatos(T e){ datos = e;}
 // hacer a ArbolBin un amigo ya que necesita acceder a los
 // campos puntero izquierdo y derecho del nodo
 friend class ArbolBin <T>;
 };
 template <class T>
 class ArbolBin
 {
 private:
 NodoArbol <T> * p;
 public:
 ArbolBin() { p = NULL; }

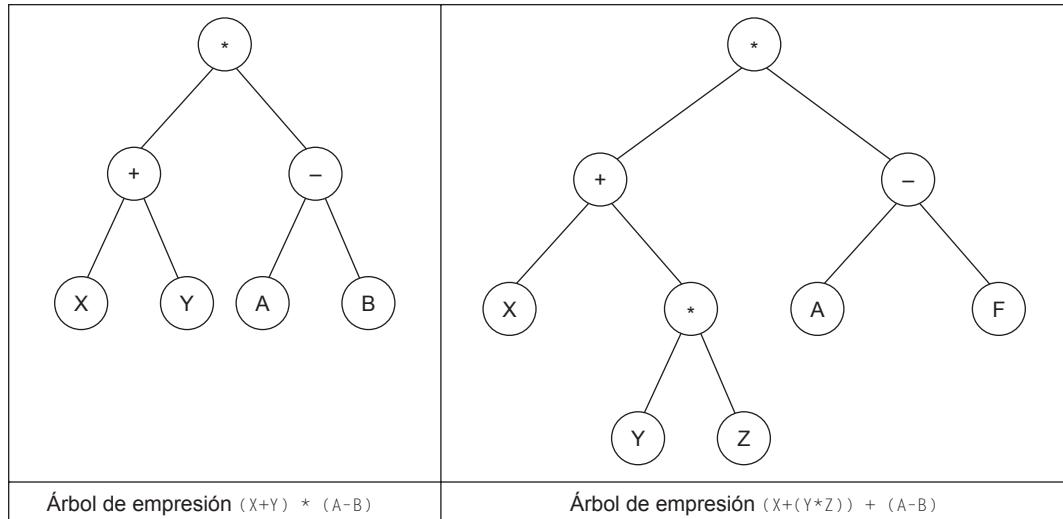
 // otras funciones miembro
 } ;
}

```

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

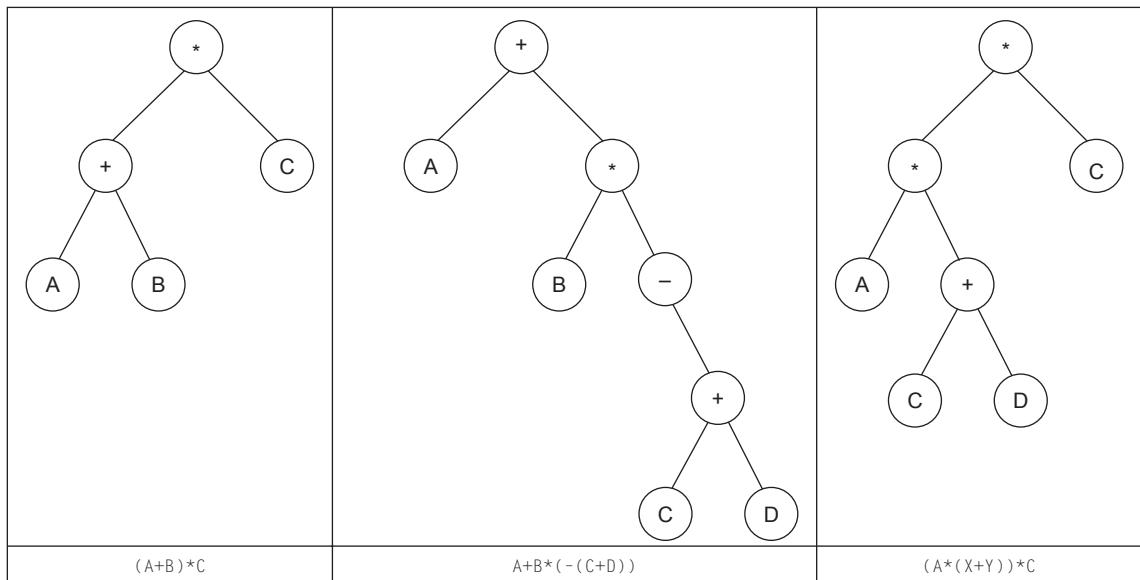
1. Cada hoja es un operando.
2. Los nodos raíz e internos son operadores.
3. Los subárboles son subexpresiones en las que el nodo raíz es un operador.

**EJEMPLO 20.3.** Árboles de expresiones.



**EJEMPLO 20.4.** Árboles de expresiones asociados a:

$(A+B)*C$ ,  $A+B*(-(C+D))$ ,  $(A*(X+Y))*C$



## 20.5. Recorridos de un árbol

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. 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

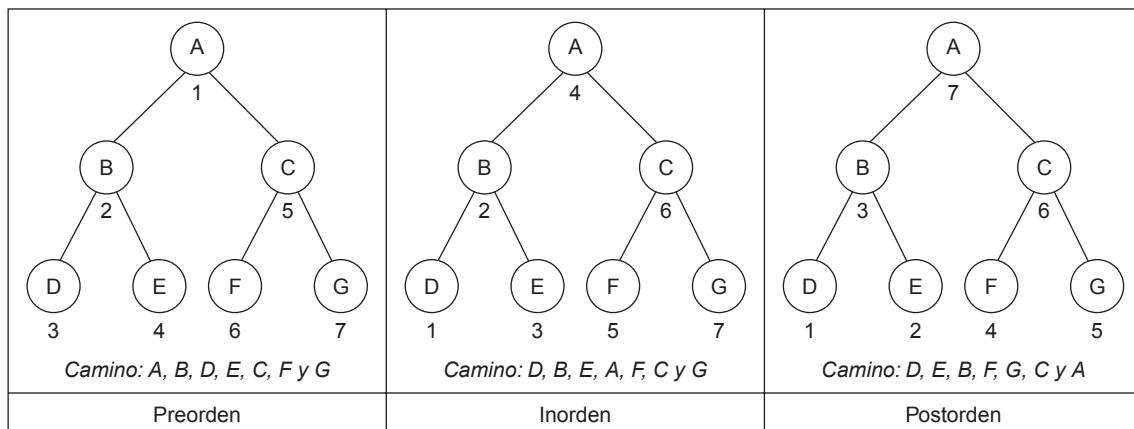
**Preorden RID.** Visitar la raíz, recorrer en preorden el subárbol izquierdo, recorrer en preorden el subárbol derecho.

**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.

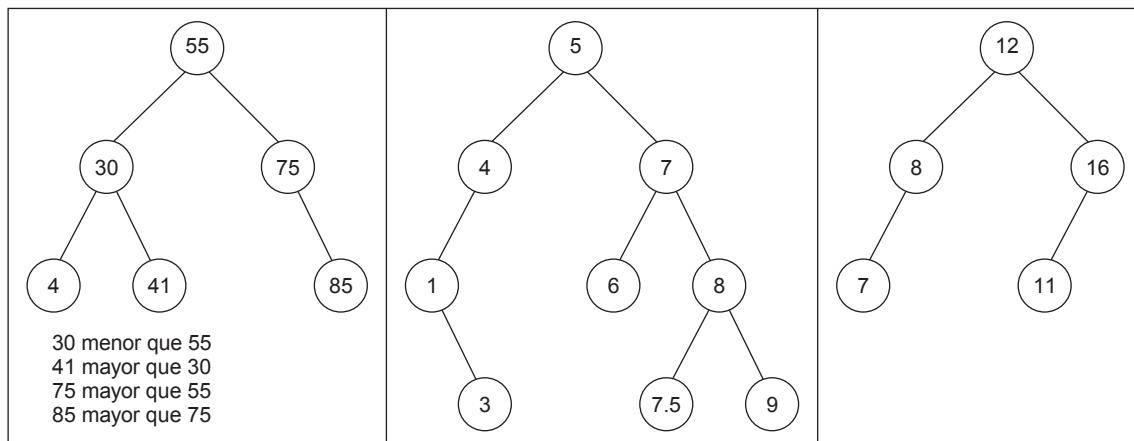
**EJEMPLO 20.5.** Recorridos en preorden, inorden y postorden.



## 20.6. Árbol binario de búsqueda

Un **árbol binario de búsqueda** es aquel que 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.

**EJEMPLO 20.6.** Árboles binarios de búsqueda.



## 20.7. Operaciones en árboles binarios de búsqueda

Las operaciones más 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.

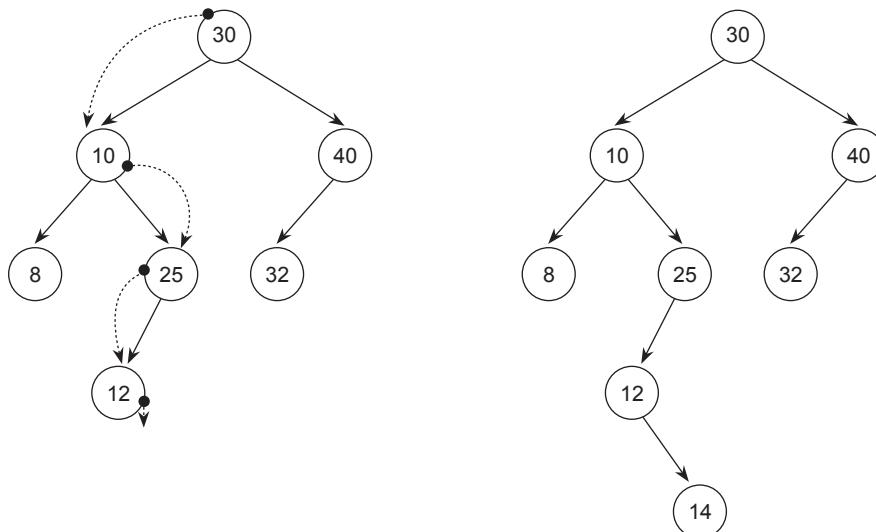
### Inserción

La operación de *inserción* de un nodo es una extensión de la operación de búsqueda. El algoritmo es:

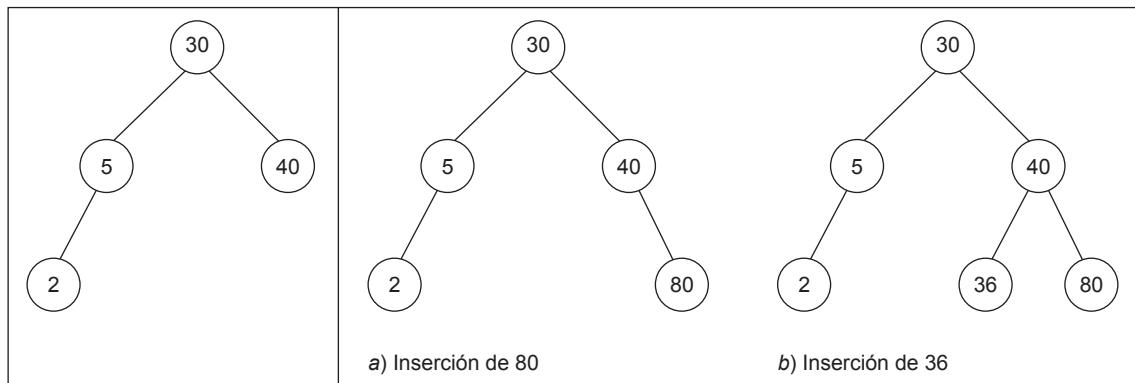
- Asignar memoria para una nueva estructura nodo.
- Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará siempre como un nuevo nodo hoja.
- Enlazar el nuevo nodo al árbol. Para ello en el proceso de búsqueda hay que quedarse con el puntero que apunta a su padre y 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.

#### EJEMPLO 20.7. Inserción de 14 en el árbol de búsqueda expresado.

El siguiente ejemplo muestra el camino a seguir para insertar el elemento de clave 14 en el árbol binario de búsqueda a continuación representado.



**EJEMPLO 20.8.** Insertar un elemento con clave 80 y 36 en el árbol binario de búsqueda siguiente:

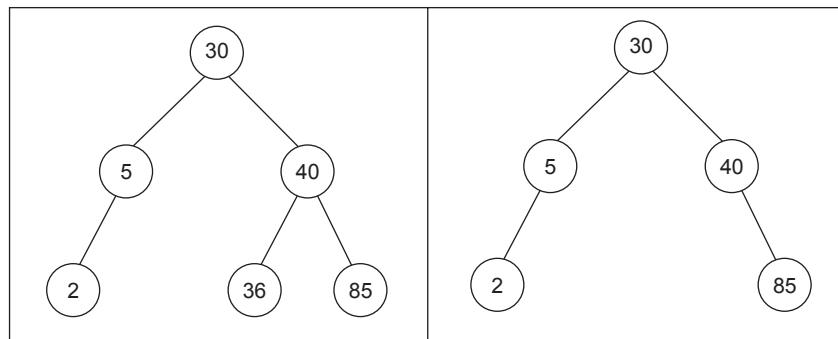


### Borrado

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:

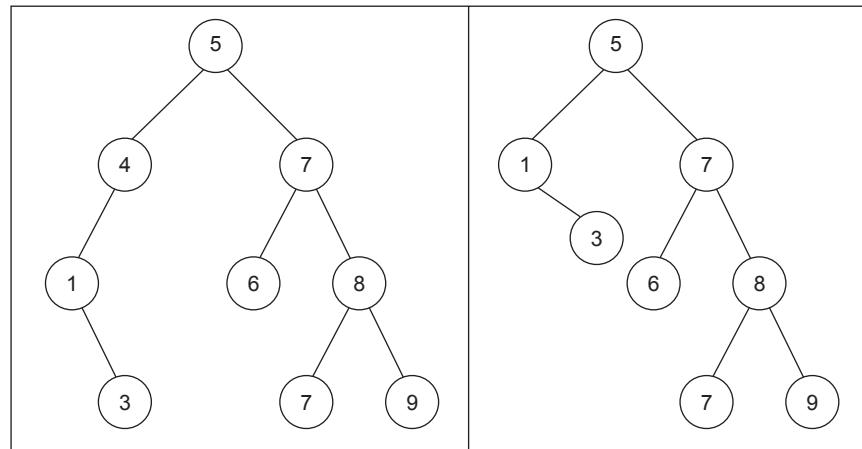
- Si el nodo es hoja, se suprime, asignando nulo al puntero de su antecesor.
- Si el nodo tiene único hijo. El nodo anterior se enlaza con el hijo del que se quiere borrar .
- Si tiene dos hijos. Se sustituye el valor almacenado en el nodo por el valor inmediato superior (o inmediato inferior). Este nodo 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.

**EJEMPLO 20.9.** Borrado de una hoja. Se borra la clave 36.



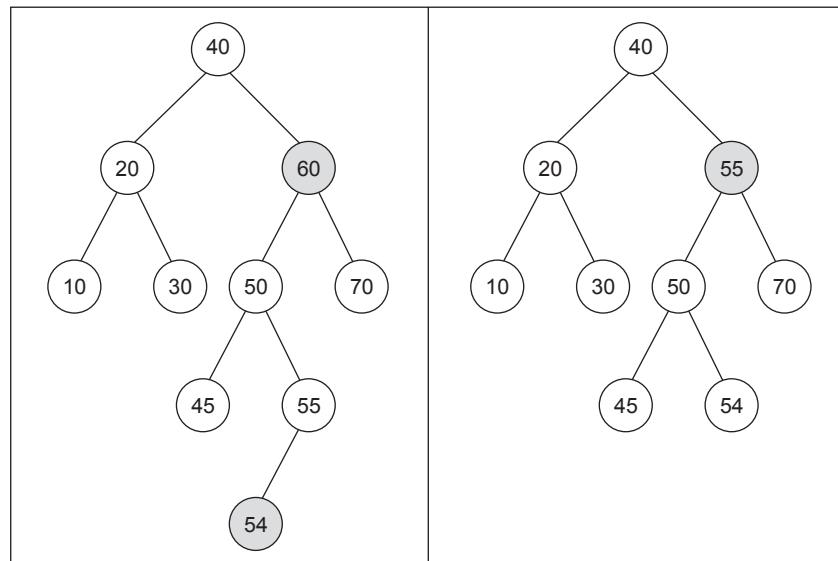
**EJEMPLO 20.10.** *Borrado de un nodo con un solo hijo. Se borra la clave 4.*

Se borra el nodo que contiene 5 puentes al nodo que contiene el 4 y se elimina el 4.



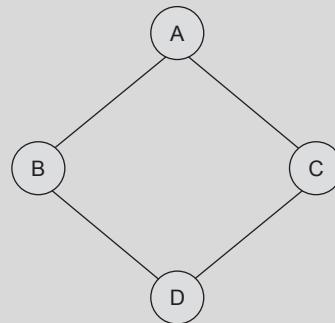
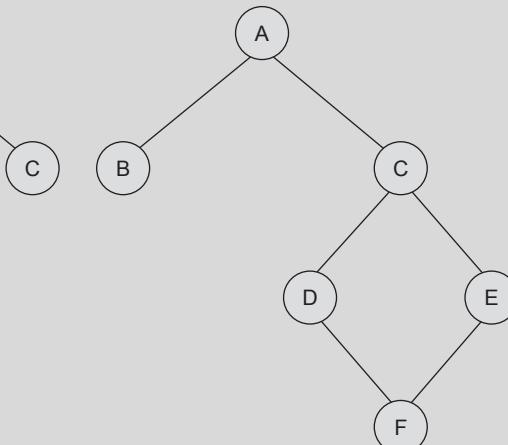
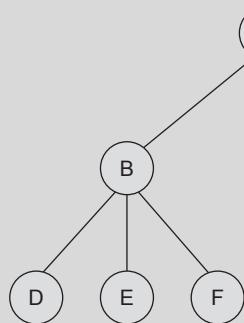
**EJEMPLO 20.11.** *Borrado de un nodo con dos hijos. Se borra la clave 60.*

Se reemplaza 60 bien con el elemento mayor (55) en su subárbol izquierdo o el elemento más pequeño (70) en su subárbol derecho. Si se opta por reemplazar con el elemento mayor del subárbol izquierdo. Se mueve el 55 al raíz del subárbol y se reajusta el árbol.



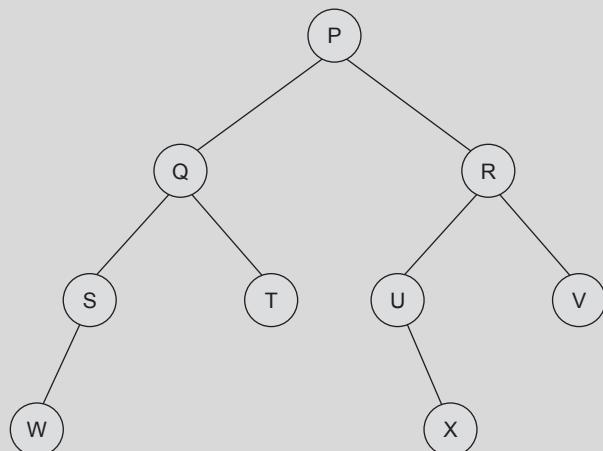
## EJERCICIOS

**20.1.** Explicar porqué cada una de las siguientes estructuras no es un árbol binario.



**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.



**20.3.** Para el árbol del ejercicio anterior realizar los siguientes recorridos: RDI, DRI, DIR.

**20.4.** 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 inorden, preorder y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.

(I) M,Y,T,E,R  
(III) T,Y,M,E,R

(II) R,E,M,Y,T  
(IV) C,O,R,N,F,L,A,K,E,S

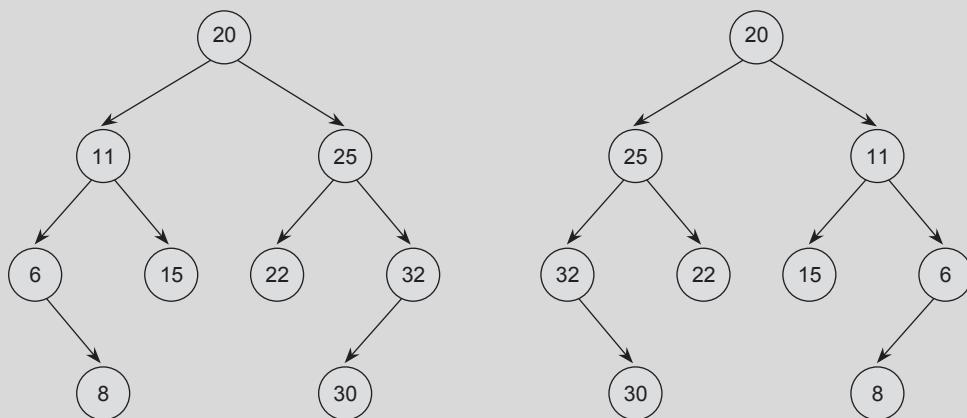
**20.5.** Dibujar los árboles binarios que representan las siguientes expresiones:

- $(A + B) / (C - D)$
- $A + B + C / D$
- $A - (B - (C - D)) / (E + F)$
- $(A + B) * ((C + D) / (E + F))$
- $(A - B) / ((C * D) - (E / F))$

**20.6.** El recorrido preorder de un cierto árbol binario produce ADFGHKL PQRWZ, y en recorrido inorden produce GFHKDLAWRQPZ. Dibujar el árbol binario.

## PROBLEMAS

- 20.1.** Codifique la clase `arbol` como clase amiga de una clase `Nodo` y que permita realizar métodos recursivos. La clase `árbol` debe contener las funciones `miembro` o `Obtener` y poner tanto el hijo izquierdo y derecho como el elemento de la raíz del árbol.
- 20.2.** Añadir a la clase `arbol` del ejercicio anterior el constructor de copia.
- 20.3.** Añadir a la clase `árbol` del Problema 20.1 una función miembro que copie el objeto actual en un objeto que reciba como parámetro.
- 20.4.** Sobrecargue el operador de asignación de la clase `arbol` para permitir asignar árboles completos.
- 20.5.** Escriba un método de la clase `arbol` que realice el espejo del objeto árbol. El espejo de un árbol es otro árbol que es el mismo que el que se ve si se reflejara en un espejo, tal y como se indica en el siguiente ejemplo:



- 20.6.** Escribir funciones miembro de la clase `árbol` del Problema 20.1 para hacer los recorridos recursivos en profundidad inversa, preorden, postorden.
- 20.7.** Escribir una función miembro de la clase `arbol` que nos cuente el número de nodos que tiene.
- 20.8.** Añadir a la clase `arbol` un método que calcule el número de hojas del objeto.
- 20.9.** Añadir a la clase `árbol` una función miembro que reciba como parámetro un número natural  $n$  que indique un nivel, y muestre todos los nodos del árbol que se encuentren en ese nivel.
- 20.10.** Construir una función miembro de la clase `arbol` para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado.
- 20.11.** Escribir una función miembro de la clase `arbol` recursiva y otra iterativa que se encargue de insertar la información dada en un elemento en un `árbol` binario de búsqueda.
- 20.12.** Escribir funciones miembros de la clase `arbol` iterativas y recursivas para borrar un elemento de un `árbol` binario de búsqueda.

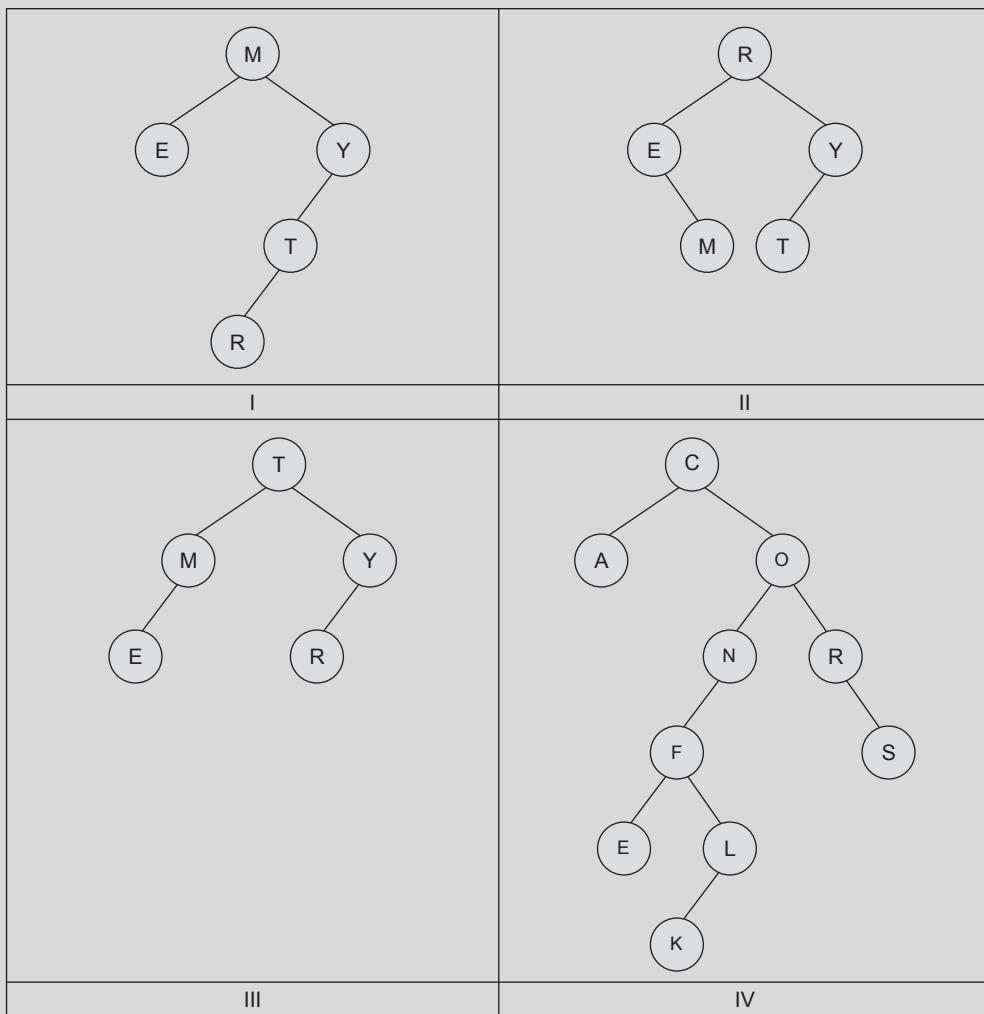
## SOLUCIÓN DE LOS EJERCICIOS

**20.1.** La primera estructura 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. Esta estructura es un árbol en general. La segunda no es un árbol binario porque hay dos caminos distintos para ir al nodo F y, por tanto, no se expresa la jerarquía de la definición de árbol un árbol en general y de un binario en particular. La tercera por la misma razón que la segunda no es un árbol binario.

- 20.2.**
- 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.**
- 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.

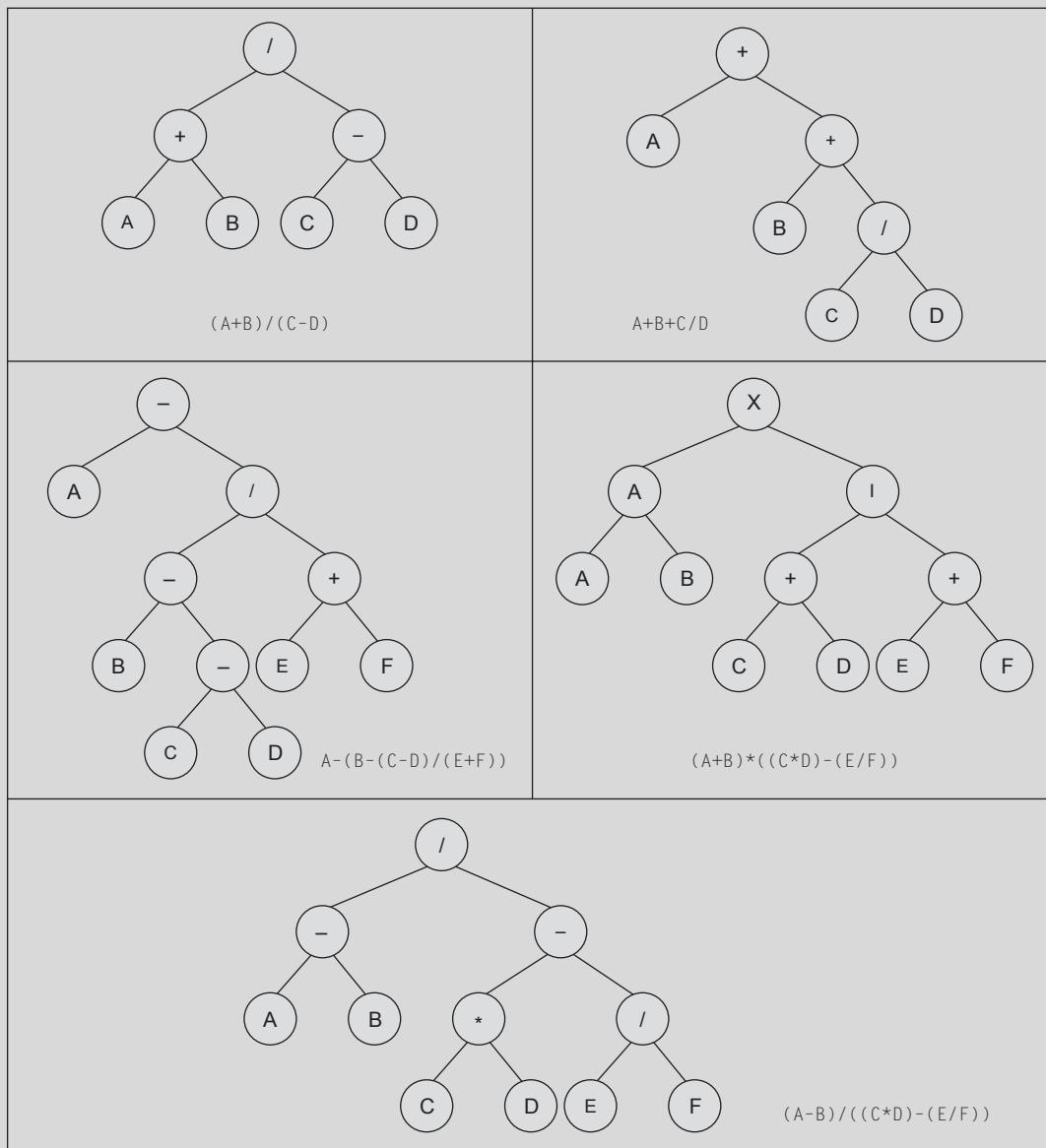
- 20.4.** Los gráficos solicitados en el apartado a son:



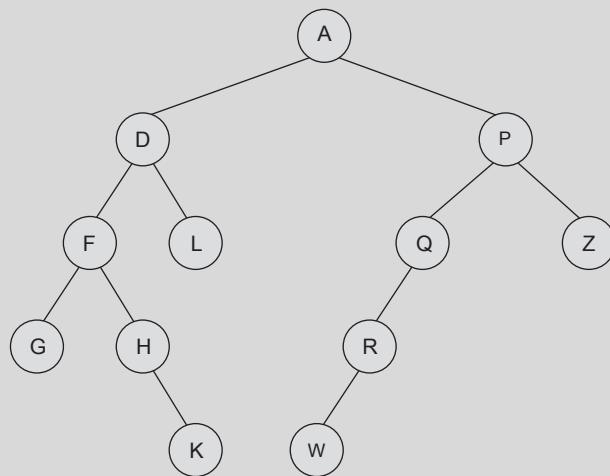
b) Los recorridos en inorder, preorder y postorden de cada uno de los árboles son:

|                                                                               |                                                                                                                            |
|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Inorden: E, M, R, T, Y<br>Preorden: M, E, Y, T, R<br>Postorden: E, R, T, Y, M | Inorden: E, M, R, T, Y<br>Preorden: R, E, M, Y, T<br>Postorden: M, E, T, Y, R                                              |
| I                                                                             | II                                                                                                                         |
| Inorden: E, M, T, R, Y<br>Preorden: T, M, E, Y, R<br>Postorden: E, M, R, Y, T | Inorden: A, C, E, F, K, L, N, O, R, S<br>Preorden: C, A, O, N, F, E, L, K, R, S<br>Postorden: A, E, K, L, F, N, S, R, O, L |
| III                                                                           | IV                                                                                                                         |

20.5. Los árboles de expresiones son los siguientes:



- 20.6.** Si se tiene el recorrido en preorden e inorden y además no hay claves repetidas, entonces se puede usar el siguiente algoritmo recursivo para resolver el problema. El primer elemento del preorden es siempre la raíz del árbol, y todos los elementos del árbol a construir se encuentran entre la posición izquierda = primera = 1 del inorden y la derecha = última. Se busca la posición  $i$  donde se encuentra la raíz del árbol en el recorrido en inorden. El subárbol izquierdo de la raíz ya encontrada (en el preorden) está entre las posiciones izquierda e  $i-1$  del recorrido en inorden (llamada recursiva), y el hijo derecho está entre las posiciones  $i+1$  y derecha del recorrido en inorden (llamada recursiva). En el ejemplo la raíz del árbol general es A, y su hijo izquierdo viene dado en el recorrido en inorden por los datos GFHKDL, y su hijo derecho por WRQPZ. El algoritmo va avanzando por el recorrido en preorden, ya que siempre se encuentra la raíz del árbol a construir en la siguiente posición, y termina cuando ya no quedan datos en el recorrido en preorden. En cada llamada recursiva hay dos posibilidades, que el árbol sea vacío, o que no lo sea. El árbol es vacío, si en el recorrido en inorden la izquierda y derecha se han cruzado, y hay datos si no se han cruzado.



PREORDEN: A, D, F, G, H, K, L, P, Q, R, W, Z  
 INORDEN: G, F, H, K, D, L, A, W, R, Q, P, Z

## SOLUCIÓN DE LOS PROBLEMAS

- 20.1.** Se declara el `Telemento` como un sinónimo del tipo de dato que almacena el árbol. En este caso un `entero`. Se avisa de la creación de la clase `Nodo`, para que la clase `Arbol` declare un puntero a la clase `Nodo` como protegido. La clase `Arbol` tiene como funciones miembro las necesarias para obtener y poner la raíz del árbol (en este caso es el elemento que almacena), y obtener y poner el hijo izquierdo y derecho, además del constructor y destructor de la clase `Arbol`. La clase `Nodo` almacena como atributos privados el hijo izquierdo y derecho del nodo del árbol así como el elemento que almacena, declarando a la clase `Arbol` como amiga para permitir acceder a los atributos de la clase `Nodo` y de esta forma sea más sencilla la codificación de las funciones miembro de la clase `Arbol`.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.2.** El constructor de copia se activa cada vez que se realiza una llamada a un objeto por valor. Este constructor copia realiza una copia del parámetro actual `arb`. La programación recursiva es tan sencilla como lo siguiente: si el árbol que se pasa está vacío (`arb.a = NULL`) el objeto actual se queda vacío; en otro caso, se crea un nuevo `Nodo` que es asignado al puntero `a`. Al hacer la llamada al constructor de `Nodo`, éste recibe como parámetros el hijo izquierdo y derecho del parámetro, por lo que se activa la copia recursivamente.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.3. Para copiar el objeto actual en un árbol que se pase por referencia, basta con crear un *Nodo* en el que se copie el elemento de la raíz del objeto, y como hijo izquierdo y derecho la copia del hijo izquierdo y derecho.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.4. La sobrecarga del operador de asignación se realiza con un código similar al del constructor de copia, excepto que se debe retornar el contenido del puntero *this*. Antes de realizar la copia se borra el árbol actual.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.5. Para realizar la función espejo, basta con crear un *Nodo* en el que se copia el elemento de la raíz del objeto, y como hijo izquierdo y derecho la copia del hijo derecho e izquierdo respectivamente.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.6. De los seis posibles recorridos en profundidad IDR,IRD,DIR,DRI,RID, RDI se pide IDR,IRD RID, que se programan en las siguientes funciones:

- *IRD o recorrido en Inorden.* Se recorre el hijo izquierdo, se visita la raíz y se recorre el hijo derecho. Por 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.
- *RID o recorrido en Preorden.* Se visita la raíz. Se recorre el hijo izquierdo, y después se recorre el hijo derecho. Por tanto, la codificación es análoga a la de Inorden, pero cambiando el orden de llamadas.
- *IDR o recorrido en Postorden.* Se recorre el hijo izquierdo, se recorre el hijo derecho, y posteriormente se visita la raíz.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.7. 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 la suma del número de los nodos que tenga su hijo izquierdo y su hijo derecho. La función miembro que resuelve el problema es *Cuantos*.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.8. 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.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.9. Si el árbol está vacío los nodos que hay en ese nivel es cero. En otro caso, hay que escribir el nodo si está en el nivel 1, y en caso de que no lo esté hay que calcular los nodos que hay en el nivel inmediatamente siguiente (estar uno al nivel que se busca) de su hijo izquierdo e hijo derecho.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.10. Basta con hacer un recorrido del árbol y escribir los datos que cumplan la condición dada.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.11. Se implementan mediante las funciones miembro *AnadeA* y *AnadeAI* 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. Si no está vacío, si la información almacenada en la raíz coincide con la que se está buscando habrá 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 se va a insertar y la información almacenada en el nodo.

- AnadeAI. Se usa un interruptor `enc` que se inicializa a `false`, un puntero a `Nodo pos` que se inicializa al primer nodo raíz del árbol, y mediante un bucle mientras no `enc` y no se haya terminado el árbol (`pos != NULL`) hacer: si coinciden los contenidos poner `enc` a `true`, en otro caso se desplaza a la izquierda o a la derecha quedándose con su anterior dependiendo del contenido de la raíz y del elemento que se esté buscando. Posteriormente, se inserta el nodo si la búsqueda terminó en fallo teniendo en cuenta que puede ser el r aíz total o bien un hijo izquierdo o bien un hijo.

**La codificación de este problema se encuentra en la página Web del libro.**

- 20.12.** El borrado que se implementa es el explicado en la teoría, usando el predecesor inmediato que se encuentra a uno a la izquierda y todo a su derecha. Las funciones que lo codifican son:

- BorrarA. Es un método público de la clase `Arbol`. 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 `BorP` que se encarga de buscar el predecesor inmediato, copia la información en el nodo que se quiere borrar, cambia el nodo a borrar que es el predecesor inmediato. Por último, se libera memoria.
- BorP. Es un método privado de la clase `Arbol`. Mediante llamadas recursivas hace lo indicado anteriormente y realiza el enlace con el hijo izquierdo del predecesor inmediato.
- BorrarAI. Realiza la búsqueda del nodo a borrar iterativamente. En el caso de éxito en la búsqueda considera los tres casos considerados en el `BorrarA`, 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 con la función `BorPI`. Una vez encontrado, se intercambian la información y se procede al borrado del nodo predecesor inmediato.
- BorPI. Es un método privado de la clase `Arbol`. Realiza la misma tarea que la función miembro `BorP`.

**La codificación de este problema se encuentra en la página Web del libro.**

## EJERCICIOS PROPUESTOS

- 20.1.** Para los árboles del Ejercicio resuelto 20.4, recorrer cada árbol utilizando los órdenes siguientes: RDI, RNI, RIN. si el árbol es completo y falso (`false`) en caso contrario.
- 20.2.** Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.
- 20.3.** Escribir una función booleana a la que se le pase un puntero a un árbol binario y de vuelva verdadero (`true`)
- 20.4.** Diseñar una función recursiva que devuelva un puntero a un elemento en un árbol binario de búsqueda.
- 20.5.** Diseñar una función iterativa que encuentre un elemento en un árbol binario de búsqueda.

## PROBLEMAS PROPUESTOS

**20.1.** 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.

**20.2.** Se dispone de un árbol binario de elementos de tipo entero. Escribir funciones que calculen:

- a) La suma de sus elementos.
- b) La suma de sus elementos que son múltiplos de 3.

**20.3.** Escribir una función booleana `IDENTICOS` que permita decir si dos árboles binarios son iguales.

**20.4.** 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 del archivo y los almacene 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 DE ACUERDO AL 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                     |

**20.5.** Diseñar un programa interactivo que permita dar altas, bajas, listar, etc., en un árbol binario de búsqueda.

**20.6.** Dados dos árboles binarios de búsqueda indicar mediante un programa si los árboles tienen o no elementos comunes.

**20.7.** 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 los correspondientes procedimientos y 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.)

**20.8.** Una matriz de N elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordenar ascendente la cadena de caracteres.

**20.9.** Dado un árbol binario de búsqueda, diseñar una función que liste los nodos del árbol ordenados descendente.

**20.10.** Escribir un programa C++ que lea una expresión correcta en forma infija y la presente en notación postfija.

**20.11.** 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.

**20.12.** Escribir funciones para recorrer un árbol en *inorden* y *preorden* iterativamente.

**20.13.** Escribir una función que realice el recorrido en anchura de un árbol binario.

**20.14.** Dado un árbol binario de búsqueda de claves enteras, diseñar una función que liste en orden creciente los nodos del árbol que cumplan la condición de ser capicúa.

# Índice

## A

Abstracción, 5  
Algoritmo, 1  
    burbuja, 248  
    concepto, 1  
Ámbito, 114  
    de programa, 114  
    de una función, 114  
    del archivo fuente, 114  
    global, 114  
    local, 114  
Apuntador (véase puntero), 203  
Árbol, 379  
    altura, 380  
    camino, 380  
    longitud, 380  
    completo, 380  
    definición, 379  
    equilibrado, 380  
    padre, 379  
    perfectamente, 380  
    hoja, 379  
    nivel, 380  
    profundidad, 380  
    recorridos, 383  
        *inorden*, 384  
        *postorden*, 384  
        *preorden*, 384  
    subárbol, 379  
    raíz, 379  
Árbol binario, 380  
    de búsqueda, 385  
    de expresión, 382  
    estructura, 381  
    representación, 381  
Árbol binario de búsqueda, 385  
    borrado, 386  
     inserción, 385  
Archivos, 311  
    apertura, 316  
    cabecera, 20  
Array, 135  
    argumentos, 141  
    caracteres, 137  
    declaración, 135  
    indexación basada en cero, 135  
     inicialización, 136

parámetros, 141  
    cadenas como parámetros, 145  
    matrices como parámetros, 144  
número de elementos, 138  
rango, 135  
subíndice, 135  
Arrays multidimensionales, 138  
    bidimensionales, 144  
     inicialización, 135  
    matrices, 144  
    tridimensionales, 140  
Arreglo, 135  
Asignación, 33  
Asociatividad, 45  
Atributos, 259  
Auto, 116

## B

Biblioteca `string.h`, 189  
Bit, 172  
`bool`, 23  
`break`, 62, 81  
Bucle, 77  
    anidados, 86  
    bandera, 80  
    centinela, 79  
    comparación, 84  
    controlados por contador, 78  
    infinito, 79, 80  
Búsqueda, 245  
    binaria, 247  
    algoritmo, 247  
    dicotómica, 245  
    lineal, 245  
    secuencial, 245  
        algoritmo, 245

## C

C++, 7  
Cadena, 183  
    declaración, 184  
     inicialización, 184  
    lectura, 183

Campos de bits, 172  
char, 21  
`cin`, 27  
    `cin.getline()`, 185  
    `cin.get()`, 185  
    `cin.putback()`, 188  
    `cin.ignore()`, 188  
Clase, 7, 354  
    abstracta, 286  
    arbol, 389  
    base, 7  
    cola, 354  
    compuesta, 268  
    concepto, 259  
    constructor, 264  
        de copia, 265  
        por defecto, 264  
definición, 259  
derivada, 7, 278  
destructor, 265  
inicialización, 265  
`ifstream`, 316  
`istream`, 312  
lista, 327  
pila, 348  
`ofstream`, 316  
`ostream`, 312  
nodo, 324  
subclase, 7  
superclase, 7  
`class`, 278  
Cola, 347, 353  
    clase, 354  
    especificación, 354  
    **FIFO**, 354  
    implementación, 354  
    con array circular, 354  
    con listas enlazadas, 356  
Comentarios, 18  
Constantes, 23  
    cadena, 23  
    carácter, 23  
    definidas, 24  
    declaradas, 24  
    enteras, 24  
    literales, 23  
    reales, 23  
    cons, 24

**D**

Datos ocultación, 6  
 Declaraciones globales, 11  
`define`, 113  
`delete`, 233  
 Diagrama, 1  
     de flujo, 1  
 Diseño, 3  
`do-while`, 77, 83, 84  
`double`, 21

**E**

Encapsulación, 6  
 Enumeración, 170  
`enum`, 170  
 Espacio de nombres, 28  
 Especialización 6  
 Estructuras, 161  
     acceso, 163  
     anidadas, 165  
     asignación, 165  
     declaración, 161  
     tamaño, 164  
 Evaluación en cortocircuito, 64  
 Estructuras de control, 57  
     iterativas, 77  
     repetitivas, 77  
 Expresiones, 35  
     aritmética, 36  
     condicional, 63  
     lógica, 64  
 Excepción, 296  
     diseño, 297  
     lanzamiento, 296  
     manejador, 297  
`extern`, 116

**F**

`false`, 23  
**FIFO**, 354  
`float`, 21  
 Flujo, 311  
     clases, 311  
`for`, 77, 81, 84  
     bucles, 77  
 Función, 103  
     alcance, 114  
     anidada, 104  
     carácter, 119  
     definición, 114  
     definidas por el usuario, 18  
     declaración en línea, 112  
     de fecha y hora, 121  
     estructura, 104  
     matemáticas, 104  
     numéricas, 120  
     plantillas, 123  
     prototipos, 106

**G**

parámetros, 108  
     argumentos por omisión, 111  
     `const`, 110  
     por referencia, 110  
     por valor, 108  
 recursivas, 368  
 sobrecarga, 121  
 utilidad, 121  
 virtuales, 287  
 visibilidad, 121

**G**

Generalización, 6  
 Genericidad, 293

**H**

*Heap*, 227  
 Herencia, 6, 277  
     múltiple, 282  
     privada, 279  
     protegida, 279  
     simple, 280  
         pública, 279  
         tipos, 280

**I**

Identificador, 19  
`if`, 57  
`if-else`, 59  
     anidados, 61  
`include`, 15  
`inline`, 112  
`int`, 20

**K**

**Knuth**, 1

**L**

**LIFO**, 347  
 Listas, 327  
     clasificación, 324  
     contiguas, 327  
     declaración, 327  
 Listas enlazadas, 323  
     clase, 327  
     implementación, 324  
     operaciones, 325  
         insertar 328  
         buscar, 330  
         eliminar, 331  
 Lista circular, 336  
     inserción, 337  
     eliminación, 337

**L**

Lista doblemente enlazada, 332  
     clase, 334  
     eliminación, 335  
     inserción, 334  
 Llamada por referencia, 108  
 Llamada por valor, 109

**M**

`main()`, 17  
 Memoria, 227  
     dinámica, 227  
 Métodos, 259  
 Módulo (o función), 103

**N**

`namespace`, 28  
`new`, 228  
 Nodo, 325  
`NULL`, 206

**O**

Objetos, 7, 259  
     estado, 261  
     métodos, 261  
     constructor, 264  
     destructor, 265  
 Ocultación, 7  
 Operador, 35  
     de inserción, 16  
     de extracción, 27  
     ?, 42, 63  
     (), 44  
     [], 44  
     +, 36  
     ++, 36  
     --, 33  
     =, 33  
     ==, 40  
     ->, 42, 164  
     ., 42, 164  
     ::, 44  
     acceso, 42, 164  
     aritméticos, 36  
     asociatividad, 46  
     coma, 43  
     de bits, 41  
     de dirección, 42  
     de incrementación, 34  
     de decrementación, 39  
     de manipulación de bits, 41  
     de desplazamiento, 41  
     lógicos, 40  
         evaluación cortocircuito, 40, 46  
         prioridad, 46  
     molde, 45  
     relacionales, 39  
     `sizeof`, 293  
     sobrecarga, 293

Ordenación, 245  
 burbuja, 248  
 inserción, 251  
 lineal, 251  
 binaria, 252  
 selección 250  
 shell, 253

**P**

Palabras reservadas, 19  
 Parámetros, 108  
 argumentos por omisión, 111  
*const*, 110  
*arrays*, 141  
 cadena, 145  
 por referencia, 109  
 por valor, 109  
 Pila, 347  
 clase, 348  
 definición, 347  
 especificación, 348  
 implementación con array, 348  
 implementación con listas enlazadas, 351  
 insertar, 347  
 Plantillas, 293  
 de clases, 294  
 de funciones, 123, 293  
 Polimorfismo, 7, 121, 277, 278  
 Prioridad, 45  
 Programa, 15  
 conceptos básicos, 15  
 elementos de un Programa C++, 19  
 estructura general, 2, 15  
 Programación, 1  
 estructurada, 3  
 orientada a objetos, 1, 11  
 Prototipo, 116  
 Pseudocódigo, 1  
 Puntero, 203  
 a constante, 209  
 a función, 213  
 a estructuras, 216

concepto, 203  
 constante, 209  
 declaración, 203  
 inicialización, 203  
 paso con parámetros, 212  
 y array, 206  
 y array de dos dimensiones, 210

**R**

Recursión, 367  
 Recursividad, 367  
 condición de terminación, 368  
 directa, 368  
 indirecta, 368  
 infinita, 370  
 versus iteración, 370  
*register*, 116  
 Registro, 161  
 Repetición, 77  
*return*, 104

**S**

Sentencias  
 asignación, 35  
*do while*, 77, 83, 84  
*for*, 77, 81, 84  
*if*, 9  
 if anidados, 61  
*switch*, 62  
*while*, 77, 84  
 Separador, 20  
 Signos de puntuación, 20  
*sizeof*, 44  
 Sobrecarga, 293  
 asignación, 305  
 de funciones, 121  
 de operadores, 293  
 unitarios, 300  
 binarios, 303  
*static*, 117

*struct*, 161  
*switch*, 62  
 etiquetas *case*, 62  
 etiquetas *default*, 62

**T**

Tablas, 135, 138  
 TAD, 5  
 Tipo abstracto de datos, 1, 5  
 Tipos de datos, 20  
 básicos, 20  
 rango de valores, 20  
 numéricos, 20  
*true*, 23  
*typedef*, 172

**U**

*Union*, 161, 168  
 definición, 168

**V**

Variables, 25  
 declaración, 25  
 definición, 25  
 de objetos, 26  
 dinámica, 26  
 duración, 25  
 estáticas, 117  
 externas, 116  
 globales, 26  
 inicialización, 25  
*void*, 206  
*volatile*, 24

**W**

*while*, 77, 84

# ***¡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!***



[www.mhe.es/joyanes](http://www.mhe.es/joyanes)

[www.mcgraw-hill.es](http://www.mcgraw-hill.es)

*The McGraw-Hill Companies*